

3.0

TURBO C[®]++

USER'S GUIDE

B O R L A N D

Turbo C[®]++

Version 3.0

User's Guide

BORLAND INTERNATIONAL, 100 BORLAND WAY
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-3249, USA

Copyright © 1992 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Windows, as used in this manual, refers to Microsoft's implementation of a windows system.

C O N T E N T S

Introduction	1	The /h option	21
What's in Turbo C++	1	The /l option	22
Hardware and software requirements	3	The /m option	22
The Turbo C++ implementation	3	The /p option	22
The Turbo C++ package	3	The /r option	22
The <i>User's Guide</i>	4	The /s option	22
Online documentation	5	The /x option	22
Using the manual	6	Exiting Turbo C++	23
Programmers learning C or C++	6	IDE components	23
Typefaces and icons used in these books	6	The menu bar and menus	23
How to contact Borland	7	Shortcuts	24
Resources in your package	7	Command sets	24
Borland resources	8	Native option	27
Part 1 Using Turbo C++		Turbo C++ windows	27
Chapter 1 Installing Turbo C++	13	Window management	29
Using INSTALL	14	The status line	30
Protected mode and memory	14	Dialog boxes	31
DPMIINST	15	Action buttons	31
DPMIMEM	15	Radio buttons and check boxes	31
DPMIRES	16	Input and list boxes	32
Extended and expanded memory	16	Configuration and project files	33
Running Turbo C++	17	The configuration file	33
Laptop systems	17	Project files	34
The README file	17	The project directory	34
The FILELIST.DOC and HELPME!.DOC	17	Desktop files	35
files	17	Changing project files	35
Example programs	18	Default files	35
Customizing the IDE	18	IDE menus	36
Chapter 2 IDE basics	19	Syntax highlighting	37
Starting and exiting	20	IDE cross-reference	37
Command-line options	20	Chapter 3 An introduction to C++	43
The /b option	20	How to run the examples	44
The /d option	21	Basic programming operations	44
The /e option	21	Basic structure of a C++ program	48
		Working with numbers	50

Numeric data types	50	The function prototype	88
Integers	52	Function declarations under	
Integer modifiers	53	Kernighan and Ritchie	89
The long modifier	53	The function definition	89
The signed and unsigned		Processing within the function	90
modifiers	53	The function return value	90
Floating-point numbers	54	Using the return value	90
The floating-point types	55	Multifunction programs	91
Variables	55	Function prototypes and global	
Initializing variables	55	declarations	94
Assignment statements	56	Setting up the graphics display	94
Combination assignments	57	Calculating the graphics	
Variable names	57	coordinates	95
More about input and output	58	Drawing the planets	95
Formatting with escape sequences	58	Header files, functions, and libraries	96
Arithmetic operators	60	Scope and duration of variables	99
Arithmetic and type conversion	61	Scope	99
Typecasting	62	Duration	101
Combining arithmetic and assignment	63	Using constant values	102
Increment and decrement	63	Building data structures	103
Working bit by bit	64	Declaring and initializing an array	104
Expressions	66	Arrays with multiple dimensions	106
Evaluating an expression	66	Arrays and strings	108
Assigning a value in an expression	68	Renaming types	109
Characters and strings	69	Enumerated types	110
Input and output for single characters	69	Combining data into structures	111
Displaying a character	71	Using parts of a structure	111
Displaying character strings	71	Pointers	112
Testing conditions and making choices	72	Declaring and using a pointer	113
Using relational operators	72	Pointers and strings	115
Using logical operators	74	Pointer arithmetic	116
Branching with if and if...else	74	Pointers, structures, and lists	116
Multiple choices with if...else	75	Using pointers to return values from	
Multiple choice tests: switch	77	functions	119
Repeating execution with loops	79	Using system resources	121
The while loop	80	Opening a stream	124
The do while loop	81	Chapter 4 Object-oriented	
The for loop	82	programming with C++	125
break and continue	84	Encapsulation	127
Nested loops	86	Inheritance	130
Choosing appropriate loops	87	Polymorphism	132
Program design with functions and		Overloading	132
macros	87	Modeling the real world with classes	133
Defining your own functions	88		

Building classes: a graphics example .	133	Where to now?	197
Declaring objects	135	Conclusion	198
Member functions	135	Chapter 5 Hands-on C++	199
Calling a member function	136	A better C: Making the transition from	
Constructors and destructors	137	C	200
Code and data together	140	Program 1	200
Member access control: private , public ,		Program 2	201
and protected	140	Program 3	201
The class: private by default	141	Program 4	202
Running a C++ program	142	Object support	203
Inheritance	144	Program 5	204
Rethinking the Point class	145	Program 6	206
Inheritance and access control	146	Program 7	209
Packaging classes into modules	148	Program 8	210
Extending classes	151	Program 9	212
Multiple inheritance	155	Summary	215
virtual functions	160	Chapter 6 Debugging in the IDE	217
virtual functions in action	162	Debugging and program development .	218
Defining virtual functions	163	Designing the example program:	
Developing a complete graphics		PLOTEMP.C	220
module	164	Writing the prototype program	222
Reference types	165	Using the integrated debugger	223
Ordinary or virtual member		Tracing the flow of a program	224
functions?	172	Tracing high-level execution	224
Dynamic objects	172	Tracing into called functions	225
Destructors and delete	174	Continuing program development	226
An example of dynamic object		Setting breakpoints	227
allocation	174	Instant breaking with <i>Ctrl-Break</i>	228
More flexibility in C++	179	Inspecting your data	229
Inline functions outside class		Inspector windows	229
definitions	179	Inspecting arrays and strings	230
Functions with default arguments ...	180	Inspecting structs and unions	230
More about overloading functions ...	181	Inspecting pointers	231
Overloading operators to provide new		Inspecting functions	231
meanings	184	When should you use inspectors? ...	231
friend functions	187	Evaluating and changing variables ...	232
The C++ streams libraries	188	Specifying display format	233
Standard I/O	189	Specifying the number of values ...	233
Formatted output	191	Copying from the cursor position ...	234
Manipulators	192	Specifying variables in other	
put, write, and get	192	functions	234
Disk I/O	193	Changing values	234
I/O for user-defined data types	196		

Monitoring your program by setting watches	237	Using the options	266
Adding a watch	237	Option precedence rules	267
Watching your watches	238	Syntax and file names	270
Controlling the debugger windows ..	239	Response files	271
Editing and deleting watches	239	Configuration files	271
Finding a function definition	240	Option precedence rules	272
Finding out who called whom	240	Compiler options	272
Multiple source files	241	Memory model	273
Preventive medicine	241	Macro definitions	274
Design defensively	241	Code-generation options	275
Write clearly	242	The -v and -vi options	278
Systematic software testing	242	Optimization options	279
Test modifications thoroughly	243	Source code options	281
Areas to watch carefully	243	Error-reporting options	282
Finishing PLOTEMP.C	244	ANSI violations	282
Finishing table_view	245	Frequent errors	283
Implementing graph_view	246	Portability warnings	283
save_temps and read_temps	247	C++ warnings	283
Answers to debugging exercises	248	Segment-naming control	284
min_max and avg_temps	248	Compilation control options	285
graph_view	249	EMS and expanded memory options ..	287
save_temps	249	C++ virtual tables	287
read_temps	249	C++ member pointers	289
Advanced options	250	Template generation options	290
Chapter 7 Managing multi-file projects	251	Linker options	290
Sampling the project manager	252	Environment options	291
Error tracking	255	Backward compatibility options	292
Stopping a make	255	Searching for include and library files	293
Syntax errors in multiple source files ..	256	File-search algorithms	294
Saving or deleting messages	257	An annotated example	295
Autodependency checking	257	Chapter 9 MAKE: The program manager	297
Using different file translators	258	How MAKE works	297
Overriding libraries	260	Starting MAKE	298
More Project Manager features	260	Command-line options	299
Notes for your project	263	The BUILTINS.MAK file	301
Chapter 8 The command-line compiler	265	A simple use of MAKE	301
Using the command-line compiler	265	Creating makefiles	303
DPMIINST	266	Components of a makefile	304
Running TCC	266	Comments	304
		Command lists for implicit and explicit rules	305

Prefixes	305	Macro undefinition directive	329
Command body and operators	305	The compatibility option -N	329
Compatibility option	307	Chapter 10 TLINK: The Turbo linker	331
Batching programs	307	Invoking TLINK	331
Executing commands	308	An example of linking	332
Explicit rules	309	File names on the TLINK command	
Special considerations	310	line	332
Multiple explicit rules for a single		Using response files	333
target	311	The TLINK configuration file	334
Examples	311	Using TLINK with Turbo C++	
Automatic dependency checking ..	312	modules	334
Implicit rules	312	Startup code	335
Macros	315	Libraries	336
Defining macros	316	BGI graphics library	336
Using macros	316	Math libraries	336
Using environment variables as		Run-time libraries	337
macros	316	Using TLINK with TCC	337
Substitution within macros	317	TLINK options	338
Special considerations	317	The TLINK configuration file	338
Predefined macros	318	/3 (32-bit code)	338
Defined Test Macro (\$d)	318	/c (case sensitivity)	338
File name macros	319	/d (duplicate symbols)	339
Base file name macro (\$*)	319	/e (no extended dictionary)	339
Full file name macro (\$<)	319	/i (uninitialized trailing segments) ..	340
File name path macro (\$:)	320	/l (line numbers)	340
File name and extension macro		/L (library search paths)	340
(\$.)	320	/m, /s, and /x (map options)	340
File name only macro (\$&)	320	/n (ignore default libraries)	342
Full target name with path macro		/o (overlays)	342
(\$@)	320	/t (tiny model .COM file)	343
All dependents macro (\$**)	321	/Td	343
All out of date dependents macro		/v (debugging information)	344
(\$?)	321	/ye (expanded memory)	344
Macro modifiers	321	/yx (extended memory)	345
Directives	322	Part 2 Programming Reference	
Dot directives	323	Chapter 11 Lexical elements	349
.precious	323	Whitespace	350
.path.ext	323	Line splicing with \	350
.suffixes	324	Comments	351
File-inclusion directive	324	C comments	351
Conditional execution directives	325	Nested comments	351
Expressions allowed in conditional		C++ comments	352
directives	327		
Error directive	328		

Comment delimiters and whitespace	352	Lvalues	370
Tokens	352	Rvalues	371
Keywords	353	Types and storage classes	371
Identifiers	354	Scope	371
Naming and length restrictions ...	354	Block scope	372
Identifiers and case sensitivity ...	354	Function scope	372
Uniqueness and scope	355	Function prototype scope	372
Constants	355	File scope	372
Integer constants	355	Class scope (C++)	372
Decimal constants	355	Scope and name spaces	372
Octal constants	356	Visibility	373
Hexadecimal constants	357	Duration	373
long and unsigned suffixes ...	357	Static duration	373
Character constants	358	Local duration	374
Escape sequences	358	Dynamic duration	374
Turbo C++ special two-character constants	359	Translation units	375
signed and unsigned char	359	Linkage	375
Wide character constants	360	Name mangling	376
Floating-point constants	360	Declaration syntax	377
Floating-point constants — data types	360	Tentative definitions	377
Enumeration constants	361	Possible declarations	378
String literals	362	External declarations and definitions	380
Constants and internal representation	363	Type specifiers	382
Constant expressions	364	Type taxonomy	382
Punctuators	365	Type void	383
Brackets	365	The fundamental types	383
Parentheses	365	Integral types	384
Braces	365	Floating-point types	385
Comma	366	Standard conversions	385
Semicolon	366	Special char , int , and enum conversions	386
Colon	367	Initialization	386
Ellipsis	367	Arrays, structures, and unions ...	387
Asterisk (pointer declaration) ...	367	Simple declarations	388
Equal sign (initializer)	368	Storage class specifiers	389
Pound sign (preprocessor directive)	368	Use of storage class specifier auto .	389
Chapter 12 Language structure	369	Use of storage class specifier extern	389
Declarations	369	Use of storage class specifier register	389
Objects	369	Use of storage class specifier static	390
		Use of storage class specifier typedef	390
		Modifiers	391

The const modifier	391	Unary operators	425
The interrupt function modifier ...	393	Binary operators	425
The volatile modifier	393	Additive operators	425
The cdecl and pascal modifiers ...	394	Multiplicative operators	425
pascal	394	Shift operators	425
cdecl	394	Bitwise operators	425
The pointer modifiers	395	Logical operators	425
Function type modifiers	396	Assignment operators	425
Complex declarations and		Relational operators	426
declarators	397	Equality operators	426
Pointers	398	Component selection operators ...	426
Pointers to objects	399	Class-member operators	426
Pointers to functions	399	Conditional operator	426
Pointer declarations	400	Comma operator	426
Pointers and constants	401	Postfix and prefix operators	426
Pointer arithmetic	402	Array subscript operator []	426
Pointer conversions	403	Function call operators ()	427
C++ reference declarations	403	Structure/union member operator	
Arrays	403	.(dot)	427
Functions	404	Structure/union pointer	
Declarations and definitions	404	operator ->	427
Declarations and prototypes	405	Postfix increment operator ++	427
Definitions	407	Postfix decrement operator --	428
Formal parameter declarations	408	Increment and decrement operators .	428
Function calls and argument		Prefix increment operator	428
conversions	408	Prefix decrement operator	428
Structures	409	Unary operators	428
Untagged structures and typedefs ...	410	Address operator &	429
Structure member declarations	410	Indirection operator *	429
Structures and functions	411	Unary plus operator +	430
Structure member access	411	Unary minus operator -	430
Structure word alignment	413	Bitwise complement operator ~ ...	430
Structure name spaces	413	Logical negation operator !	430
Incomplete declarations	414	The sizeof operator	430
Bit fields	414	Multiplicative operators	431
Unions	415	Additive operators	432
Anonymous unions (C++ only)	416	The addition operator +	432
Union declarations	417	The subtraction operator -	432
Enumerations	417	Bitwise shift operators	433
Expressions	419	Bitwise shift operators (<< and >>) .	433
Expressions and C++	422	Relational operators	433
Evaluation order	422	The less-than operator <	434
Errors and overflows	423	The greater-than operator >	434
Operators	423		

The less-than or equal-to operator		
<=	434	
The greater-than or equal-to operator >=	434	
Equality operators	435	
The equal-to operator ==	435	
The inequality operator !=	436	
Bitwise AND operator &	436	
Bitwise exclusive OR operator ^	436	
Bitwise inclusive OR operator	437	
Logical AND operator &&	437	
Logical OR operator	437	
Conditional operator ?:	438	
Assignment operators	439	
The simple assignment operator =	439	
The compound assignment operators	439	
Comma operator	440	
C++ operators	440	
Statements	441	
Blocks	441	
Labeled statements	442	
Expression statements	442	
Selection statements	443	
if statements	443	
switch statements	444	
Iteration statements	444	
while statements	444	
do while statements	445	
for statements	445	
Jump statements	446	
break statements	446	
continue statements	447	
goto statements	447	
return statements	447	
Chapter 13 C++ specifics	449	
Referencing	449	
Simple references	450	
Reference arguments	450	
Scope access operator	452	
The new and delete operators	452	
Handling errors	453	
The operator new with arrays	453	
The operator delete with arrays	454	
The ::operator new	454	
Initializers with the new operator	454	
Classes	455	
Class names	455	
Class types	455	
Class name scope	456	
Class objects	457	
Class member list	457	
Member functions	457	
The keyword this	457	
Inline functions	458	
Static members	459	
Member scope	460	
Nested types	461	
Member access control	462	
Base and derived class access	464	
virtual base classes	466	
friends of classes	466	
Constructors and destructors	468	
Constructors	469	
Constructor defaults	470	
The copy constructor	471	
Overloading constructors	472	
Order of calling constructors	472	
Class initialization	474	
Destructors	476	
When destructors are invoked	477	
atexit() , #pragma exit , and destructors	477	
exit() and destructors	477	
abort() and destructors	478	
virtual destructors	478	
Operator overloading	479	
Overloaded operators and inheritance	481	
Operators new and delete	481	
Unary operators	482	
Binary operators	483	
The assignment operator=()	484	
The function call operator()	484	
The subscript operator	484	
The class member access operator	485	
virtual functions	485	

Abstract classes	487	#pragma hdrstop	517
C++ scope	488	#pragma inline	517
Class scope	488	#pragma option	517
Hiding	488	#pragma saveregs	518
C++ scoping rules summary	489	#pragma warn	519
Templates	490	Predefined macros	519
Function templates	490	__CDECL__	519
Overriding a template function	492	__cplusplus	520
Implicit and explicit template		__DATE__	520
functions	492	__FILE__	520
Class templates	493	__LINE__	520
Arguments	494	__MSDOS__	521
Angle brackets	495	__OVERLAY__	521
Type-safe generic lists	495	__PASCAL__	521
Eliminating pointers	496	__STDC__	521
Template compiler switches	497	__TCPLUSPLUS__	521
Using template switches	498	__TEMPLATES__	521
		__TIME__	521
		__TURBOC__	522
Chapter 14 The preprocessor	501	Chapter 15 The main function	523
Null directive #	503	Arguments to main	523
The #define and #undef directives	503	An example program	524
Simple #define macros	503	Wildcard arguments	525
The #undef directive	504	An example program	526
The -D and -U options	505	Using -p (Pascal calling conventions)	527
The Define option	505	The value main returns	527
Keywords and protected words	506		
Macros with parameters	506	Chapter 16 Using C++ streams	529
File inclusion with #include	509	What is a stream?	529
Header file search with <header_name>		The iostream library	530
.....	510	The streambuf class	530
Header file search with "header_name"		The ios class	530
.....	510	Output	531
Conditional compilation	510	Fundamental types	532
The #if, #elif, #else, and #endif conditional		I/O formatting	532
directives	511	Manipulators	533
The operator defined	511	Filling and padding	534
The #ifdef and #ifndef conditional		Input	535
directives	512	I/O of user-defined types	536
The #line line control directive	513	Simple file I/O	537
The #error directive	514	String stream processing	538
The #pragma directive	515	Screen output streams	539
#pragma argsused	515	Stream class reference	541
#pragma exit and #pragma startup	515		
#pragma hdrfile	516		

conbuf	541	Paths for .h and .LIB files	562
Member functions	541	MAKE	563
constream	543	Command-line compiler	563
Member functions	543	Command-line options and libraries .	565
filebuf	543	Linker	565
Data members	543	Source-level compatibility	566
Member functions	544	__MSC macro	566
fstream	545	Header files	567
Member functions	545	Memory models	567
fstreambase	545	Keywords	568
Member functions	546	Floating-point return values	568
ifstream	546	Structures returned by value	569
Member functions	547	Conversion hints	569
ios	547	Chapter 18 Memory management	571
Data members	547	Running out of memory	571
Member functions	549	Memory models	571
iostream	551	The iAPx86 registers	572
iostream_withassign	551	General-purpose registers	572
Member functions	551	Segment registers	573
istream	551	Special-purpose registers	573
Member functions	551	The flags register	573
istream_withassign	553	Memory segmentation	574
Member functions	553	Address calculation	575
istrstream	553	Pointers	576
ofstream	553	Near pointers	576
Member functions	554	Far pointers	576
ostream	554	Huge pointers	577
Member functions	554	The six memory models	578
ostream_withassign	555	Mixed-model programming: Addressing	
Member functions	555	modifiers	582
ostrstream	555	Segment pointers	583
Member functions	556	Declaring far objects	584
streambuf	556	Declaring functions to be near or far .	585
Member functions	556	Declaring pointers to be near, far, or	
strstreambase	558	huge	586
Member functions	559	Pointing to a given segment:offset	
strstreambuf	559	address	586
Member functions	559	Using library files	586
strstream	560	Linking mixed modules	586
Member function	560	Overlays (VROOMM)	587
Chapter 17 Converting from Microsoft		How overlays work	588
C	561	Getting the best out of Turbo C++	
Environment and tools	561	overlays	589

Requirements	590	An example	611
Using overlays	590	The <i>text_modes</i> type	611
Overlay example	591	Text colors	612
Overlaying in the IDE	591	High-performance output	613
Overlaid programs	592	Programming in graphics mode	614
The far call requirement	592	The graphics library functions	615
Buffer size	592	Graphics system control	615
What not to overlay	593	A more detailed discussion	617
Debugging overlays	593	Drawing and filling	617
External routines in overlays	593	Manipulating the screen and	
Swapping	594	viewport	619
Expanded memory	594	Text output in graphics mode	620
Extended memory	595	Color control	622
Chapter 19 Mathematical		Pixels and palettes	622
operations	597	Background and drawing color	623
Floating-point options	597	Color control on a CGA	623
Emulating the 80x87 chip	598	CGA low resolution	623
Using 80x87 code	598	CGA high resolution	624
No floating-point code	598	CGA palette routines	625
Fast floating-point option	598	Color control on the EGA and	
The 87 environment variable	599	VGA	625
Registers and the 80x87	600	Error handling in graphics mode ..	625
Disabling floating-point exceptions ..	600	State query	626
Using complex math	601	Appendix A Editor reference	629
Using BCD math	602	Block commands	631
Converting BCD numbers	603	Other editing commands	633
Number of decimal digits	603	Appendix B Precompiled headers	635
Chapter 20 Video functions	605	How they work	635
Some words about video modes	605	Drawbacks	636
Some words about windows and		Using precompiled headers	636
viewports	606	Setting file names	637
What is a window?	606	Establishing identity	637
What is a viewport?	607	Optimizing precompiled headers	637
Coordinates	607	Appendix C Error messages	639
Programming in text mode	607	Finding a message in this	
The console I/O functions	607	appendix	639
Text output and manipulation	608	Types of messages	640
Window and mode control	609	Compile-time messages	640
Attribute control	609	DPMI server messages	641
State query	610	MAKE messages	641
Cursor shape	610	Run-time error messages	642
Text windows	610	TLIB messages	642

TLINK messages 642
Message explanations 643

Index 713

T A B L E S

2.1: General hot keys	25	11.6: Turbo C++ integer constants without L or U	357
2.2: Menu hot keys	25	11.7: Turbo C++ escape sequences	359
2.3: Editing hot keys	26	11.8: Turbo C++ floating constant sizes and ranges	361
2.4: Window management hot keys	26	11.9: Data types, sizes, and ranges	363
2.5: Online Help hot keys	26	12.1: Turbo C++ declaration syntax	379
2.6: Debugging/Running hot keys	27	12.2: Turbo C++ declarator syntax	380
2.7: Manipulating windows	29	12.3: Turbo C++ class declarations (C++ only)	381
2.8: IDE overview	36	12.4: Declaring types	383
2.9: IDE menu cross-reference	37	12.5: Integral types	384
3.1: Data types, sizes, and ranges	52	12.6: Methods used in standard arithmetic conversions	386
3.2: Character escape sequences	59	12.7: Turbo C++ modifiers	391
3.3: Type promotions for arithmetic	62	12.8: Complex declarations	398
3.4: Bit manipulation operators	64	12.9: External function definitions	407
3.5: Precedence and associativity of operators	68	12.10: Associativity and precedence of Turbo C++ operators	420
3.6: Relational operators	73	12.11: Turbo C++ expressions	421
3.7: Logical operators	74	12.12: Bitwise operators truth table	436
3.8: Preopened streams in Turbo C++	122	12.13: Turbo C++ statements	441
4.1: Class access	147	14.1: Turbo C++ preprocessing directives syntax	502
8.1: Command-line options summary	267	16.1: Stream manipulators	534
9.1: MAKE options	300	16.2: Console stream manipulators	540
9.2: MAKE prefixes	305	17.1: CL and TCC options compared	564
9.3: MAKE predefined macros	318	17.2: LINK and TLINK options compared	566
9.4: MAKE filename macros	318	18.1: Memory models	582
9.5: MAKE macro modifiers	322	18.2: Pointer results	583
9.6: MAKE directives	322	20.1: Graphics mode state query functions	627
9.7: MAKE operators	327	A.1: Editing commands	629
10.1: TLINK options	331	A.2: Block commands in depth	632
10.2: .OBJ and .LIB files	337	A.3: Borland-style block commands	633
10.3: TLINK overlay options	343		
11.1: All Turbo C++ keywords	353		
11.2: Turbo C++ extensions to C	353		
11.3: Keywords specific to C++	353		
11.4: Turbo C++ register pseudovariables	354		
11.5: Constants—formal definitions	356		

A.4: Other editor commands in depth . . .	633	C.3: TLIB message variables	642
C.1: Compile-time message variables . . .	641	C.4: TLINK error message variables	643
C.2: MAKE error message variables	642		

F I G U R E S

2.1: A typical window	28	6.3: Inspecting the <i>temps</i> array	230
2.2: A typical status line	31	6.4: Inspecting the <i>min_max</i> function ...	231
2.3: A sample dialog box	31	10.1: Detailed map of segments	341
3.1: Interpreting memory locations as numbers (in 1-byte increments)	51	11.1: Internal representations of data types	364
3.2: How a string is stored in memory ...	71	16.1: Class streambuf and its derived classes	530
3.3: Information flow to and from the tax function	90	16.2: Class ios and its derived classes ...	531
3.4: Simple program structure (all in one) .	97	18.1: iAPx86 registers	572
3.5: Program built from several files	98	18.2: Flags register of the iAPx86	574
3.6: Program using custom libraries	99	18.3: Tiny model memory segmentation .	579
3.7: Two ways to deal with sets of data ..	104	18.4: Small model memory segmentation	580
3.8: How pointers point (and what they point to)	115	18.5: Medium model memory segmentation	580
3.9: Using pointers to access an array of structures	118	18.6: Compact model memory segmentation	580
3.10: Using pointers in a function	121	18.7: Large model memory segmentation	581
4.1: Traditional C versus encapsulated C++	130	18.8: Huge model memory segmentation	581
4.2: A partial taxonomy chart of insects .	131	18.9: Memory maps for overlays	589
4.3: Multiple inheritance	156	20.1: A window in 80x25 text mode	611
4.4: Circles with messages	160		
6.1: Program development flowchart	220		
6.2: Graph view of temperature data	222		

I N T R O D U C T I O N

Turbo C++ is a powerful compiler for beginner and experienced C++ and C programmers. With Turbo C++, you get both C++ (AT&T v.2.1 compliant) *and* ANSI C. It is a powerful, fast, and efficient compiler for creating practically any application.

C++ is an object-oriented programming (OOP) language that allows you to take advantage of OOP's advanced design methodology and labor-saving features. It's the next step in the natural evolution of C. C++ application programs are portable, so you can easily transfer them from one system to another. C++ is suitable for almost any programming task.

What's in Turbo C++

Chapter 1 tells you how to install Turbo C++. This Introduction tells you where you can find out more about each of these features.

Turbo C++ includes the latest features programmers have asked for:

- **C and C++:** Turbo C++ offers you the full power of C and C++ programming, with a complete implementation of the AT&T v. 2.1 and ANSI C specifications. Turbo C++ 3.0 also provides a number of useful C++ class libraries, plus the first complete commercial implementation of templates. With templates, efficient collection classes can be built using parameterized types.
- **Faster compilation speed:** Typically, Turbo C++ 3.0 cuts compilation time for C++ in half compared to previous versions of the product. Precompiled headers, a Borland exclusive, significantly shorten recompilation time.
- **DPMI Compiler:** Turbo C++ compiles huge programs of a size limited only by the memory on your system. Turbo C++ 3.0 now uses the industry-standard DPMI (DOS Protected Mode

Interface) protocol that runs the compiler (as well as the IDE, the linker, and other programs) in DOS protected mode.

■ **Programmer's Platform:** Turbo C++ 3.0 comes with an improved version of the Programmer's Platform, Borland's open-architecture IDE that gives you access to the following full range of programming tools and utilities:

- a multi-file editor, featuring both an industry-standard Common User Access (**CUA**) interface and a familiar alternate interface, compatible with previous versions of Turbo C++
- advanced Turbo Editor Macro Language (TEML) and compiler
- multiple overlapping windows with full mouse support
- fully integrated debugger running in DPML, for debugging large applications
- support for inline assembler code
- complete undo and redo capability with an extensive buffer and much more.

■ **VROOMM:** Turbo C++'s Virtual Run-time Object-Oriented Memory Manager lets you overlay your code without complexity. You select the code segments for overlaying; VROOMM takes care of the rest, doing the work needed to fit your code into 640K.

■ **Help:** Online context-sensitive hypertext help has copy-and-paste program examples for practically every function.

■ **Streams:** Turbo C++ includes full support for C++ iostreams, plus special Borland extensions.

■ **Container classes:** Advanced container class libraries gives you sets, bags, lists, arrays, B-trees and other reusable data structures, implemented both as templates and as object-based containers for maximum flexibility.

Other features:

- Over 200 new library functions for maximum flexibility and compatibility.
- Complex and BCD math, fast huge arithmetic.
- Heap checking and memory management functions, with **far** objects and **huge** arrays.

- New BGI fonts and BGI support for the full ASCII character set.
- Response files for the command-line compiler.
- NMAKE compatibility for easy transition from Microsoft C.

Hardware and software requirements

Turbo C++ runs on the IBM PC compatible family of computers, including the AT and PS/2, along with all true IBM compatible 286, 386 or 486 computers. Turbo C++ requires a 286 or higher, DOS 3.31 or higher, a hard disk, a floppy drive, and at least 640K plus 1MB of extended memory; it runs on any 80-column monitor.

Turbo C++ includes floating-point routines that let your programs make use of an 80x87 math coprocessor chip. It emulates the chip if it is not available. Though it is not required to run Turbo C++, the 80x87 chip can significantly enhance the performance of your programs that use floating point math operations.

Turbo C++ also supports (but does not require) a mouse.

The Turbo C++ implementation

Turbo C++ is a full implementation of the AT&T C++ version 2.1, and it includes an implementation of templates. It also supports the American National Standards Institute (ANSI) C standard. In addition, Turbo C++ includes certain extensions for mixed-language and mixed-model programming that let you exploit your PC's capabilities.

The Turbo C++ package

Your Turbo C++ package consists of a set of disks and this manual, which tells you how to use the product, how to program in C and C++, and how to use specialized programming tools.

In addition to this manual, you'll find a convenient *Quick Reference* card. The disks contain all the programs, files, and libraries you need to create, compile, link, and run your Turbo C++ programs; they also contain sample programs, many standalone utilities, a context-sensitive help file, an integrated

debugger, and additional C and C++ documentation not covered in this documentation.

The User's Guide

The *User's Guide* introduces you to Turbo C++ and shows you how to create and run both C and C++ programs. It consists of information you'll need to get up and running quickly, and provides reference chapters on the features of Turbo C++: Borland's Programmer's Platform, including the editor and Project Manager, as well as details on using the command-line compiler. This manual includes the following chapters:

Introduction introduces you to Turbo C++ and tells you where to look for more information about each feature and option.

Chapter 1: Installing Turbo C++ tells you how to install Turbo C++ on your system; it also tells you how to customize the colors, defaults, and many other aspects of Turbo C++.

Chapter 2: IDE Basics introduces the features of the Programmer's Platform, tells you how to start up and exit from the IDE, and presents examples of how to use the IDE.

Chapter 3: An introduction to C++ covers basic C++ syntax.

Chapter 4: Object-oriented programming with C++ describes the major concepts involved in object-oriented programming.

Chapter 5: Hands-on C++ is step-by-step instruction in C++ programming.

Chapter 6: Debugging in the new IDE is a tutorial on how to debug a C program.

Chapter 7: Managing multi-file projects introduces you to Turbo C++'s built-in project manager and shows you how to build and update large projects from within the IDE.

Chapter 8: The command-line compiler tells how to use the command-line compiler and configuration files.

Chapter 9: MAKE: The program manager introduces the Turbo C++ MAKE utility, describes its features and syntax, and presents some examples of usage.

Chapter 10: TLINK: The Turbo linker is a complete reference to the features and functions of the Turbo Linker (TLINK).

Chapters 11: Lexical elements describes the Turbo C++ language tokens.

Chapter 12: Language structure explains how C++ tokens can be grouped together.

Chapter 13: C++ specifics describes details of the language and how to use C++ with and without the classes.

Chapter 14: The preprocessor describes preprocessor directives, their syntax and semantics, and the macro processor incorporated in the preprocessor.

Chapter 15: The main function describes the **main** function.

Chapter 16: Using C++ streams tells you how to use the C++ iostreams library.

Chapter 17: Converting from Microsoft C provides some guidelines on converting your Microsoft C programs to Turbo C++.

Chapter 18: Memory management briefly describes the DOS Protected Mode Interface (DPMI) and other memory-related topics.

Chapter 19: Mathematical operations covers floating-point and BCD math.

Chapter 20: Video functions is devoted to handling text and graphics in Turbo C++.

Appendix A: Editor reference provides a convenient command reference to using the editor with both the CUA command interface and the Turbo C++ alternate interface.

Appendix B: Precompiled headers tells you how to use Turbo C++'s exclusive precompiled headers feature to save substantial time when recompiling large projects.

Appendix C: Error messages lists and explains run-time, compile-time, linker, librarian, and Help compiler errors and warnings with suggested solutions.

Online documentation

In addition to the README.DOC and HELPMEN!.DOC, the following online documents are included with Turbo C++:

- **ANSI.DOC** covers those aspects of the ANSI C standard that have been left loosely defined or undefined by ANSI and how Borland has chosen to implement them.
- **CONTAIN.DOC** tells you how to use the Turbo C++ container class library in your programs.
- **UTIL.DOC** tells how to write inline assembly language functions that can be assembled with the built-in assembler (BASM) and used within your Turbo C++ program.
- **UTIL.DOC** describes TLIB and other utilities.
- Online help describes functions in the run-time library and contains information of a wide-range of other topics.

Using the manual




The manual is arranged so you can pick and choose among the chapters to find exactly what you need to know at the time you need to know it.

Programmers learning C or C++

If you don't know C or C++, there are many good products on the market that can get you going in these languages. You can use Chapters 11 through 16 for reference on specific technical aspects of Turbo C++.

Your next step is to start programming in C and C++. Chapter 15, "The main function," provides information on aspects of the **main()** function that is seldom found elsewhere. Or, you might prefer to use the online help; it contains much of the same information, and includes programming examples that you can copy into your own programs.

Typefaces and icons used in these books

- Monospace type* This typeface represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as TC to start up Turbo C++ 3.0).
- ALL CAPS** We use all capital letters for the names of constants and files.
- () Square brackets [] in text or DOS command lines enclose optional items that depend on your system. *Text of this sort should not be typed verbatim.*
- < > Angle brackets in the function reference section enclose the names of include files.
- Boldface** Turbo C++ function names (such as **printf**), class, and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used in text for Turbo C++ reserved words (such as **char**, **switch**, **near**, and **cdecl**), for format specifiers and escape sequences (**%d**, **\t**), and for command-line options (**/A**).
- Italics* *Italics* indicate variable names (identifiers) that appear in text. They can represent terms that you can use as is, or that you can think up new names for (your choice, usually). They are also used to emphasize certain words, such as new terms.
- Keycaps* This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."
-  This icon indicates keyboard actions.
-  This icon indicates mouse actions.
-  This icon indicates language items that are specific to C++.

How to contact Borland

Borland offers a variety of services to answer your questions about this product. Be sure to send in the registration card; registered owners are entitled to technical support and may receive information on upgrades and supplementary products.

Resources in your package

This product contains many resources to help you:

- The manual provides information on every aspect of the program. Use it as your main information source.
- While using the program, you can press *F1* for help.
- Many common questions are answered in the DOC files listed in the README file located in the program directory.

Borland resources

Borland Technical Support publishes technical information sheets on a variety of topics and is available to answer your questions.

800-822-4269 (voice)
Techfax

TechFax is a 24-hour automated service that sends free technical information to your fax machine. You can use your touch-tone phone to request up to three documents per call.

408-439-9096 (modem)
File Download BBS
2400 Baud

The Borland File Download BBS has sample files, applications, and technical information you can download with your modem. No special setup is required.

Subscribers to the CompuServe, GEnie, or BIX information services can receive technical support by modem. Use the commands in the following table to contact Borland while accessing an information service.

Online information services

Service	Command
CompuServe	GO BORLAND
BIX	JOIN BORLAND
GEnie	BORLAND

Address electronic messages to Sysop or All. Don't include your serial number; messages are in public view unless sent by a service's private mail system. Include as much information on the question as possible; the support staff will reply to the message within one working day.

408-438-5300 (voice)
Technical Support
6 a.m. to 5 p.m. PST

Borland Technical Support is available weekdays from 6:00 a.m. to 5:00 p.m. Pacific time to answer any technical questions you have about Borland products. Please call from a telephone near your computer, and have the program running. Keep the following information handy to help process your call:

- Product name, serial number, and version number.

- The brand and model of any hardware in your system.
- Operating system and version number. (Use the DOS command VER to find the version number.)
- Contents of your AUTOEXEC.BAT and CONFIG.SYS files (located in the root directory (\) of your computer's boot disk).
- A daytime phone number where you can be contacted.
- If the call concerns a problem, the steps to reproduce the problem.

*408-438-5300 (voice)
Customer Service
7 a.m. to 5 p.m. PST*

Borland Customer Service is available weekdays from 7:00 a.m. to 5:00 a.m. Pacific time to answer any non-technical questions you have about Borland products, including pricing information, upgrades, and order status.

P A R T

1

Using Turbo C++

Installing Turbo C++

Your Turbo C++ package includes two different versions of Turbo C++: the IDE (Programmer's Platform) and the DOS command line version.

IMPORTANT! To create backup copies of your disks, put the backup on the same type of disk as the source. For example, if you're backing up the 5 1/4-inch 1.2 Mb disk set, use only blank 5 1/4-inch 1.2 Mb disks for backup. The installation doesn't work correctly unless the original disks and backup disks are the same type of storage media.

Turbo C++ comes with an automatic installation program called INSTALL. Because we used file-compression techniques, you must use this program; you can't just copy the Turbo C++ files onto your hard disk. Instead, INSTALL automatically copies and uncompresses the Turbo C++ files. For reference, the README file on the installation disk includes a list of the distribution files.

We assume you are already familiar with DOS commands. If you don't already know how to use DOS commands, refer to your DOS reference manual before setting up Turbo C++ on your system. For example, you'll need the DISKCOPY command to make backup copies of your distribution disks. Make a complete working copy of your distribution disks when you receive them, then store the original disks away in a safe place.

None of Borland's products use copy protection schemes. If you are not familiar with Borland's No-Nonsense License Statement, read the agreement included with your Turbo C++ package. Fill in the product registration card and return it by mail to Borland. Returning this information ensures that you'll receive important upgrades and new product announcements promptly.

This chapter contains the following information:

- installing Turbo C++ on your system
- accessing the README file
- accessing the HELPME! file

- a pointer to more information on example programs
- information about customizing Turbo C++ default settings, display colors, and so on.

Once you have installed Turbo C++, you'll be ready to start digging into Turbo C++. The Introduction tells where to find out more about Turbo C++ features in the documentation.

Using INSTALL

We recommend that you study the README file before proceeding with the installation.

INSTALL detects what hardware you're using and configures Turbo C++ appropriately. It creates required directories and transfers files from the distribution disks to your hard disk. The distribution disks are the disks included in the Turbo C++ product package.

To install Turbo C++, perform the following steps:

1. Insert the installation disk (disk 1) into drive A. Type the following command, then press *Enter*.

```
A:INSTALL
```

2. Press *Enter* at the installation screen.
3. Follow the prompts.
4. At the end of installation, you might want to add the following line to your CONFIG.SYS file:

```
FILES = 20
```

You might want to add the following line to your AUTOEXEC.BAT file or change your PATH statement accordingly:

```
PATH = C:\TC\BIN
```

Important! When INSTALL is finished, it displays the README file, which contains important, last-minute information about Turbo C++. The HELPME!.DOC file is available to answer many common technical support questions, also.

Protected mode and memory

Turbo C++ utilizes the DPMS (Dos Protected Mode Interface) to run the compiler in protected mode, giving you access to all your computer's memory without swapping. The protected mode interface is transparent to the user, and you usually don't have to

think about it. The following sections discuss the rare circumstances that require your interaction with the DPMI.

DPMIINST When you run Turbo C++ for the first time, you might receive an error message and have to run DPMIINST. Turbo C++ uses an internal database of various machine characteristics to determine how to enable protected mode operations on your machine, and it configures itself accordingly. If your machine isn't recognized by Turbo C++, you receive the following error message:

```
Machine not in database (RUN DPMIINST)
```

If you get this message, run the DPMIINST program by typing the following command at the DOS prompt:

```
DPMIINST
```

Follow the program's instructions. DPMIINST runs your machine through a series of tests to determine the best way of enabling protected mode, and automatically configures Turbo C++ accordingly. Once you run DPMIINST, you never have to run it again.

DPMIMEM By default, the Turbo C++ DPDI interface allocates all available extended and expanded memory for its own use. If you don't want all of the available memory to be taken by the DPDI kernel, set the environment variable DPDI, which specifies a maximum amount of memory to use, using the following syntax:

```
set DPMIMEM=MAXMEM nnnn
```

where *nnnn* is the amount of memory in kilobytes.

You can set DPMIMEM by entering the command at the DOS prompt, or you can put it into your AUTOEXEC.BAT file to set DPMIMEM automatically.

For example, if a user has a system with 4MB and wants the DPDI kernel to use 2MB of it, leaving the other 2MB alone, the DPMIMEM variable would be set as follows:

```
c:> set DPMIMEM=MAXMEM 2000
```

DPMIRES DPMIRES is a utility that can be used with Turbo C++ to increase performance of some language tools under certain conditions. In particular, the performance of TCC and TLINK can be enhanced through its use.

When run, DPMIRES enables the Dos Protected Mode interface and spawns a DOS command shell. Applications such as TLINK load faster into this shell. Typing 'EXIT' to the shell removes it.

DPMIRES is especially useful if you're compiling with batch files, instead of using the protected mode MAKE. In this situation, it's more efficient to run DPMIRES before running the batch file, since the compile loads faster on each invocation.

Extended and expanded memory

Once the DPMI kernel is loaded (either by running TC or through the DPMIRES utility), the Turbo C++ integrated development environment interacts directly with the DPMI server to allocate its memory during both loading and operating. By default, the IDE uses all the extended memory reserved by the DPMI kernel and all available EMS (expanded) memory with the EMS memory being used as a swap device.

The Options | Environment | Startup... dialog items (Use Extended Memory and Use EMS Memory) and the corresponding **/X** and **/E** command line switches change the default method of memory allocation, which affects how much memory the IDE uses. The settings don't change the memory reserved by the kernel itself.

The Use Extended Memory dialog item corresponds to the **/X** command line option. It tells TC how much of the memory reserved by the DPMI kernel to use. By limiting TC's use of the kernel's memory, other DPMI applications can be run from within the IDE's memory (using the Transfer capability), or other applications can be run from a DOS shell opened from the IDE.

The Use EMS Memory dialog item corresponds to the **/E** command line option. It tells the IDE how many 16K EMS pages to use as a swap device. Unless the kernel has been instructed to set aside some available memory, no EMS pages are available to the IDE.

Running Turbo

C++

Once you have installed Turbo C++, change to the Turbo C++ \ BIN directory, type TC and press *Enter*.

After experimenting with the IDE, select the Options | Environment | Startup and Options | Environment | Colors if you want to customize the IDE.

Laptop systems

If you have a laptop computer (one with an LCD or plasma display), set your screen parameters before using Turbo C++ to the recommended setting by typing the following command at the DOS prompt:

```
MODE BW80
```

To set the MODE automatically, either create a batch file to set Mode to BW80, or better yet, install Turbo C++ for a black-and-white screen from within the IDE using the Options | Environment | Startup option. Choose "Black and White / LCD" from the Video options group.

The README file

The README file contains last-minute information that might not be in the manual.

Turbo C++ automatically displays the README file when you run the INSTALL program. To access the README file at a later time type the following command at the DOS command line:

```
README
```

The FILELIST.DOC and HELPME!.DOC files

Other files on the installation disk are FILELIST.DOC and HELPME!.DOC. FILELIST.DOC briefly describes every file on the distribution disk. HELPME!.DOC contains answers to frequently-encountered problems. Consult these files to try to

solve difficulties. You can use the README program to look at HELPME!.DOC. Type the following command at the command line:

```
README HELPME!.DOC
```

Example programs

Your Turbo C++ package includes the source code for a large number of example programs in C and C++, including a complete spreadsheet program called Turbo Calc. These programs are located in the EXAMPLES directory (and subdirectories) created by INSTALL. The EXAMPLES directory also contains subdirectories for examples of the other tools and utilities that come with Turbo C++. Before you compile any of these example programs, you should read the printed or online documentation for them.

Customizing the IDE

For detailed information on the menus and options in the IDE, see Chapter 2, "IDE Basics."

Turbo C++ allows you to customize your installation from within the IDE, using the various options in the Options | Environment menu. The options specify the video mode, editing modes, default directories, menu colors, and control color syntax highlighting during debugging.

IDE basics

Borland's Programmer's Platform, also known as the integrated development environment or IDE, has everything you need to write, edit, compile, link, and debug your programs. It provides

- multiple, movable, resizable windows
- mouse support and dialog boxes
- syntax highlighting in colors you can change
- cut, paste, and copy commands that use the Clipboard
- full editor undo and redo
- online Help
- examples ready to copy and paste from Help
- a built-in assembler
- quick transfer to other programs and back again
- an editor macro language

This chapter explains how to start up and exit the Turbo C++ IDE, discusses its generic components, and explains how configuration and project files work. The table at the end of the chapter cross-references IDE menu items to descriptions throughout the manual.

Starting and exiting

Turbo C++ runs only in protected mode.

To start the IDE, type `TC` at the DOS prompt. You can follow it with one or more IDE command-line options.

Command-line options

The command-line options for Turbo C++'s IDE are `/b`, `/d`, `/e`, `/h`, `/l`, `/m`, `/p`, `/rx`, `/s`, and `/x` using this syntax:

```
TC [option [option...]] [sourcename | projectname [sourcename]]
```

where *option* can be one or more of the options, *sourcename* is any ASCII file (default extension assumed), and *projectname* is your project file (it *must* have the `.PRJ` extension). The order and case is not important.

To turn an option off, follow the option with a minus sign. For example,

```
TC /e-
```

turns off the default swap to expanded memory option.

The `/b` option

The `/b` (build) option causes Turbo C++ to recompile and link all the files in your project, print the compiler messages to the standard output device, and then return to the operating system. This option lets you start Turbo C++ from a batch file so you can automate project builds. Turbo C++ determines what `.EXE` to build based on the file you specify on the command line. If it doesn't find a project file it builds the active file loaded into the IDE edit window. It looks for project file (`.PRJ`) and source file (`.CPP`) extensions.

To specify a project file, enter the `TC` command followed by `/b` and the project file name. For example,

```
TC /b myproj.prj
```

If there is no `MYPROG.PRJ` file, the following command loads the file `MYPROG.CPP` in the editor and then compiles and links it:

```
TC MYPROG /B
```

The `/d` option The `/d` option causes Turbo C++ to run in dual monitor mode if it detects two video cards installed in your computer (for example, a monochrome card and a color card); otherwise, the `/d` option is ignored. Using dual monitor mode makes it easier to watch a program's output while you are debugging the program.

If your system has two monitors, DOS treats one monitor as the active monitor. Use the DOS `MODE` command to switch between the two monitors (`MODE CO80`, for example, or `MODE MONO`). In dual monitor mode, the normal Turbo C++ screen appears on the inactive monitor, and program output goes to the active monitor. So when you type `TC /d` at the DOS prompt on one monitor, Turbo C++ comes up on the other monitor. When you want to test your program on a particular monitor, exit Turbo C++, switch the active monitor to the one you want to test with, and then issue the `TC /d` command again. Program output then goes to the monitor where you typed the `TC` command.

Keep the following in mind when using the `/d` option:

- Don't change the active monitor (by using the DOS `MODE` command, for example) while you are in a DOS shell (File | DOS Shell).
- User programs that directly access ports on the inactive monitor's video card are not supported, and have unpredictable results.
- Don't use it when you run or debug programs that explicitly make use of dual monitors.

The `/e` option The `/e` option tells Turbo C++ to swap to expanded memory if necessary; it is on by default. The syntax for this option is as follows:

```
/e[=n]
```

where *n* equals the number of pages of expanded memory that you want the IDE to use for swapping. Each page is 16K.

The `/h` option Typing `TC /h` on the command line, you get a list of all the command-line options available. Their default values are also shown.

- The `/l` option Use the `/l` (lowercase l) option if you're running Turbo C++ on an LCD screen.
- The `/m` option The `/m` option lets you do a make rather than a build. That is, only outdated source files in your project are recompiled and linked. Follow the instructions for the `/b` option, but use `/m` instead. See page 297 for MAKE information.
- The `/p` option If your program modifies the EGA palette registers (or has BGI do it), use the `/p` option, which controls palette swapping on EGA video adapters. The EGA palette is restored each time the screen is swapped.
- The `/r` option Use `/rx` to specify a swap drive, usually a RAM disk, if all your virtual memory fills up. The `x` in `/rx` is the letter of the swap drive. For example, `/rd` specifies drive D as the swap drive.
- The `/s` option Using the `/s` option, (on by default) the compiler allows the majority of available memory to be allocated for its internal tables while compiling. If it is compiling large modules, little memory may remain for the needed overlays; therefore, the compiler may spend a long time "thrashing," that is, swapping overlays in and out of memory.
- If you specify `/s-`, the compiler won't permit its internal tables to severely restrict the overlay space in memory. As a result, if you are compiling very large modules, the compilation may fail and you'll get an out-of-memory error, but the compiler won't thrash excessively.
- The `/x` option Use the `/x` switch to tell Turbo C++ how much of the available extended memory to use for its heap space.
- `/x`
- uses all available memory.
- `/x[=n]`
- where *n* equals the amount of memory in kilobytes, let's you specify how much extended memory should be used.

Exiting Turbo C++

There are three ways to leave the IDE.

- Choose File | Exit to leave the IDE completely; you have to type TC again to reenter it. You'll be prompted to save your programs before exiting, if you haven't already done so.
- Choose File | DOS Shell to shell out from the IDE to enter commands at the DOS command line. When you're ready to return to the IDE, type EXIT at the command line and press *Enter*. The IDE reappears just as you left it.
- Choose a program from the System menu (≡) to temporarily transfer to another program without leaving the IDE. You can add new Transfer programs with the Options | Transfer command.

You return to the IDE after you exit the program you transferred to.

IDE components

There are three visible components to the IDE: the menu bar at the top, the window area in the middle, and the status line at the bottom of the screen. Many menu items also offer dialog boxes. Although there are several different ways to make selections in the IDE, they access the same commands and dialog boxes. Before we list menu items in the IDE, we'll explain these more generic components.

The menu bar and menus

The menu bar is your primary access to all the menu commands. The menu bar is always visible except when you're viewing your program's output or transferring to another program.

If a menu command is followed by an ellipsis (...), choosing the command displays a dialog box. If the command is followed by an arrow (▶), the command leads to another menu. If the command has neither an ellipsis nor an arrow, the action occurs as soon as you choose the command.

Here is how you choose menu commands using the keyboard:



1. Press *F10*. This makes the menu bar active; the next thing you type will relate to the items on the menu bar.

To cancel an action,
press *Esc*.

2. Use the arrow keys to select the menu you want to display. Then press *Enter*.

As a shortcut for this step, you can just press the highlighted letter of the menu title. For example, when the menu bar is active, press *E* to move to and display the Edit menu. At any time, press *Alt* and the highlighted letter (such as *Alt+E*) to display the menu you want.

3. Use the arrow keys again to select a command from the menu you've opened. Then press *Enter*.

At this point, Turbo C++ either carries out the command, displays a dialog box, or displays another menu.



Turbo C++ uses only the left mouse button. You can, however, customize the right button and make other mouse option changes, by choosing *Options | Environment | Mouse*.

There are two ways to choose commands with a mouse:

- Click the desired menu title to display the menu and click the desired command.
- Or, drag straight from the menu title down to the menu command. Release the mouse button on the command you want. (If you change your mind, just drag off the menu; no command will be chosen.)

Note that some menu commands are unavailable when it would make no sense to choose them. However, you can always get online Help about currently unavailable commands.

Shortcuts

Turbo C++ offers a number of quick ways to choose menu commands. The click-drag method for mouse users is an example. From the keyboard, you can use a number of keyboard shortcuts (or *hot keys*) to access the menu bar, choose commands, or work within dialog boxes. You need to hold down *Alt* while pressing the highlighted letter when moving from an input box to a group of buttons or boxes. Here's a list of the shortcuts available:

Input boxes are described on
page 32

Do this...	To accomplish this...
Press <i>Alt</i> plus the highlighted letter of the command (just press the highlighted letter in a dialog box). For the \equiv menu, press <i>Alt+Spacebar</i> .	Display the menu or carry out the command.
Type the keystrokes next to a menu command.	Carry out the command.

For example, to cut selected text, press *Alt+E T* (for Edit | Cut) or you can just press *Shift+Del*, the shortcut displayed next to it.

Command sets Turbo C++ has two command sets: the Common User Access (CUA) command set and the Alternate command set popularized in previous Borland products. The set determines the shortcuts available to you, which keys you use within the editor, and, to some extent, how the editor works. See more about using command sets in the editor in Appendix A. A Native command set option is discussed at the end of this section.

You can select a command set by choosing Options | Environment | Preferences and then selecting the command set you prefer in the Preferences dialog box. If you are a long-time Borland language user, you may prefer the Alternate command set.

The following tables list the most-used Turbo C++ hot keys in both command sets.

Table 2.1: General hot keys

CUA	Alternate	Menu item	Function
F1	F1	Help	Displays context-sensitive help screen.
	F2	File Save	Saves the file that's in the active edit window.
	F3	File Open	Brings up a dialog box so you can open a file.
	F4	Run Go to Cursor	Runs your program to the line where the cursor is positioned.
	F5	Window Zoom	Zooms the active window.
Ctrl+F6	F6	Window Next	Cycles through all open windows.
F7	F7	Run Trace Into	Runs your program in debug mode, tracing into functions.
F8	F8	Run Step Over	Runs your program in debug mode, stepping over function calls.
F9	F9	Compile Make	Invokes the Project Manager to make an .EXE file.
F10	F10	(none)	Takes you to the menu bar.

Table 2.2: Menu hot keys

CUA	Alternate	Menu item	Function
Alt+Spacebar	Alt+Spacebar	≡ menu	Takes you to the ≡ (System) menu
Alt+C	Alt+C	Compile menu	Takes you to the Compile menu
Alt+D	Alt+D	Debug menu	Takes you to the Debug menu
Alt+E	Alt+E	Edit menu	Takes you to the Edit menu
Alt+F	Alt+F	File menu	Takes you to the File menu
Alt+H	Alt+H	Help menu	Takes you to the Help menu
Alt+O	Alt+O	Options menu	Takes you to the Options menu
Alt+P	Alt+P	Project menu	Takes you to the Project menu
Alt+R	Alt+R	Run menu	Takes you to the Run menu
Alt+S	Alt+S	Search menu	Takes you to the Search menu

Table 2.2: Menu hot keys (continued)

<i>Alt+W</i> <i>Alt+F4</i>	<i>Alt+W</i> <i>Alt+X</i>	Window menu File Exit	Takes you to the Window menu Exits Turbo C++ to DOS
-------------------------------	------------------------------	----------------------------	--

Table 2.3: Editing hot keys

CUA	Alternate	Menu item	Function
<i>Ctrl+Ins</i>	<i>Ctrl+Ins</i>	Edit Copy	Copies selected text to Clipboard
<i>Shift+Del</i>	<i>Shift+Del</i>	Edit Cut	Places selected text in the Clipboard, deletes selection
<i>Shift+Ins</i>	<i>Shift+Ins</i>	Edit Paste	Pastes text from the Clipboard into the active window
<i>Ctrl+Del</i>	<i>Ctrl+Del</i>	Edit Clear	Removes selected text from the window but doesn't put it in the Clipboard
<i>Alt+Bkspc</i>	<i>Alt+Bkspc</i>	Edit Undo	Restores the text in the active window to a previous state
<i>Alt+Shft+Bksp</i> <i>F3</i>	<i>Alt+Shft+Bksp</i> <i>Ctrl+L</i> <i>F2</i> <i>F3</i>	Edit Redo Search Search Again File Save File Open	"Undoes" the previous Undo. Repeats last Find or Replace command Saves the file in the active edit window Lets you open a file

Table 2.4: Window management hot keys

CUA	Alternate	Menu item	Function
<i>Alt+#</i>	<i>Alt+#</i>		Displays a window, where # is the number of the window you want to view
<i>Alt+0</i>	<i>Alt+0</i>	Window List All	Displays a list of open windows
<i>Ctrl+F4</i>	<i>Alt+F3</i>	Window Close	Closes the active window
<i>Shift+F5</i>		Window Tile	Tiles all open windows
<i>Alt+F5</i>	<i>Alt+F4</i>	Debug Inspect	Opens an Inspector window
<i>Shift+F5</i>	<i>Alt+F5</i>	Window User Screen	Displays User Screen
	<i>F5</i>	Window Zoom	Zooms/unzooms the active window
<i>Ctrl+F6</i>	<i>F6</i>	Window Next	Switches the active window
	<i>Ctrl+F5</i>		Changes size or position of active window

Table 2.5: Online Help hot keys

CUA	Alternate	Menu item	Function
<i>F1</i> <i>F1 F1</i>	<i>F1</i> <i>F1 F1</i>	Help Contents	Opens a context-sensitive help screen Brings up Help on Help. (Just press <i>F1</i> when you're already in the help system.)
<i>Shift+F1</i>	<i>Shift+F1</i>	Help Index	Brings up Help index
<i>Alt+F1</i>	<i>Alt+F1</i>	Help Previous Topic	Displays previous Help screen
<i>Ctrl+F1</i>	<i>Ctrl+F1</i>	Help Topic Search	Calls up language-specific help (in the active edit window)

Table 2.6: Debugging/Running hot keys

CUA	Alternate	Menu item	Function
Alt+F5	Alt+F4	Debug Inspect	Opens an Inspector window
Alt+F7	Alt+F7	Search Previous Error	Takes you to previous error
Alt+F8	Alt+F8	Search Next Error	Takes you to next error
Alt+F9	Alt+F9	Compile Compile to OBJ	Compiles to .OBJ
Ctrl+F2	Ctrl+F2	Run Program Reset	Resets running program
	Ctrl+F3	Debug Call Stack	Brings up call stack
	Ctrl+F4	Debug Evaluate/Modify	Evaluates an expression
Ctrl+F5	Ctrl+F7	Debug Add Watch	Adds a watch expression
F5	Ctrl+F8	Debug Toggle Breakpoint	Sets or clears conditional breakpoint
Ctrl+F9	Ctrl+F9	Run Run	Runs program
	F4	Run Go To Cursor	Runs program to cursor position
F7	F7	Run Trace Into	Executes one line, tracing into functions
F8	F8	Run Step Over	Executes one line, skipping function calls
F9	F9	Compile Make	Makes (compiles/links) program

Native option

Native makes the Alternate command set the default.

If you choose Options | Environment | Preferences to display the Preferences dialog box, you'll notice another option: Native. This is the default setting.

The IDE uses the configuration file, TCCONFIG.TC, to determine which command set is in effect. Therefore, if you have selected the CUA command set in the IDE, that is the one in effect the next time you start up.

With Native selected, Turbo C++ for DOS uses the Alternate command set automatically.

Turbo C++ windows

If you exit Turbo C++ with a file open in a window, you are returned to your desktop, open file and all, when you next use Turbo C++.

Most of what you see and do in the IDE happens in a *window*. A window is a screen area that you can open, close, move, resize, zoom, tile, and overlap.

You can have many windows open in the IDE, but only one window can be *active* at any time. Any command you choose or text you type generally applies only to the active window. (If you have the same file open in several windows, the action will apply to the file everywhere that it's open.)

You can spot the active window easily: It's the one with the double-lined border around it. The active window always has a close box, a zoom box, and scroll bars. If your windows are over-

lapping, the active window is always the one on top of all the others (the foremost one).

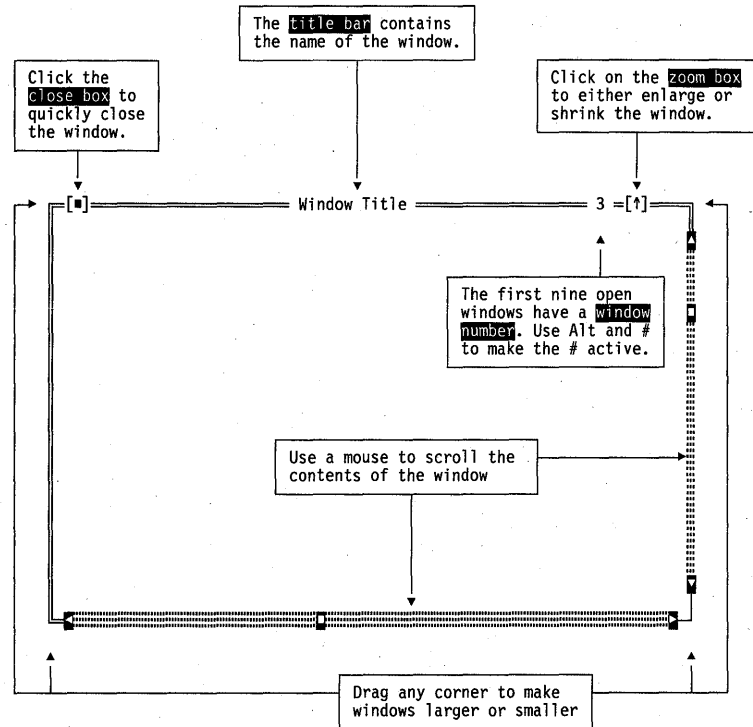
There are several types of windows, but most of them have these things in common:

- a title bar
- a close box
- scroll bars
- a zoom box
- a window number (1 to 9)

A edit window also displays the current line and column numbers in the lower left corner. If you've modified your file, an asterisk (*) appears to the left of the column and line numbers.

The following figure shows a typical window:

Figure 2.1
A typical window



The *close box* of a window is the box in the upper left corner. Click this box to quickly close the window. (Or choose Window | Close.) The Inspector and Help windows are considered temporary; you can close them by pressing *Esc*.

The *title bar*, the topmost horizontal bar of a window, contains the name of the window and the window number. Double-clicking the title bar zooms the window. You can also drag the title bar to move the window around.

Shortcut: Double-click the title bar of a window to zoom or restore it.



The *zoom box* of a window appears in the upper right corner. If the icon in that corner is an up arrow (↑), you can click the arrow to enlarge the window to the largest size possible. If the icon is a doubleheaded arrow (↕), the window is already at its maximum size. In that case, clicking it returns the window to its previous size. To zoom a window from the keyboard, choose Window | Zoom.

Alt+0 gives you a list of all windows you have open.

The first nine windows you open in Turbo C++ have a *window number* in the upper right border. You can make a window active (and thereby bring it to the top of the heap) by pressing *Alt* in combination with the window number. For example, if the Help window is #5 but has gotten buried under the other windows, *Alt+5* brings it to the front.

Scroll bars are horizontal or vertical bars that look like this:



Scroll bars also show you where you are in your file.



You use these bars with a mouse to scroll the contents of the window. Click the arrow at either end to scroll one line at a time. (Keep the mouse button pressed to scroll continuously.) You can click the shaded area to either side of the scroll box to scroll a page at a time. Finally, you can drag the scroll box to any spot on the bar to quickly move to a spot in the window relative to the position of the scroll box.

You can drag any corner to make a window larger or smaller. To resize using the keyboard, choose *Size/Move* from the Window menu.

Window management

Table 2.7 gives you a quick rundown of how to handle windows in Turbo C++. Note that you don't need a mouse to perform these actions—a keyboard works just fine.

Table 2.7
Manipulating windows

To accomplish this:	Use one of these methods
Open an edit window	Choose File Open to open a file and display it in a window.
Open other windows	Choose the desired window from the Window menu

Table 2.7: Manipulating windows (continued)

Close a window	Choose Close from the Window menu or click the close box of the window.
Activate a window	Click anywhere in the window, or Press <i>Alt</i> plus the window number (1 to 9, in the upper right border of the window), or Choose Window List or press <i>Alt+0</i> and select the window from the list, or Choose Window Next to make the next window active (next in the order you first opened them).
Move the active window	Drag its title bar. Or choose Window Size/Move and use the arrow keys to place the window where you want it, then press <i>Enter</i> .
Resize the active window	Drag any corner. Or choose Window Size/Move and press <i>Shift</i> while you use the arrow keys to resize the window, then press <i>Enter</i> .
Zoom the active window	Click the zoom box in the upper right corner of the window, or Double-click the window's title bar, or Choose Window Zoom.

The status line

The status line appears at the bottom of the screen. It

- reminds you of basic keystrokes and shortcuts (or hot keys) applicable at that moment in the active window.
- lets you click the shortcuts to carry out the action instead of choosing the command from the menu or pressing the shortcut keystroke.
- tells you what the program is doing. For example, it displays Saving *filename...* when an edit file is being saved.
- offers one-line hints on any selected menu command and dialog box items.

The status line changes as you switch windows or activities. One of the most common status lines is the one you see when you're actually writing and editing programs in an edit window. Here is what it looks like:

Figure 2.2
A typical status line

F1 Help F2 Save F3 Open F7 Trace F8 Step F9 Make F10 Menu

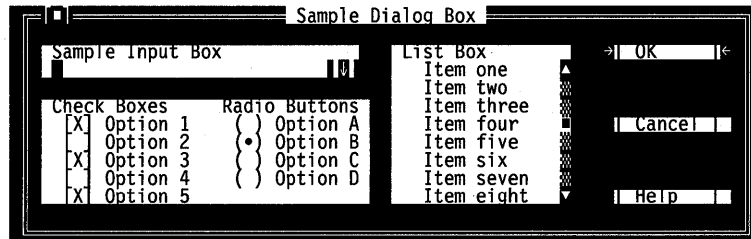
When you've selected a menu title or command, the status line changes to display a one-line summary of the function of the selected item.

Dialog boxes

A menu command with an ellipsis (...) after it leads to a *dialog box*. Dialog boxes offer a convenient way to view and set multiple options. When you're making settings in dialog boxes, you work with five basic types of onscreen controls: action buttons, radio buttons, check boxes, input boxes, and list boxes. Here's a sample dialog box that illustrates some of these items:

Figure 2.3
A sample dialog box

If you have a color monitor, Turbo C++ uses different colors for various elements of the dialog box.



Action buttons

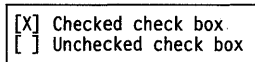
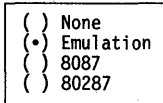
This dialog box has three standard buttons: OK, Cancel, and Help. If you choose OK, the choices in the dialog box are accepted; if you choose Cancel, nothing changes, no action takes place, and the dialog box is put away. Choose Help to open a Help window about this dialog box. *Esc* is always a keyboard shortcut for Cancel (even if no Cancel button appears).

If you're using a mouse, click the dialog box button you want. When you're using the keyboard, press *Alt* and the highlighted letter of an item to activate it. For example, *Alt+K* selects the OK button because the K in OK is highlighted. Press *Tab* or *Shift+Tab* to move forward or back from one item to another in a dialog box. Each element is highlighted when it becomes active.

You can select another button with *Tab*; press *Enter* to choose that button.

In this dialog box, OK is the *default button*, which means you need only press *Enter* to choose that button. (On monochrome systems, arrows indicate the default; on color monitors, default buttons are highlighted.) Be aware that tabbing to a button makes that button the default.

Radio buttons and check boxes



Radio buttons are like car radio buttons. They come in groups, and only one radio button in the group can be on at any one time. To choose a radio button, click it or its text. From the keyboard, select *Alt* and the highlighted letter, or press *Tab* until the group is highlighted and then use the arrow keys to choose a particular radio button. Press *Tab* or *Shift+Tab* again to leave the group with the new radio button chosen.

Check boxes differ from radio buttons in that you can have any number of check boxes checked at any time. When you select a check box, an *x* appears in it to show you it's on. An empty box indicates it's off. To change the status of a check box, click it or its text, press *Tab* until the check box is highlighted and then press *Spacebar*, or select *Alt* and the highlighted letter.

If several check boxes apply to a topic, they appear as a group. In that case, tabbing moves to the group. Once the group is selected, use the arrow keys to select the item you want, and then press *Spacebar* to check or uncheck it. On monochrome monitors, the active check box or group of check boxes will have a chevron symbol (*»*) to the left and right. When you press *Tab*, the chevrons move to the next group of check boxes or radio buttons.

Input and list boxes

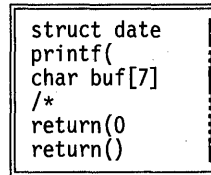
Input boxes let you type in text. Most basic text-editing keys work in the text box (for example, arrow keys, *Home*, *End*, and *Ins*). If you continue to type once you reach the end of the box, the contents automatically scroll. If there's more text than what shows in the box, arrowheads appear at the end (*◀* and *▶*). You can click the arrowheads to scroll or drag the text. If you need to enter control characters (such as *^L* or *^M*) in the input box, then prefix the character with a *^P*. So, for example, to enter *^L* into the input box, hold down the *Ctrl* key and press *P L*. (This capability is useful for search strings.)

You can control whether history lists are saved to the desktop using Options | Environment | Desktop.

If an input box has a down-arrow icon (*↓*) to its right, there is an associated *history list*. Click the *↓* to display the list. You'll find text you typed the last few times you used the input box. Press *Enter* to choose an item from this list. The Find box, for example, has such a history list, which keeps track of the text you searched for previously. Try choosing a previous search string. You can also edit an entry in the history list. Press *Esc* to exit from the history list without making a selection.

Here is what a history list for the Find text box might look like if you had used it six times previously:

Text to find ↓



```
struct date
printf(
char buf[7]
/*
return(0
return()
```

A final component of many dialog boxes is a *list box*, which lets you scroll through and select from variable-length lists (often file names) without leaving a dialog box. If a blinking cursor appears in the list box and you know what you're looking for, you can type the word (or the first few letters of the word) and Turbo C++ will search for it.

You make a list box active by clicking it or by choosing the highlighted letter of the list title (or press *Tab* until it's highlighted). Once a list box is displayed, you can use the scroll box to move through the list or press ↑ or ↓ from the keyboard.

Configuration and project files

With configuration files, you can specify how you want to work within the IDE. Project files contain all the information necessary to build a project, but don't affect how you use the IDE.

The configuration

file The IDE configuration file, `TCCONFIG.TC`, contains only environmental (or global) information, including

- editor key binding and macros,
- editor mode setting (such as autoindent, use tabs, etc.),
- mouse preferences, and
- auto-save flags.

The configuration file is not required to build programs defined by a project. The project (`.PRJ`) file handles those details.

When you start a programming session, Turbo C++ looks for TCCONFIG.TC first in the current directory and then in the directory that contains TC.EXE.

Project files

The IDE places all information needed to build a program into a binary project file, a file with a .PRJ extension. Project files contain the settings for

- compiler, linker, make and librarian options
- directory paths
- the list of all files that make up the project
- special translators (such as Turbo Assembler)

In addition, the project file contains other general information on the project, such as compilation statistics (shown in the project window), and cached autodependency information.

IDE .PRJ project files correspond to the .CFG configuration files that you supply to the command-line compiler (the default command-line compiler configuration file is TURBOC.CFG). The PRJCFG utility can convert .PRJ files to .CFG files and .CFG files to .PRJ files.

You can load project files in any of three ways:

1. When starting Turbo C++, give the project name with the .PRJ extension after the TC command; for example,

```
TC myproj.PRJ
```

You must use the .PRJ extension to differentiate it from source files.
2. If there is only one .PRJ file in the current directory, the IDE assumes that this directory is dedicated to this project and automatically loads it.
3. To load a project from within the IDE, select Project | Open Project.

The project directory

When a project file is loaded from a directory other than the current directory, the current DOS directory is set to where the project is loaded from. This allows your project to be defined in terms of relative paths in the Options | Directories dialog box and also allows projects to move from one drive to another or from

one directory branch to another. Note, however, that changing directories after loading a project may make the relative paths incorrect and your project unbuildable. If this happens, change the current directory back to where the project was loaded from.

Desktop files Each project file has an associated desktop file (*prjname.DSK*) that contains state information about the associated project. While none of its information is needed to build the project, all of the information is directly related to the project. The desktop file includes

You can set some of these options on or off using Options | Environment | Desktop.

- the context information for each window of the desktop (for example, your positions in the files or bookmarks)
- the history lists for various input boxes (for example, search strings or file masks)
- the layout of the windows on the desktop
- the contents of the Clipboard
- watch expressions
- breakpoints

Changing project files Because each project file has its own desktop file, changing to another project file causes the newly loaded project's desktop to be used, which can change your entire window layout. When you create a new project (by using Project | Open Project and typing in a new .PRJ file name), the new project's desktop inherits the previous desktop. When you select Project | Close Project, the default project is loaded and you get the default desktop and project settings.

Default files When no project file is loaded, there are two default files that serve as global place holders for project- and state-related information: TCDEF.DPR and TCDEF.DSK files, collectively referred to as the *default project*.

These files are usually stored in the same directory as TC.EXE, and are created if they are not found. When you run the IDE from a directory without loading a project file, you get the desktop and settings from these files. These files are updated when you change any project-related options (for example, compiler options) or when your desktop changes (for example, the window layout).

When you start a new project, the options you set in your previous project will be in effect.

IDE menus

The IDE is designed to explore and learn online. As you scroll the menus, notice the status line explanation of each selection. When you select an item, press F1 for online Help. The following table has general information. The cross-reference table refers individual menu entries to related information in the rest of the manual.

Table 2.8
IDE overview

Menu	Use
≡	Repaint Desktop. Transfer to displayed programs. If you don't have the sample programs, add your own standalone programs to the menu using the Options Transfer command.
File	Open and create program files in edit windows. Save changes, perform other file functions, and quit the IDE.
Edit	Cut, copy marked text to the Clipboard, and paste from the Clipboard to the cursor position in an edit window. Undo changes and even reverse the changes you've just undone. The Edit Copy Example command copies the preselected example text in the current Help window to the Clipboard.
Search	Search for text, function declarations, and error locations in your files.
Run	Run your program or start and end debugging sessions. Set arguments for the IDE to pass to the program as if you entered them on the command line. For example, if you would type <code>progname args</code> , enter <code>args</code> in the Run Argument dialog box.
Compile	Compile the currently selected module. Make (compiles source files that have changed since the last compile then links if necessary) or build (compiles and links all modules, regardless of change) your current project.
Debug	Use the integrated debugger. See Chapter 6, "Debugging in the new IDE." Specify whether or not debugging information is generated in the Options Debugger dialog box.
Project	Control the features of the Project manager. See Chapter 7, "Managing multi-file projects".
Options	View and change various default settings in Turbo C++ including syntax highlighting.
Window	Manage windows.
Help	Online help. Try F1, Alt F1, Shift F1, and see Help on Help.

Syntax highlighting

The edit window is syntax-sensitive, helping you see what the code is doing. Use Options | Environment | Editor | Syntax Highlighting to toggle the feature on and off. To choose the colors, select Options | Environment | Colors | Edit Window. Syntax items include Marked Text, Normal Text, Break Point Line, Comment, Reserved, Identifier, Symbol, String, Integer, Float, Octal, Hex, Character, Preprocessor, and Illegal Char.

IDE cross-reference

Specific IDE menu items and dialog boxes relate to topics discussed in other parts of this manual or in online Help. Table 2.9 helps you find them in the book. We capitalize each word in the menu item to make them stand out. Switches refer to TCC command-line options unless specifically defined as TC, MAKE, TLINK, or TLIB. An asterisk (*) shows that the option is enabled by default. Since the IDE has an integrated debugger, the standalone debugger command-line options are documented in the manuals for that product.

Table 2.9: IDE menu cross-reference

IDE menu selection	Switch	Page
File		36
Edit		36, 629
Search		36
Locate Function ...		240
Run		
Run		36
Program Reset (Integrated debugger)		250
Go To Cursor		225
Trace Into		224
Step Over		224
Arguments ...		36, 250
Compile		
Compile To OBJ		36, 286
Make	-c	22, 36
Build All	TC /m	20, 36
Debug	TC /b	
Inspect		229
Evaluate/Modify		232
Call Stack		240

Table 2.9: IDE menu cross-reference (continued)

Watches		237
Add Watch...		237
Delete Watch		239
Edit Watch...		239
Remove All Watches		239
Toggle Breakpoint		228
Breakpoints...		227
Project		
Open Project...		253
Add Item...		253
Delete Item		253
Local Options		
Command Line Options		267
Output Path (for .OBJ file)		254
Project File Translators		259
Overlay This Module (with O L S Output Overlaid EXE)	TLINK /o	342, 587
Exclude Debug Information	TLINK /v-	344
Include Files... (compiled include files)		257
Options		
Application		
Standard*	-ms	273
Overlay	-mm -Y	273, 279
Library	-ms	273
	TLIB	UTIL.DOC
Compiler		
Code Generation		
Model		
Tiny	-mt	273
Small	-ms	273
Medium	-mm	273
Compact	-mc	273
Large	-ml	273
Huge	-mh	273
Options		
Treat Enums As Ints*	-b	275
Word Alignment	-a	275
Duplicate Strings Merged	-d	275
Unsigned Characters	-K	277
Precompiled Headers	-H	286, 635
Assume SS Equals DS		273
Default For Memory Model		589
Never		273
Always		274
Defines		274
Advanced Code Generation		
Floating Point		
None	-f-	276
Emulation*	-f	276
8087	-f87	277
80287	-f287	277

Table 2.9: IDE menu cross-reference (continued)

Instruction Set		
8088/8086	-1-	275
80186	-1	275
80286	-2	275
Options		
Generate Underbars*	-u	278
Line Numbers Debug Info	-y	279
Debug Info In OBjs*	-v	278
Fast Floating Point*	-ff	276
Fast Huge Pointers	-h	277
Generate COMDEFS	-Fc	275
Automatic Far Data	-Ff	276
Far Data Threshold	-Ff=size	276
Entry/Exit Code		
Prolog/Epilog Code Generation		
Standard*	-Y-	
Overlay	-Y	279, 589
Calling Convention		
C*	-P-	278
Pascal	-P	278
Stack Options		
Standard Stack Frame*	-k	278
Test Stack Overflow	-N	278
C++ Options		
Use C++ Compiler		
CPP Extension*	-P- cpp	286
C++ Always	-P	286
C++ Virtual Tables		
Smart*	-V	287
Local	-Vs	288
External	-V0	288
Public	-V1	288
Template Generation		
Smart*	-Jg	290
Global	-Jgd	290
External	-Jgx	290
Options		
Out-Of-Line Inline Functions*	-vi	279
Far Virtual Tables	-Vf	288
Optimization Options		
Optimizations		
Suppress Redundant Loads	-Z	280
Jump Optimization	-O	280
Register Variables		
None	-r	280
Register Keyword	-rd	280
Automatic*	-r	280
Optimize For		
Size*	-G-	280
Speed	-G	279
Source		

Table 2.9: IDE menu cross-reference (continued)

Keywords		
Turbo C++*	-A-, -AT	281
ANSI	-A	281
UNIX V	-AU	281
Kernighan & Ritchie	-AK	281
Source Options		
Nested Comments	-C	281
Identifier Length (32 by default)	-in	281
Messages		
Display		
Display Warnings		
All	-w	282
Selected	-wxxx	282
None	-w-	
Errors: Stop After	-jn	282
Warnings: Stop After	-gn	282
Portability		
Non-portable Pointer Conversion*	-wrpt	
Non-portable Pointer Comparison*	-wcp	
Constant Out Of Range In Comparison*	-wrng	
Constant Is Long	-wcln	
Conversion May Lose Significant Digits	-wsig	
Mixing Pointers To Signed And Unsigned Char	-wucp	
ANSI violations		
Void Functions May Not Return A Value*	-wvoi	
Both Return And Return Of A Value Used*	-wret	
Suspicious Pointer Conversion*	-wsus	
Undefined Structure 'ident'*	-wstu	
Redefinition Of 'ident' Is Not identical*	-wdup	
Hexadecimal Value More Than Three Digits*	-wbig	
Bit Fields Must Be Signed Or Unsigned Int	-wbbf	
'ident' Declared As Both External And Static*	-wext	
Declare 'ident' Prior To Use In Prototype*	-wdpu	
Division By Zero*	-wzdi	
Initializing 'ident' With 'ident'*	-wbei	
Initialization Is Only Partially Bracketed	-wpin	
C++ Warnings		
Base Initialization Without A Class Name Is Obsolete*	-wobi	
Functions Containing 'ident' Are Not Expanded Inline*	-winl	
Temporary Used To Initialize 'ident'*	-wlin	
Temporary Used For Parameter 'ident'*	-wlvc	
Constant Member 'ident' Is Not Initialized*	-wnci	
This Style Of Function Definition Is Now Obsolete*	-wofp	
Use Of Overload Is Now Unnecessary And Obsolete*	-wovl	
Assigning 'type' to 'enumeration'*	-wbei	
'Function1' Hides Virtual Function 'Function2'*	-whid	
Non-const Function 'ident' Called For Const Object*	-wncf	
Base Class 'ident' Inaccessible Because Also In 'ident'*	-wibc	
Overloaded Prefix Operator Used As Postfix Operator*	-wpre	
Array Size For Delete Ignored*	-wdsz	
Use Qualified Name To Access Nested Type 'ident'*	-wnst	

Table 2.9: IDE menu cross-reference (continued)

Frequent Errors		
Function Should Return A Value*	-wrvl	
Unreachable Code*	-wrch	
Code Has No Effect*	-weff	
Possible Use Of 'ident' Before Definition*	-wdef	
'ident' Is Assigned A Value That Is Never Used*	-wauS	
Parameter 'ident' Is Never Used*	-wpar	
Possibly Incorrect Assignment*	-wpia	
Less Frequent Errors		
Superfluous & With Function	-wamp	
Ambiguous Operators Need Parentheses	-wamb	
Structure Passed By Value	-wstv	
No declaration For Function 'ident'	-wnod	
Call To Function With No Prototype	-wpro	
Unknown Assembler Instruction	-wasm	
Ill-formed Pragma*	-will	
Condition Is Always (True/False)*	-wccc	
Array Variable 'ident' Is Near*	-wias	
'ident' Declared But Never Used	-wuse	
Names		
Code Segment	-zC	284
Code Group	-zP	285
Code Class	-zA	284
Data Segment	-zR	285
Data Group	-zS	285
Data Class	-zT	285
BSS Segment	-zD	284
BSS Group	-zG	285
BSS Class	-zB	284
Far Data Segment	-zE	284
Far Data Group	-zH	285
Far Data Class	-zF	284
Transfer		
Program Titles		
~GREP		259
Make		
Break Make On		297
All Sources Processed	MAKE -i	300
Check Auto-dependencies*	MAKE -a	300
Linker		
Settings		331
Options		
Initialize Segments	TLINK /i	340
Default Libraries*	TLINK /n	336, 342
Warn Duplicate Symbols	TLINK /d	339
"No stack" Warning*		335
Case-Sensitive Link*	TLINK /c	338
Map file		
Off*	TLINK /x	340
Segments	TLINK	340

Table 2.9: IDE menu cross-reference (continued)

Publics	TLINK /m	340
Detailed	TLINK /s	340
Output		
Standard EXE*	TLINK /Td	332
Overlaid EXE	TLINK /o	342, 589
Libraries		
Container Class		CLASSLIB.DOC
Graphics Library		607
Standard Run Time*		Online Help
Librarian		
Options		
Generate List File	TLIB	UTIL.DOC
Case-sensitive Library	TLIB /C	UTIL.DOC
Purge Comment Records	TLIB /O	UTIL.DOC
Create Extended Library	TLIB /E	UTIL.DOC
Library Page Size	TLIB /Psize	UTIL.DOC
Debugger		
Source Debugging		
On*	-y TLINK /v	279 344
Directories		
Include Directories	-Ipath	293
Library Directories	-Lpath	293
Output Directory	TLINK -Lpath -npath	340 291
Environment		
Preferences		
Command Set		629
Editor		629
Syntax Highlighting (toggle on/off)		37
Startup		
Video Startup Options		
Dual Monitor Mode	TC /d	21
Video Mode		
Black & White / LCD	TC /l	22
Swap File Drive	TC /r	22
Use Extended Memory	TC /x	22
	-Qx=nnnn	287
	TLINK /yx	345
Use EMS Memory*	TC /e	21
	-Qe	287
	TLINK /ye	344
Colors		
Edit Window (Syntax Highlighting)		37

* enabled by default

An introduction to C++

If you have never programmed in C or C++, this chapter is for you.

The best way to learn anything new is to start at the beginning. In the case of C++, which began as an extension to the C programming language, a knowledge of C is generally considered a necessary starting point. This tutorial takes a somewhat different approach, which makes it suitable for programmers and novice programmers who want to learn C++ without first mastering C. We can take this approach because, even though C++ supports a radically different style of programming (known as *object-oriented programming*), the basic language elements are, with few exceptions, the same as C. What C++ adds to C is mostly a number of high-level features to support this new programming style.

If you are an experienced C programmer, you may want to scan this chapter briefly, then skip to Chapter 4, "Object-oriented programming with C++."

In this chapter you will develop a working knowledge of the fundamental language features of C++, many of which were inherited from C. The program examples used here are designed to illustrate the key concepts of C++ as clearly as possible. This chapter is a necessary first step that will lead to an understanding of the more complex and abstract principles of object-oriented programming found in later chapters.

You will be introduced to a substantial body of new terminology and concepts. There are sample programs that demonstrate how these concepts are used. The best way to learn to program in any language is to compile and run the sample programs. When you understand how a particular sample program works, change it, expand it, play with it.

The sample programs are included in the *EXAMPLES* subdirectory.

You'll solve a variety of problems involving numbers, words, and graphics. We also provide some guidelines for designing and structuring programs.

How to run the examples

You can follow along with the examples in this chapter by compiling and running the designated programs. To run the example programs using the Turbo C++ IDE, follow these steps:

1. From the *EXAMPLES* subdirectory, start Turbo C++ by typing `TC`.
2. Open the example file you wish to compile by selecting **File | Open | *example_file_name*** from the IDE menu.
3. Run the example program by selecting **Run | Run**.
4. To see the program's output, select **Window | User** screen.

Alternatively, you may compile and run the examples from the DOS command line. In the *EXAMPLES* directory, give the command

```
TCC example_file_name <Enter>
```

After the example is compiled, you may run it and view the output by typing

```
example_file_name <Enter>
```

Basic programming operations

Computer programs vary greatly in purpose, style, and complexity. Nearly all programs, however, go through a process consisting of three phases:

- describing, collecting, and storing information (data)
- processing the data to achieve the desired result
- formatting, displaying and/or storing the results

Any data used by a program has to be described so that Turbo C++ knows how to store and retrieve it. Memory must be set aside to hold the amount of data expected. The program must

then use some means to get the actual data into storage — this could involve reading the characters from the keyboard, retrieving data from a file on disk, receiving data over a telephone line, or using some other kind of input device.

Once the data has been stored in numeric variables, character strings, arrays, or more complicated data structures, it must be processed. The processing varies with the purpose of the program, of course: A spreadsheet program might apply a formula to a set of data to calculate a result, while a word-processing program might rearrange lines of text to fit new margins.

Once the data is processed, the results must be made available in some way to the user. Lines of text can be rearranged on the screen or sent to the printer, and the spreadsheet cells can be re-displayed to show their new values. Most data must eventually be stored on disk for later use.

Let's use the three phases of program design in this short example program.

To try out this program, load and run `INTRO1.CPP` (File | Open | `INTRO1`), which can be found in the `EXAMPLES` subdirectory. For more about how to load and run the example programs refer to chapter 2, "IDE basics."

```
//INTRO1.CPP--Example from Chapter 3, "An Introduction to C++"  
#include <iostream.h>  
  
int main()  
{  
    int bushels;  
    float dollars, rate;  
    cout << "How many dollars did you get? $";  
    cin >> dollars;  
    cout << "For how many bushels? ";  
    cin >> bushels;  
    rate = dollars / bushels;  
    cout.precision(2);  
    cout << "You have received $" << rate << " for each bushel.\n";  
    return 0;  
}
```

The first line of this program, `//INTRO1.CPP -- Example from Chapter 3 " An Introduction to C++"` is a *comment*. Comments are preceeded by the `//` symbol and continue to the end of the line. (Turbo C++ also recognizes the older C-style notation for comments, which begin with `/*` and end with `*/`. This style of comments may include more than one line.) Comments are ignored by the compiler. The comment symbol tells Turbo C++ to ignore all characters from the `//` symbol to the end of the line. You use comments to describe the purpose of a program, function, or statement. Appropriate comments make it easier for you to

remember just what a particular part of your program does — and it helps other programmers who may later be called on to modify your program.

Header files are also called
include files.

The second line, `#include <iostream.h>`, tells the compiler to add the Turbo C++ *header* file `iostream.h` to `INTRO1.CPP` before compiling. `iostream.h` contains the declarations and functions for the `iostream` input and output library. A library is a collection of ready to use functions which your program can call to take care of basic programming chores. We need `iostream.h` because it contains the information which allows us to use the **cout** and **cin** operators described below.

The next line, `int main()` defines a function. A function is a group of related program instructions. Every C++ program must have a `main()` function, which is where program execution begins. Functions are the building blocks of C++ programs. The open brace “{” indicates the beginning of a group of program instructions, or *statements*—in this case, the statements which define what will happen when the function **main()** is executed. Each group of statements ends with a closing brace “}”.

Notice that each statement ends in a semicolon (;). While Turbo C++ lets you string several statements together on the same line, we don't recommend this; it makes programs harder to read.

Describing the data

The first two statements in `INTRO1.CPP` are

```
int bushels;  
float dollars, rate;
```

Recall that the first step in writing a program is “describing, collecting, and storing information.” In C++, you must declare each item of data before you can do anything else with it. To declare an item of data, list what *type* of data it is, then give it a name. Here, you have one data item that is of *type* **int** (integer, or whole number), and is named *bushels*. You also have two data items, *dollars* and *rate*, that are of *type* **float** (a floating-point number is a number that has a decimal fraction). Notice that we have declared two data items, *dollars* and *rate*, in the same statement. You must separate the data items with a comma, and the data items must be of the same *type*—in this case both are of *type* **float**. These data items are also called *variables*, since their value may be updated or changed as the program runs.

Collecting and storing the
data

The next four statements obtain and store the data we've just described. The **cout** statements prompt for the number of bushels

and the number of dollars received for those bushels, and the **cin** statements get these values and store them in the variables named. Most of the actual work done in a Turbo C++ program is accomplished by calling upon the functions provided in the libraries included with Turbo C++. In this example our input and output are handled by the *iostream* library. For now it is sufficient to know that **cout** is the *standard output stream* (normally your computer's monitor screen), and **cin** is the *standard input stream* (normally the keyboard). The operators **<<** and **>>** are the insertion ("put to"), and extraction ("get from") operators. Operators are symbols that tell the computer to perform some type of basic operation on your data. The insertion and extraction operators are members of an extensive set of operators provided by Turbo C++.

We will discuss other operators later in this chapter.

Processing the data

The statement, `rate = dollars/bushels`, does the processing part of the program, dividing the number of dollars by the number of bushels to get the dollars per bushel.

Formatting the data

The statement, `cout.precision(2)`, is a call to a member function declared in *iostream.h* which formats our data for output. `cout.precision(2)` is a special function which formats the way our data will be displayed. `cout.precision(2)` affects all floating-point input/output operations until its next use. For example, you might want to display four decimal places at some other point in the program, in which case you would use the statement `cout.precision(4)`. In `INTRO1.CPP`, `cout.precision(2)` specifies that the floating-point numbers displayed by **cout** should be rounded off to two decimal places, since we want to represent dollars and cents. The **cout** operator defaults to six decimal places if no provision is made to format the output. (Even though **cout** defaults to six decimal places for floating point numbers, it will automatically drop any trailing zeros).

Displaying the data

The final statement,

```
cout << "You have received $ " << rate << " for each bushel\n";
```

once again uses **cout** to display the results of this calculation. The first **<<** (insertion) operator directs the output of the phrase "You have received " to the screen. The monitor screen is the default output device of the **cout** operator. The second **<<** operator outputs the value of the variable `rate`, and the third **<<** operator directs the phrase " dollars for each bushel" to the screen. The `\n`

is the newline symbol which places the cursor at the beginning of the next line.

When you run the program, the output looks like this:

```
How many dollars did you get? $32
For how many bushels? 24
You have received $1.33 for each bushel
```

Basic structure of a C++ program

This next example demonstrates functions, variables, and the preprocessor directive **#include**.

Load and run `INTRO2.CPP`.

```
// INTRO2.CPP--Example from Chapter 3, "An Introduction to C++"
// INTRO2.CPP calculates a sales slip.

#include <iostream.h>
float tax (float);

int main()
{
    float purchase, tax_amt, total;
    cout << "\nAmount of purchase: ";
    cin >> purchase;

    tax_amt = tax(purchase);
    total = purchase + tax_amt;
    cout.precision(2);
    cout << "\nPurchase is: " << purchase;
    cout << "\nTax: " << tax_amt;
    cout << "\nTotal: " << total;

    return 0;
}

float tax (float amount)
{
    float rate = 0.065;
    return(amount * rate);
}
```

The first and second line of the program are comments. Line three is a *preprocessor directive* which tells Turbo C++ to read in and compile the contents of the header file `iostream.h`.

The general format of a function declaration is:
return_type
function_name(parameter_type
(parameter_name) ...);

The next statement is a *function declaration*, it declares or describes the user-defined function `float tax(float)`. Function declarations, also called *prototypes*, give the compiler important information about the function so it can recognize and use it in the program.

The first word of the function declaration, `float`, specifies that this function will *return* a value of type **float** (a floating-point number). When we say that a function will *return* a value, we mean that the function will pass its answer back to the program when it has finished its work. The word `tax` tells the compiler the name of the function, and the word in parentheses (`float`) tell the compiler that this function will expect a floating point number as input. Inputs to a function are called *arguments*. If there are several *arguments* then you must separate them with commas. The parentheses delimit or surround the *argument list* (a function does not have to take *arguments*, in which case the parentheses are left empty or the word *void* is placed inside the parentheses). Finally we terminate the function declaration with a semicolon. Putting the function declaration at the top of the program helps Turbo C++ make sure that your program doesn't try to give the function the wrong kind of data (a character string, for example).

```
float purchase, tax_amt, total;
```

declares three floating-point variables (*purchase*, *tax_amt* and *total*). The next two statements, beginning with **cout** and **cin**, prompt for and obtain the amount of purchase.

Now it's time for the actual computing.

```
tax_amt = tax(purchase);
```

is a *function call*. It *calls* the user-defined **tax()** function, passing it a value based on *purchase*. To find out what the function does, skip down to the bottom of the program, where you see its definition:

```
float tax (float amount)
{
    float rate = 0.065;
    return(amount * rate);
}
```

This specifies that the **tax()** function takes the floating point value (in this case, *purchase*) that it receives from the *calling statement* and places that value into its argument *amount*, multiplies *amount* by the defined variable *rate*, and returns the result back to the *calling statement*. Thus, when the line

```
tax_amt = tax(purchase);
```

is executed, the tax on the amount of *purchase* is calculated and returned by the **tax()** function and then stored in the variable

tax_amt for later use. In the next line this amount is added to *purchase* to obtain *total*.

It may seem unnecessary to have a whole separate function just to calculate the tax, and it is in this program. But it becomes useful in more complicated situations, such as when there are several tax rates to choose from according to the purchaser's county of residence. Perhaps you also have to check a product code to determine whether the item is taxable in the first place. In that case, separating the mechanism for figuring tax makes the main part of the program easier to follow. If necessary, you can later change how the tax is calculated without affecting the rest of the program.

The final four lines of the main program set the floating-point precision to 2 decimal places, then print out the purchase amount, tax, and total. A sample run looks like this:

```
Amount of purchase:  
Purchase is: 24.95  
Tax: 1.62  
Total: 26.57
```

Working with numbers

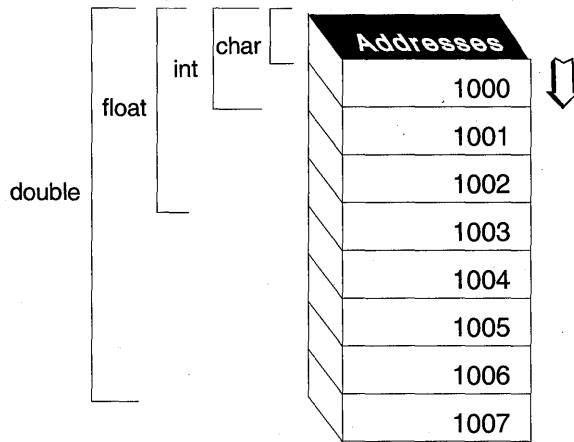
Numbers are the fundamental data used by computers. The actual contents of computer memory consists of binary numbers. These are usually organized in groups of 8 bits (1 byte) or 16 bits (2 bytes, or 1 *word*). Even those computing activities that involve words or graphics basically involve series of numbers stored in memory.

Numeric data types

The same part of memory could be interpreted as several different kinds of numbers, depending on how many bytes are grouped together. The name of a variable, such as *total*, actually refers to the contents of one or more bytes following a specific address in memory — this address is assigned by Turbo C++ when you first define (or initialize) the variable. But you and the compiler must agree about what kind of number will be represented by a given variable, and thus how many bytes will be stored and fetched starting at the variable's address. You make this agreement by specifying a *data type* when you declare the variable. For example,

```
int    total, count, step;
float  cost, profit;
```

Figure 3.1
Interpreting memory
locations as numbers (in
1-byte increments)



Each data type represents a different kind of number. You must choose an appropriate type for the kind of number you need. In this variable declaration,

- The variable *total* is of type **int** (integer). When you tell your program to use the value of *total* in a statement, it fetches 2 bytes, starting at *total*'s address.
- The variable *cost* is of type **float** (floating point). When your program uses *cost*, it fetches 4 bytes, starting at *cost*'s address. This is because a floating-point number needs the two extra bytes to represent the significant digits of the number and the magnitude of the number in terms of powers of two.

Table 3.1 shows the basic Turbo C++ data types and their variations. Notice the variety of numbers that can be accommodated. This chapter shows you how to use many of these data types.

Table 3.1: Data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	**	**	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4×10^{-38} to 3.4×10^{38}	Scientific (7-digit precision)
double	64	1.7×10^{-308} to 1.7×10^{308}	Scientific (15-digit precision)
long double	80	3.4×10^{-4932} to 1.1×10^{4932}	Financial (19-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

** In C++ **enum** can be of any type, consequently, its range is consistent with the type declared.

Integers The basic integer type is **int**, which can express either negative or positive numbers, but within a limited range (-32,768 to 32,767). Here's an example program that performs some operations with integers:

To try out this program load and run *INTRO3.CPP*

```
//INTRO3.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int pounds;
    int total, bags;
    pounds = 50;
    bags = 1000;
    total = bags * pounds;
    cout << "There are " << total << " lbs. in 1000 bags of beans\n";
    return 0;
}
```

The output from this program is a little surprising:

```
There are -15536 lbs. in 1000 bags of beans
```

The compiler doesn't always warn you if you try to store a value that doesn't fit into the specified data type.

Why is this? It is because the total of *bags * pounds*, 50,000, is too large for an ordinary **int** (which you can verify from Table 3.1). When your program tried to store 50,000 in a type that could only hold 32,767, the result overflowed. How do you solve this problem? Use **long int**.

Integer modifiers

The long modifier

long int, which is usually abbreviated just as **long**, gives you a larger integer range. You can solve the problem of the negative beans by declaring:

```
long total;
```

This gives you room for more pounds of beans than you'd ever be likely to see, because a **long**, which is stored in 32 bits instead of the 16 used by an ordinary **int**, can accommodate a value between -2,147,483,648 and 2,147,483,647. But what about *pounds*? This variable should be fine as an **int**, since the weight of one bag is unlikely to exceed 32,767 pounds. The variable *bags*, however, might conceivably exceed 32,767, so make it a **long** also. Why not use **long** instead of **int** variables for everything? A **long** takes up 4 bytes of memory, while an **int** takes only 2. If you have many variables, you'll end up wasting a lot of memory.

```
int pounds;  
long bags, total;
```

The signed and unsigned modifiers

All the data types listed in Table 3.1 are **signed** by default—one of the bits in the stored value is used to indicate whether the number is positive or negative. (Those marked **unsigned** are, of course, explicitly **unsigned**.) Some values encountered in your work can be either positive or negative — for example, temperatures and bank balances. Many other values, however, are never negative—a business can't have a negative number of employees, for example. By adding the word **unsigned** to any data type, you restrict its range to positive numbers. Since a sign bit is no longer needed, this doubles the maximum value stored by the type. For example, while an ordinary (**signed**) **int** ranges from -32,768 to 32,767, an **unsigned int** ranges from 0 to 65,535. (An **unsigned long** ranges between 0 and 4,294,967,295.) The preceding program example would also have worked correctly if you had declared

To decide which data type to use, consider the possible results of a calculation or other operation.

Since the default for all numeric data types is signed, you don't have to declare **signed int** or **signed long**.

```
unsigned int total;
```

though you'd be getting uncomfortably close to the limits of the **unsigned int** type.

Floating-point numbers

Many numbers involve a fractional part set off with a decimal point, such as prices in dollars and cents. These are called *floating-point numbers* (also often called *real numbers*). Most exact measurements involve fractions: If you buy screws at the hardware store, you'll probably have to specify the diameter in fractions of an inch. The **float** data type covers such situations. Here's an example:

To try out this program, load and run `INTRO4.CPP`.

```
//INTRO4.CPP--Example from Chapter 3.
#include <iostream.h>

int main()
{
    float num, denom;    // numerator and denominator of fraction
    float value;        // value of fraction as decimal

    cout << "Convert a fraction to a decimal\n";
    cout << "Numerator: ";
    cin >> num;
    cout << "Denominator: ";
    cin >> denom;

    value = num / denom; // convert fraction to decimal

    cout << "\n" << num << "/" << denom << " = " << value;

    return 0;
}
```

The program prompts for the numerator and denominator of a fraction, then converts them to a decimal value and prints the result. For example,

```
Convert a fraction to a decimal
Numerator: 7
Denominator: 8

7/8 = 0.875
```

Clearly *value* must be a **float** in order to hold a fraction, but you may not realize that either *num* or *denom* (or both) has to be of the **float** type if you wish to divide *num/denom* correctly. Try this example:

If you divide integers by integers, the result is rounded down to the nearest whole number before it is assigned to the float variable.

```
// INTRO5.CPP -- Example from chapter 3 "Introduction to C++  
#include <iostream.h>  
int main()  
{  
    int num = 3, denom = 4;  
    float value;  
    value = num / denom;  
    cout << value;  
  
    return 0;  
}
```

The result is zero — not the 0.75 you'd expect.

The floating-point types

The **double** and **long double** types are like **float**, only they accommodate larger numbers with more precision. Precision is important in both scientific and financial calculations. When you declare a variable give some thought to the kind of numbers the variable will represent, this will ensure that the results of the program execution will be meaningful.

Variables

As you have learned, every variable must be declared before it can be used. A *declaration* consists of a data type followed by one or more variable names. Declarations simply tell Turbo C++ that you intend to use a particular variable, and what type of data it will store.

```
int hours;  
float total_pay, pay_rate;  
long id_number;
```

Initializing variables

You also need to *initialize* a variable—set it to a specific value, such as 0. What do you think the following program will display?

```

#include <iostream.h>

int main()
{
    int something;
    cout << something;
    return 0;
}

```

The result will vary — on our machine, it was -32,417. Did you notice that the program did not assign any value to the variable *something* before trying to print it out? With the exception of global or static variables (discussed later), variables in C++ do not have a default value. The value of *something*, therefore, is whatever number happens to be stored at the address Turbo C++ assigned to the variable. This value is unpredictable. In fact, if you compiled this program, you might have noticed a warning in the Message window: "Possible use of 'something' before definition in function **main**." When you get this warning, you should check the variable named to make sure you initialize it before you use it for anything.

Assignment statements

You give a value to a variable with an *assignment statement*. Assignment consists of a variable name followed by an equals sign and the value to be assigned. Here are some examples:

```

count = 0;
total = purchase + tax_amt;
tax_amt = tax(purchase);

```

In the first statement, an actual number, or *numeric constant*, is assigned to the variable *count*. The second statement uses an expression to assign the sum of *purchase* and *tax_amt* to the variable *total*. An *expression* is any combination of values and *operators* (such as + or *) that yield a single value. In C++, you can use an entire expression just about anywhere that you can use a single numeric value. You can assign it to a variable, send it to a function for processing, or display it with **cout**.

The third statement is slightly more complex: it first calls the function **tax**, giving it the value of the variable *purchase*. The function uses this value and other information to calculate the tax, then returns the result. In other words, the function call **tax(purchase)** is *evaluated*, then replaced by an actual value, such as 1.14. Finally, the assignment operator "=" assigns this value to

tax_amt. Assignment statements using function calls are very common in C++.

Combination assignments

C++ often lets you combine two or more distinct operations in a single statement. You can declare a variable and assign it a value in a single statement. Instead of

```
float total_expenses;  
total_expenses = 0;
```

most C++ programmers write

```
float total_expenses = 0;
```

You can also assign several variables the same value in one statement. A word processor might start processing text by setting

```
page = line = column = 1;
```

This works because an assignment statement not only assigns a value, it also provides a value that can be used by other parts of a statement in which it is embedded. That is, `column = 1` assigns 1 to *column*, and makes this value, 1, available. Moving right to left, we get the equivalent of `line = 1`. In turn, that assignment passes on the value 1, so the final assignment is `page = 1`.

But don't go overboard. It is often better to declare and initialize one variable per statement, so you can include a comment describing the purpose of each variable:

```
int lines = 0;    // Lines of text, ending in new line char  
int words = 0;   // Words are groups of characters surrounded  
                 // by space, tabs, or new lines  
int chars = 0;   // Every character is counted
```

Taking the time to do this might also alert you to potential problems. For example, is it really a good idea for *chars* to be an `int`?

Variable names

It's time now to consider what names you can give to variables. C++ is quite flexible in this regard. User-supplied names (called *identifiers*) must follow these rules:

- All identifiers must start with a letter (*a* to *z* or *A* to *Z*) or an underscore (`_`).

- The rest of the identifier can use letters, underscores, or digits (0 to 9). Other characters (such as punctuation marks or control characters) cannot be used. C++ identifiers are significant to any length.
- Identifiers are case sensitive. This means that *amount* and *Amount* are completely separate variables.

By these rules, *deduction*, *tax_status*, and *amt_1099* are all legal identifiers, while *1989_tax* and *stop!* are not. (*1989_tax* begins with a digit instead of a letter or underscore, and *stop!* contains an exclamation point, which is not a letter, underscore, or digit.)

Besides following the rules, it is important to give some thought to naming your variables. Here are some suggestions:

- The name should describe what the variable contains. *a* doesn't tell you anything. *amt* is better, but *taxable_amount* is most clear and specific.
- Use capital letters or underscores to separate words in a long identifier. *PricePer100* or *price_per_100* are much easier to read than *priceper100*.
- Use comments to describe the nature and purpose of a variable, particularly if it is not obvious.

More about input and output

Formatting with escape sequences

There are a number of characters that control how text appears onscreen; for example, the tab character advances the cursor to the next tab position, the newline character moves the cursor to the next line, and the formfeed starts a new screen or page of text. **cout** lets you include any of these characters (and others) in the text to be printed, simply by prefixing the symbol for the character with a backslash (\). The backslash is called an *escape* because it tells Turbo C++ to interpret the following character not as a literal *n* or *f* or whatever, but as the symbol for a special character.

Indeed, you have already seen numerous examples using **\n** in a string being displayed with **cout**. While the print statement in languages such as BASIC automatically advances the cursor or print head to the next line, there is no such default in C++. This gives you more flexibility, since you can use separate statements within **cout** to display text on the same line, and advance to the next line only when you specifically wish to. The next table lists Turbo C++'s escape sequences.

Table 3.2
Character escape
sequences

Sequence	Name	Meaning
\a	Alert	Sounds a beep
\b	Backspace	Backs up one character
\f	Formfeed	Starts a new screen or page
\n	Newline	Moves to beginning of next line
\r	Carriage return	Moves to beginning of current line
\t	Horizontal tab	Moves to next tab position
\v	Vertical tab	Moves down a fixed amount
\\	Backslash	Displays an actual backslash
\'	Single quote	Displays an actual single quote
\"	Double quote	Displays an actual double quote
\?	Question mark	Displays an actual question mark
\OOO		Displays a character whose ASCII code is an octal value (one to three digits)
\xHHH		Displays a character whose ASCII code is a hexadecimal value (one or more digits)

- "Newline" on MS-DOS systems is equivalent to a carriage return (CR) plus a linefeed (LF). This is not true of some other systems.
- A backslash in front of the single and double quotes is needed only when Turbo C++ would otherwise interpret these characters as having a special meaning. For example, " normally delimits a string. To print a string in quotes, use `"\"a string in quotes\""`.
- The octal or hexadecimal values are often used to send special graphics characters or printer control characters. For example, `printf("\xDB")` on the IBM PC displays a solid square character.

After `\n`, the most commonly used escape sequence is probably `\t`, the tab character. It is useful for aligning tables of numbers. For example, this code:

To try this out, load and run `INTRO6.CPP`.

```
// INTRO6.CPP--Example from Chapter 3, "Introduction to C++"
#include <iostream.h>

int main()
{
    int i = 101, j = 59, k = 0;
    int m = 70, n = 85, p = 5;
    int q = 39, r = 110, s = 11;

    cout << '\t' << "Won" << '\t' << "Lost" << '\t' << "Tied\n\n";
    cout << "Eagles" << '\t' << i << '\t' << j << '\t' << k << '\n';
    cout << "Lions" << '\t' << m << '\t' << n << '\t' << p << '\n';
}
```

```

    cout << "Wombats" << '\t' << q << '\t' << r << '\t' << s << '\n';
    return 0;
}

```

produces the following neatly formatted table:

	Won	Lost	Tied
Eagles	101	59	0
Lions	70	85	5
Wombats	39	110	11

Arithmetic operators

Now that you know how to get and display values for different kinds of variables, let's look more closely at the variety of operators provided by Turbo C++. You are already familiar with several operators: the assignment operator (=) and four arithmetic operators (+, -, *, and /, for addition, subtraction, multiplication, and division, respectively).

These operators work pretty much the way you would expect them to, though with some differences. For example, dividing two **int** values gives you an **int** result, with any fraction dropped. There is also a specific order, called *precedence*, in which operators take effect. For arithmetic operators, multiplication and division come before addition and subtraction. Try to guess the four numbers that will be displayed by this program (to try it out, load and run INTRO7.CPP)

```

// INTRO7.CPP--Example from Chapter 3, " An Introduction to C++"
#include <iostream.h>

int main()
{
    float result;
    result = 1.0 + 2.0 * 3.0 / 4.0;
    cout << '\n' << result;
    result = 1.0 / 2.0 + 3.0;
    cout << '\n' << result;
    result = (1.0 + 2.0) / 3.0;
    cout << '\n' << result;
    result = (1.0 + 2.0 / 3.0) + 4.0;
    cout << '\n' << result;

    return 0;
}

```

It doesn't hurt to use parentheses, even if they aren't strictly needed. They can make expressions easier to read.

Here they are. How did you do?

2.5
3.5
1
5.666667

In the first expression, the multiplication $2.0 * 3.0$ is done first, yielding 6. Next, $6 / 4.0$ gives 1.5, which is finally added to 1.0 to get the final result, 2.5. Notice that when operators have equal precedence ($*$ and $/$ have equal precedence, as do $+$ and $-$), operations are done from left to right.

In the second expression, the division is done first, then the addition, so the result is $0.5 + 3$, or 3.5.

In the third expression, $(1.0 + 2.0)$ is in parentheses, so it is performed first: The result, 3.0, is then divided by 3 to get 1.

Finally, the last expression places $1.0 + 2.0 / 3.0$ within parentheses. Within the parentheses, the usual rules are followed: 2 is divided by 3, and then added to 1. The result, 1.666667, is then added to 4 to get 5.666667.

modulus (%)

The *modulus* operator (%) divides two numbers and keeps only the remainder. For example, the expression $5 \% 2$ gives a result of 1, while $18 \% 3$ gives 0, since 3 divides evenly into 18.

Arithmetic and type conversion

What happens if you add an **int** to a **float**? You would want the result to be a **float** so that any fractional part is retained, and that is what happens. Turbo C++ *promotes* smaller types to larger ones according to a set of rules (see the next table). From the table, you can see that when an **int** and a **float** are added, the **int** is promoted to a **float**. The two numbers are then added, resulting in a **float**.

Table 3.3
Type promotions for
arithmetic

Some types in this table have
not been discussed yet.

Type	Converts to
These types are converted automatically:	
char	int
unsigned char	int
signed char	int
short	int¹
unsigned short	unsigned int¹
enum	int
float	double
Then these rules are applied <i>in this order</i> , until both operands have the same type:	
<i>If either operand is...</i>	<i>The other is converted to...</i>
long double	long double
double	double
float	float
unsigned long	unsigned long
long	long
unsigned	unsigned

¹This is an ANSI requirement. However, **short** and **int** are the same size for all C++ compilers on the PC, so no conversion is done.

Typecasting

It is sometimes useful or necessary to explicitly convert a data item to a specified type. For example, if you have

```
#include <iostream.h>

int main()
{
    int a = 5, b = 2;
    cout << a / b;
    return 0;
}
```

you'll get a result of 2, since integer division drops any fractional part. If, however, you do it this way:

```
#include <iostream.h>

int main()
{
    int a = 5, b = 2;
    cout << (float)a / (float)b;
    return 0;
}
```

the values *a* and *b* will be converted to the type enclosed in parentheses (**float** in this case) before the division, so the value of the expression will be 2.5. This forced conversion is called a *type cast*, or just a *cast*.

Combining arithmetic and assignment

A common operation in programming involves adding a fixed amount to a variable (*incrementing* it). For example, if a program is counting words, when it finds a word, it will do something like `total_words = total_words + 1`. Later, you will also see how loops usually involve repeatedly adding or subtracting a number until a variable reaches a specified limit.

A shorthand way of doing things in C++ is to perform arithmetic and assignment in one step. You can combine any binary arithmetic operator with the assignment operator. The preceding statement can also be written as `total_words += 1`. Read this as "add 1 to the current value of *total_words* and assign this quantity as the new value of *total_words*." Similarly, a checkbook-balancing program might execute the statement `balance -= check_amt` (subtract the amount of the check from the balance and make that the new value of *balance*). The somewhat less common combinations `*=` and `/=` work in the same way.

Increment and decrement

Adding and subtracting exactly one is so common an operation that two special operators, increment "`++`" and decrement "`--`", are provided for the purpose. Thus, `++total_words` does exactly the same thing as `total_words += 1`. A program that does a countdown for the space shuttle might use `count--` in a loop until zero is reached.

The increment and decrement operators can come either before or after the affected variable. When the operator comes before the variable it is applied to the variable *first*, and then the result is used in the expression as a whole; this is called *prefix notation*. When the operator comes after the variable, the value of the variable is used first, and *then* the operator is applied to the variable; this is called as *postfix notation*. For example, look at `INTRO8.CPP`:

Load and run `INTRO8.CPP`

```
//INTRO8.CPP--Example from Chapter 3, " An Introduction to C++"  
#include <iostream.h>
```

```

int main()
{
    int val = 1;

    cout << "\nval is " << val++ << " and then post-incremented\n";
    cout << "val is now " << val << '\n';
    cout << "val is pre-incremented to " << ++val << '\n';

    return 0;
}

```

which gives the results

```

val is 1 and then post-incremented
val is now 2
val is pre-incremented to 3

```

In the first **cout** statement, *val* is still 1 when it is printed, but becomes 2 afterward, as shown in the second **cout**. In the third statement, *val* is incremented first, so it is 3 by the time it is displayed by the third **cout**.

Working bit by bit

Sometimes you'll find that you have to manipulate the actual bits that make up each byte in memory. C++ provides a set of bitwise operators, shown in the next table.

Table 3.4
Bit manipulation operators

Operator	Meaning
Operators that take two operands:	
&	AND; if both bits are 1, result is 1.
	OR; if either bit is 1, result is 1.
^	Exclusive OR; if only 1 bit is 1, result is 1.
>>	For a signed int , shift bits right the number of times specified by following number; fill in 1s at left if the number is negative, 0s if positive. For an unsigned int , fill in 0s to left.
<<	Shift bits left the number of times specified by following number; fill in zeros at right.
Operator that has a single operand (unary operator):	
~	1's complement; reverse all bit values.

The following program illustrates the use of these operators:

Load and run *INTRO9.CPP*

```

//INTRO9.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()

```

```

{
    cout << "1 & 1 is " << (1 & 1) << '\n';
    cout << "1 | 1 is " << (1 | 1) << '\n';
    cout << "1 ^ 1 is " << (1 ^ 1) << '\n';
    cout << "255 << 2 is " << (255 << 2) << '\n';
    cout << "255 >> 2 is " << (255 >> 2) << '\n';
    cout << "~255 is " << ~(unsigned int)255 << '\n';
    return 0;
}

```

Remember that the integer 1 is stored in 2 bytes as 00000000 00000001, while 255 is 00000000 11111111. Here is the output:

```

1 & 1 is 1
1 | 1 is 1
1 ^ 1 is 0
255 << 2 is 1020
255 >> 2 is 63
~255 is 65280

```

Notice that the **&** operator gives 1 only if the corresponding bits are both 1. This makes it useful for “masking off” selected bits of a value by using a bit pattern that has zeros in the positions you wish to turn off. The **|** operator turns on a bit if either or both values have a 1 in that position. If you want to guarantee that a certain bit is on, “OR” that value with a value that has a 1 in that position.

The fourth statement left-shifts the value 255 twice (00000000 11111111 becomes 00000011 11111100). Since each place in a binary number is two times the preceding place, this is equivalent to multiplying $255 * 2 * 2$, or 1020. In the next statement, 255 is right-shifted twice, so 00000000 11111111 becomes 00000000 00111111, or 63. Finally, the last statement turns 00000000 11111111 into 11111111 00000000. This is equivalent to $65,535 - 255$, or 65,280.

A few points should be made about the properties of the *insertion* **<<** and *extraction* **>>** stream operators. The reasons for overloading the *shift-left operator* **<<** and the *shift-right operator* **>>** for use as stream operators are that they are among the least commonly used operators, and the precedence of the *shift-left* **<<** and *shift-right* **>>** operators is low enough to allow the use of arithmetic expressions as operands without parentheses. However, if you write expressions containing operators of lower precedence you must contain the expression within parentheses. Specifically, there are three operators which have a lower precedence than the shift

operators. These are the *AND* **&**, the exclusive or *XOR* **^**, and the inclusive or *OR* **|** bitwise operators. Also you must parenthesize an expression containing the *shift-left* **<<** or *shift-right* **>>** bitwise operators when intermingled with the overloaded *insertion* **<<** or *extraction* **>>** stream operators. When one of these operators is used within the **cout** or **cin** statements you must parenthesize them as follows:

```
cout << "1 & 1 is " << (1 & 1) << '\n';
```

If you should fail to parenthesize, as in the above example, the compiler will provide you with the error message `Illegal use of pointer`, in other words Turbo C++ attempts to interpret the ampersand **&** as the address-of operator.

Expressions

You have now seen a variety of statements that use expressions, so this is a good time to review how expressions work in general. An expression is any combination of variables, defined constants, or literal numbers which together with one or more operators yield a single value and possibly produce one or more side effects. Thus

```
purchase * TAX_RATE  
dollars / bushels  
count++  
STATUS & SWITCH_ON
```

are all examples of expressions. An expression can be assigned to a variable with the assignment operator **=**. It can be displayed by **cout** in the same way as a single variable. The preceding program, for example, used statements such as

```
cout<<"255 << 2 is " << (255 << 2) << "\n";
```

Where the expression `(255 << 2)` is an expression which is equivalent to 1020, the result of left-shifting the value 255 twice.

Evaluating an expression

Turbo C++ evaluates expressions by applying the operators involved in order of their precedence, starting first with any parts of the expression that are enclosed in parentheses. Table 3.4 lists all of the C++ operators in order of their precedence and associativity. *Associativity* is the direction in which the compiler evaluates

the operators and operands. For example, the various assignment operators (=, +=, *=, and so on) associate from right to left (assigning the expression on the right to the variable on the left), while the arithmetic operators (*, /, +, -, and %) associate from left to right.

Precedence is the order in which evaluations are done. For example, multiplication is done before addition, so that the statement

```
count = 5 + 3 * 4
```

results in 17, not 32.

The best way to become familiar with these rules is to use an expression with a **cout** statement so you can see the result, then check the table to see how that result was arrived at. (As an alternative, Chapter 6, "Debugging in the new IDE," shows how you can use the built-in Turbo C++ debugger to evaluate many kinds of expressions.)

The rules of precedence and associativity are as follows:

- Unary operators (operators with only one operand, such as ++) have a higher order of precedence than binary operators (such as /).
- Arithmetic operators come ahead of comparison operators.
- Greater than and less than come ahead of equals and not-equals.
- Comparison operators come ahead of bit manipulation operators (except for left and right shifts).
- Bit manipulation operators come ahead of logical operators.
- Logical AND "&&" comes ahead of logical OR "||".
- Everything except for the comma operator comes before assignment operators.

Table 3.5
Precedence and
associativity of operators

Operators, precedence, and associativity are explained in full in chapter 12, "Language structure".

Operators	Associativity
() [] -> :: .	Left to right
! ~ + - ++ -- & * (typecast) sizeof new delete	Right to left
.* ->*	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?: (conditional expression)	Right to left
= *= /= %= += -= &= ^= = <<= >>=	Right to left
,	Left to right

Assigning a value in an expression

Try this example program,
load and run `INTRO10.CPP`

You have been seeing examples of the assignment operator used in a complete statement, such as the initialization of a variable, `total = 0`. Assignment statements like `total = count = line = 0` remind us that every assignment is also an expression. What do you think this program will display?

```
#include <iostream.h>

int main()
{
    cout<<"1 & 1 is "<<(1 & 1)<<"\n";
    cout<<"1 | 1 is "<<(1 | 1)<<"\n";
    cout<<"1 ^ 1 is "<<(1 ^ 1)<<"\n";
    cout<<"255 << 2 is "<<(255 << 2)<<"\n";
    cout<<"255 >> 2 is "<<(255 >> 2)<<"\n";
    cout<<"~255 is "<<~(unsigned int)255<<"\n";
    return 0;
}
```

If you guessed 7, you're right. The value of an assignment statement as an expression is the value assigned.

You have also seen that a call to a function that returns a value has that value. For example, `total += tax(total)` is equivalent to

```
tax_amt = tax(total);
total = total + tax_amt;
```

The first form eliminates the extra variable at the cost of being a bit more cryptic.

Characters and strings

We've seen several types of variables which store numbers, such as integers and floating point numbers. Now it's time to look at characters and character strings. A character — whether an uppercase or lowercase letter of the alphabet, a numeral, a punctuation symbol, a carriage return, or *Ctrl-C*—is stored as a single byte (8 bits).

The values for characters are assigned by the ASCII (American Standards Committee for Information Interchange) code. The first 128 values are pretty much the same throughout the industry, but IBM PC-compatible machines use the second 128 values (128 to 255) for special graphics and line-drawing characters. In C++, you use the type **char** (which means the same as **signed char**) to access the values from 0 to 127, with room for negative values for special purposes (such as indicating an error or the end of a file). If you want to access the full character set of the PC, use the type **unsigned char**.

Input and output for single characters

Here is one way to get a character from the keyboard and store it in a variable (to try it out, load and run INTRO11.CPP):

```
// INTRO11.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    char one_char;
    cout << "\nEnter a character: ";
    cin >> one_char;
    cout << "The character you entered was " << one_char << '\n';
    cout << "Its ASCII value is " << (int) one_char << '\n';
    return 0;
}
```

A sample run looks like this:


```
Enter a character: A
The character you entered was A
Its ASCII value is 65
```

`one_char` is a variable of type **char**. The **cin** statement notes the *type* of the declared variable `char one_char`, and stores the entered value in `one_char`. The **cout** statement, `cout << "\n" << "The character you entered was " << one_char << "\n";`, displays the value of `one_char` as type *char*. The next **cout** statement prints the ASCII code for the entered character. It does this by using type casting, `... << (int) one_char;`. This converts the character value to its ASCII code. A character *is* an integer, but certain facilities in C++ (such as the **char** type specifier) display the value as a character rather than as an integer.

An alternate way to get a character from the keyboard is to use the *library functions* **getch()** or **getche()** which come supplied with Turbo C++. A *library function* performs many activities such as mathematical computations, graphics, file operations, displaying output to the screen, and getting information from the keyboard. To use a function contained in a library file you must include the header file which contains the function declarations you want to use. To use **getch()** or **getche()** you must include the header file `conio.h`.

```
#include <conio.h>
```

The previous program could be rewritten as follows:

This one is INTRO12.CPP; load and run it to see how it works.

```
// INTRO12.CPP--Example from Chapter 3, " An Introduction to C++"
#include <iostream.h>
#include <conio.h>

int main()
{
    int one_char;
    cout << "\nEnter a character: ";
    one_char = getch();
    cout << "\nThe character you entered was "
         << (char) one_char << '\n';
    cout << " Its ASCII value is "<< (int) one_char << '\n';

    return 0;
}
```

With this version of the program, the character you type won't be displayed onscreen. If you want the entered character to be visible, use **getche** instead.

Displaying a character

This example also shows one way to display a character line. By *typecasting* the declared variable `int one_char` as a *char* in the first `cout` statement we display the variable as a single character. Another way to display a single character is with the library function `putch`, which is defined in the header file `conio.h`. The function `putch('>')` displays the specified literal character `>`. If `one_char` is a variable of type `char` (or, in some cases, of type `int`), `putch(c)` displays the value of `one_char` as a character.

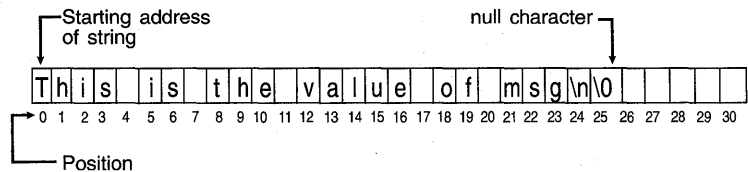
Displaying character strings

A string is a series of characters. You have already seen strings used in `cout` statements. For example,

```
cout << "Enter a character: ";
```

calls `cout` and tells it to display a string of characters beginning with an E and ending with a space. The double quotes tell Turbo C++ to treat the group of characters as a string. Each character is stored in a consecutive byte of memory, and `cout` receives the address of the first character. How does `cout` know when it has reached the end of the string? Whenever you define such a *literal string*, Turbo C++ invisibly tacks a *null character* on the end. This character has an ASCII value of 0, and is represented symbolically as `\0`. The next figure shows how a string is stored in memory.

Figure 3.2
How a string is stored in memory



```
char message[30];  
strcpy(message, "This is the value of msg\n");
```

Important!

`message` is actually a pointer to a location for the string. Therefore, the code

```
message = "This is the value of msg\n";
```

makes `message` point to where that string is. It does *not* copy the literal string into the storage area of `message`. This is why you

must use **strcpy** to copy the characters of the string to the variable.

For convenience, Turbo C++ also provides the library function **puts**. This function writes a string to the standard output, normally the screen. Thus `puts("Enter a character: ")` works almost the same as the **cout** statement just described, with one exception: **puts** automatically puts a new line (**\n**) at the end of the string, advancing the cursor to the next line. If your program doesn't need the formatting capabilities of **cout**, you can save considerable program space by using the simpler **puts** to display strings.

There is an important difference between double quotes and single quotes for specifying strings and characters.

- `"a"` is a *string* consisting of one character, *a*, and the invisible null character
- `'a'` is the single *character a*.

Since strings and characters are different data types, specifiers or functions that work with strings do not work with characters, and vice versa. To display `"a"`, use **puts**; for `'a'`, use **putch**.

Testing conditions and making choices

You have now learned many elements of the C++ language. So far, all programs have run straight through from beginning to end. Often, however, programs must make choices based on certain values. For example, consider this code fragment:

```
if (bill > credit_limit)
    puts("Consult with the manager");
```

If the amount of the bill is greater than the amount of the credit limit, the code displays the message about consulting with the manager.

Using relational operators

The `>` (greater than) operator in the preceding statement is a *relational operator*. It expresses a relationship between two values—in this case, whether *bill* is greater than *credit_limit*. Computers use a simple two-valued logic: If a relationship is true,

it has a value of 1; if it is false, it has a value of 0. The relational operators are listed in the next table.

Table 3.6
Relational operators

Operator	Meaning	Example
>	Greater than	5 > 4
>=	Greater than or equal to	5 >= x
<	Less than	4 < 5
<=	Less than or equal to	x <= 5
==	Equal to	5 == 5
!=	Not equal to	5 != 4

This example program shows you how relational operators can be used:

To try out this program, load and run *INTRO13.CPP*.

```
// INTRO13.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int first, second;
    cout << "\nInput two numbers ";
    cin >> first >> second;
    cout << "First > second has the value "
         << (first > second) << '\n';
    cout << "first < second has the value "
         << (first < second) << '\n';
    cout << "first == second has the value "
         << (first == second) << '\n';

    return 0;
}
```

Here's a sample run, using the values 3 and 5 (be sure to type a space between the 3 and the 5):

```
Input two numbers: 3 5
first > second has the value 0
first < second has the value 1
first == second has the value 0
```

Notice that a relational test is an expression, since it gives a value. Thus, it can be displayed by **cout**, and you can assign it to a variable with a statement like `hot = (temperature > 90)`. Be careful not to confuse **==** (the relational equals operator) with **=** (the assignment operator). Try editing the last statement in the example so that it reads

```
cout << "First == second has the value " << (first = second) << '\n';
```

The expression `first = second` evaluates to the value of `second`, or 5 in the sample run. The **if**, **for**, and other statements that test conditions consider any nonzero condition to be true. You can see that using `=` where you meant `==` will cause inappropriate program behavior (such as being stuck in an endless loop).

Using logical operators

Table 3.7
Logical operators

You can combine more than one condition in a test. To do so, use one of the three *logical operators* shown in the next table.

Operator	Meaning
<code>&&</code>	AND (both conditions must be true)
<code> </code>	OR (at least one condition must be true)
<code>!</code>	NOT (reverse the truth value of a condition)

For example, the conditional expression

```
(employee_type == temporary) && (wage > 6.00)
```

is true only if the employee is a temporary *and* his or her wage is over \$6.00 an hour. C++ is efficient at handling these operators: If the first condition (`employee_type == temporary`) is found to be false, the second condition isn't tested, since an AND expression is false if either condition is false.

The expression

```
(employee_type == temporary) || (employee_type == hourly)
```

is true if *either* or both of the two conditions is true. Thus if the first condition is found to be true, there's no need to test the second.

Branching with **if** and **if...else**

Now that you've surveyed relational and logical operators, it's time to put them to work. The simple **if** statement takes the form

```
if (conditional expression)  
    statement or group of statements;
```

The condition can be a single relational expression or a combination of expressions joined by logical operators. It must be enclosed in parentheses. The **if** statement acts according to the true or false value from the conditional expression. If the expression is true, the statement or group of statements that follow is executed. If you want a group of statements to be executed, enclose them in

braces. The following program uses two **if** statements to tell you whether the number you entered is odd or even:

To try out this code, load and run `INTRO14.CPP`.

```
// INTRO14.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int your_number;

    cout << "Enter a whole number: ";
    cin >> your_number;

    if (your_number % 2 == 0)
        cout << "\nYour number is even\n";

    if (your_number % 2 != 0)
        cout << "Your number is odd.\n";
    cout << "That's all!\n";

    return 0;
}
```

After prompting for and storing *your_number*, the program uses an **if** statement to test whether the number is even, using the modulus (%) operator. (Since all even numbers are evenly divisible by 2, any even number mod 2 gives a result of 0.) If the number is even, the first **cout** statement is executed. The second **if** statement tests whether *your_number* is odd, that is, if it has a nonzero remainder when it's divided by 2. If the number is odd, you see the following:

```
Your number is odd.
That's all!
```

The final message, "That's all!" isn't part of an **if** statement. It is executed regardless of whether *your_number* is even or odd.

Multiple choices with **if...else**

The previous example probably looked awkward to you. If a number is even, it *can't* be odd, so why make two separate tests? This example can be rewritten much more compactly by adding an **else** branch to the **if**. The **if ... else** statement has this form:

```
if (conditional expression)
    statement or group of statements;
else
    alternative statement or group of statements;
```

Applying this to the previous example, you get

To try out this program, load and run `INTRO15.CPP`.

```
// INTRO15.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int your_number;
    cout << "Enter a whole number: ";
    cin >> your_number;

    if (your_number % 2 == 0)
        cout << "Your number is even\n";
    else
        cout << "Your number is odd.\n";
    cout << "That's all!\n";
    return 0;
}
```

You can nest **if** and **else** clauses as deeply as you wish, although after a few dozen nested **if** statements the compiler is likely to run out of memory. Suppose you had to write a program that allocated computers to employees, subject to the following conditions.

- If your employees are programmers and have been working at least two years, give them an 80486 PC.
- If your employees are programmers and have been working *less than* two years, give them an 80386 PC.
- If your employees have been working at least two years, but aren't programmers, give them an 80286 PC.
- Finally, if your employees don't meet any of these conditions, give them an 8088 PC.

Try to map out how you could specify these conditions with **if** and **else** statements. Here's one way:

```
if (employee_type == PROGRAMMER) {
    if (years_worked >= 2)
        give_employee(PC486);
    else
        give_employee(PC386);
}

else if (years_worked >= 2)
    give_employee(PC286);
else
    give_employee(PC88);
```

Notice how we used indentation to show which conditions depend on other conditions. First the program determines whether the employee is a programmer. If so, it checks the number of years worked, and awards the appropriate computer. If the employee isn't a programmer, the outer **if...else** statement checks the number of years worked, and awards the machines assigned to nonprogrammers of varying seniority.

Multiple choice tests: **switch**

A long series of **if** and **else if** statements is tedious to write, confusing, and prone to error. Consider the next program, which has to decide how to graph a set of data based on the character the user has entered in response to a menu. Here's one way to do it:

To try out this code, load and run `INTRO16.CPP`.

```
// INTRO16.CPP--Example from Chapter 3, "An Introduction to C++"
#include <conio.h>
#include <iostream.h>
#include <ctype.h>

int main()
{
    char cmd;

    cout << "Chart desired: Pie Bar Scatter Line Three-D";
    cout << "\n Press first letter of the chart you want: ";
    cmd = toupper(getch());
    cout << '\n';

    if (cmd == 'P')
        cout << "Doing pie chart\n";
    else if (cmd == 'B')
        cout << "Doing bar chart\n";
    else if (cmd == 'S')
        cout << "Doing scatter chart\n";
    else if (cmd == 'L')
        cout << "Doing line chart\n";
    else if (cmd == 'T')
        cout << "Doing 3-D chart\n";
    else cout << "Invalid choice.\n";

    return 0;
}
```

The program displays a menu line, then gets a value for `cmd` via the **getch** function. Along the way, the value is passed to the **toupper** function. This ensures that you only need to deal with uppercase characters. The series of **if** and **else if** branches then test

for each valid value and execute the corresponding function. The last **else** serves as the *default* case, handling invalid values.

The **switch** statement makes these multipath branches easier to code. It uses the form

*The ... means that you can have as many **case** clauses as you want.*

```
switch(value)
{
    case value : statement or group of statements
    ...
    default : statement or group of statements
}
```

The value is tested against the value for the first **case**. If they are the same, the program code given after the colon for the first case is executed until the end of the **switch** statement or until the special statement **break** is reached. If they are different, the value for the next **case** is tested, and so on. If none of the values are the same as the **switch** value, the statement or group of statements following **default** is executed. The **default** is optional. If you don't supply one, and no condition is met, then no statements within the **switch** are executed.

The preceding program example can be rewritten using a **switch** as follows:

To try out this program, load and run INTRO17.CPP.

```
// INTRO17.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>
#include <conio.h>
#include <ctype.h>

int main()
{
    char cmd;

    cout << "Chart desired: Pie Bar Scatter Line Three-D";
    cout << "\nPress first letter of the chart you want: ";
    cmd = toupper(getch());
    cout << '\n';

    switch (cmd)
    {
        case 'P': cout << "Doing pie chart\n"; break;
        case 'B': cout << "Doing bar chart\n"; break;
        case 'S': cout << "Doing scatter chart\n"; break;
        case 'L': cout << "Doing line chart\n"; break;
        case 'T': cout << "Doing 3-D chart\n"; break;
        default : cout << "Invalid choice.\n";
    }
}
```

```
    return 0;
}
```

The **break** statement at the end of each case is very important. It causes execution to jump past the end of the **switch** statement. You usually want to include a **break** statement as the last statement for each **case**. For example, remove the **break** at the end of the statement for case 'L' and run the program again. If you select L, you'll see

```
Doing line chart
Doing 3-D chart
```

The statements for both the L and T cases are executed. In other words, if you leave out the **break**, execution continues until it finds a **break** or the end of the **switch** statement.

Sometimes this behavior can be useful. Suppose that you want the user of your program to be able to use either D (for delete) or E (for erase) to remove the current file. You could then code

```
switch (cmd)
{
    case 'I': insert_file(); break;
    case 'F': format_file(current_file); break;
    case 'D':
    case 'E': erase_file(current_file);
}
```

Since there is no **break** statement for case 'D', **erase_file** will be executed for this case as well as for case 'E'.

Repeating execution with loops

The most significant characteristic of the **if**, **else**, and **switch** statements is that they perform their test only once, and execute whatever statements are specified only once. But many computer tasks involve *repetition*; they involve instructions such as "use the same process on each item in this file until you get to the end of the file" or "use the same process on each item in this set of data." For this kind of task, you'll want to use a loop. *Loops* cause a statement or series of statements to be executed repeatedly, monitoring a specified condition in order to determine when to stop. C++ provides three kinds of loops: **while**, **do**, and **for**.

The **while** loop

The **while** loop executes one or more statements as long as a specified condition is true. The syntax is

```
while (condition)  
    statement or group of statements;
```

The following program lets you enter numbers from the keyboard. It keeps a running total. When you enter a 0, it gives you the total and average of the numbers entered.

To try out this example, load and run `INTRO18.CPP`.

```
// INTRO18.CPP--Example from Chapter 3, "An Introduction to C++"  
  
#include <iostream.h>  
  
int main()  
{  
    int number= 1;          // Number entered by user  
    int total = 0;         // Total of numbers entered so far  
    int count = 0;         // Count of numbers entered  
  
    cout << "\nEnter a number, 0 to quit:\n";  
    while (number != 0)  
    {  
        cin >> number;  
        if(number == 0)  
            cout << "Thank you. Ending routine.\n";  
        else  
            count++;  
            total += number;  
    }  
    cout << "Total is " << total << '\n';  
    cout << "Count is " << count << '\n';  
    cout << "Average is " << total / count << '\n';  
  
    return 0;  
}
```

On the first time through the loop the value of *number* is 1. It is tested by

```
while (number != 0)
```

Since it isn't a 0, it enters the body of the loop, where several things happen:

- A new number is obtained with **cin**.

- That number is tested. If it is equal to 0, a goodbye message is printed. Otherwise the count of numbers is incremented by one.
- The number just entered is added to *total*.
- The **while** statement is tested again to see if the number equals zero, and if not, the loop begins again.
- When the loop finally exits (after a 0 is entered), the values of *total*, *count*, and *total/count* (average) are printed out.

The **do while** loop

The **do while** loop is very similar to the **while** loop. It takes the form

```
do statement or group of statements
while (condition is true)
```

Important!

What's the difference between a **do while** and a **while** loop?

- The **while** loop performs the test *first* and executes the enclosed statements only if the result of the test is true.
- The **do while** loop executes the enclosed statements and *then* performs the test. This means that the enclosed statements are performed at least once, even if the test turns out to be false.

A good situation for using the **do while** loop is processing a menu. The earlier menu examples had the drawback that they only executed once, and then terminated the program. Here is the menu program rewritten to use a **do while** statement:

To try out this example, load and run *INTRO19.CPP*.

```
// INTRO19.CPP--Example from Chapter 3, "An Introduction to C++"
#include <conio.h>
#include <ctype.h>
#include <iostream.h>
int main()
{
    char cmd;

    do {
        cout << "Chart desired: Pie Bar Scatter Line Three-D
Exit";

        cout << "\nPress first letter of the chart you want: ";
        cmd = toupper(getch());
        cout << '\n';

        switch (cmd)
        {
```

```

        case 'P': cout << "Doing pie chart\n"; break;
        case 'B': cout << "Doing bar chart\n"; break;
        case 'S': cout << "Doing scatter chart\n"; break;
        case 'L': cout << "Doing line chart\n"; break;
        case 'T': cout << "Doing 3-D chart\n"; break;
        case 'E': break;
        default : cout << "Invalid choice. Try again\n";
    }
} while (cmd != 'E');
return 0;
}

```

What has changed? The active part of the program, including the statements that display the menu and get a character as well as the **switch** statement, have been enclosed in a **do while** loop. An additional menu case, *E*, allows the user to exit the program. This **case** simply has a **break** associated with it. If any other character (including an invalid character) had been typed, the **while** condition causes the menu to be redisplayed. But if *E* (or *e*) is typed, the condition `cmd != 'E'` is false, and control drops out of the bottom of the **do while** loop. The program then terminates.

The **for** loop

The **for** loop steps through a series of values, performing the specified actions once for each value. The form for this statement is

```

for (starting values; condition; changes)
{
    statement or group of statements;
}

```

The following **for** loop displays the visible characters from the PC's character set:

To try out this example, load and run INTRO20.CPP

```

// INTRO20.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>
int main()
{
    int ascii_val;
    for (ascii_val = 32; ascii_val < 256; ascii_val++)
    {
        cout << '\t' << (char)ascii_val;
        if (ascii_val % 9 == 0)
            cout << '\n';
    }
}

```

```

    }
    return 0;
}

```

The variable *ascii_val* is the *counter variable* for the **for** loop. The **initialization** (starting) value is `ascii_val = 32`. The condition `ascii_val < 256` is the *limit* for the loop. The *change* that is made in the counter variable is `ascii_val++` (in other words, it is incremented by one each time through the loop.)

The **cout** and **if** statements are enclosed by braces and make up the body of the loop. The **cout** statement displays the character corresponding to the current value of *ascii_val*, using a tab character (**\t**) for spacing. The **if** statement begins a new line whenever *ascii_val* is evenly divisible by 9 — in other words, it ensures that each row will have 9 characters.

There are many variations on the theme of **for** loops. The body of the loop can have only one statement, in which case the braces are optional but recommended for clarity. A **for** loop can have no body at all, with all of the work being done in the change part of the control statement. For example, this loop totals up the numbers from 1 through 10:

Try out this example, load and run `INTRO21.CPP`

```

// INTRO21.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int number, total;
    for (number = 1, total = 0; number < 11; total += number,
        number++);
    cout << "\nTotal of numbers from 1 to 10 is "<< total;
    return 0;
}

```

This example also shows that the starting and change parts of the loop specification can have multiple expressions, separated by commas. The loop initializes two variables, *number* and *total*. Each time the loop runs, it adds *number* to *total* and then increments *number*.

Some programmers put the semicolon on a separate line, so it will stand out.

The semicolon immediately following the closing parenthesis indicates that the body of the loop is empty. If it were omitted, the loop would execute the **cout** statement repeatedly, treating it as the body.

A **for** loop could be written instead as a **while** loop. The previous example could be rendered as

*To try this out, load and run
INTRO22.CPP.*

```
// INTRO22.CPP--Example from Chapter 3, "An Introduction to C++"  
#include <iostream.h>  
  
int main()  
{  
    int number = 1, total = 0;  
    while (number < 11)  
    {  
        total += number;  
        number++;  
    }  
    cout << "\nTotal of numbers from 1 to 10 is " << total;  
  
    return 0;  
}
```

The **while** loop updates the counter variable within the body of the loop. The **for** loop performs both of these operations within the loop specification itself. The **for** loop is more compact, but it can be harder to read if you place too many expressions between the parentheses.

break and continue

Sometimes it's necessary to abandon the statements in a loop even before you test the condition again. The **break** statement has two different uses: to break out of a **switch** statement after statements provided for a particular **case** have been executed, and to exit a **while**, **do while**, or **for** loop immediately, without performing the rest of the statements enclosed in the loop. For example, suppose you didn't want to launch a space shuttle if any warning lights were on:

*To try out this program, load
and run INTRO23.CPP.*

```
// INTRO23.CPP--Example from Chapter 3, "An Introduction to C++"  
#include <iostream.h>  
  
int warning = -1;  
  
int get_status(void)  
{  
    return warning;  
}  
  
int main()  
{  
    int count = 10;
```

```

while (count-- > 1)
{
    if (get_status() == warning)
        break;
    cout << '\n' << count;
}
if (count == 0)
    cout << "Shuttle launched\n";
else
{
    cout << "\nWarning received\n";
    cout << "Count down held at t - " << count;
}
return 0;
}

```

A **while** loop runs the countdown, each time checking the function **get_status** to see if it returns a value of -1 (which you have defined as the variable `int warning = -1;`). This function would presumably be reading the warning system in real time. If **get_status** returns -1 at any time during the countdown, **break** stops the countdown. Following the **while** statement, the **if** statement checks to see if *count* reached zero. If it did, then no warning occurred. If *count* is greater than 0, however, the countdown must have been interrupted, and the shuttle is not launched.

The **continue** statement, like **break**, causes all remaining statements in the loop to be skipped. But while **break** completely exits the loop, **continue** simply skips to the loop's test condition. **INTRO24.CPP** displays the even numbers up through 10:

*To try out this program, load and run **INTRO24.CPP**.*

```

// INTRO24.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    int num = 0;
    while (num++ <= 10)
    {
        if (num % 2 != 0)
            continue;
        cout << '\n' << num;
    }

    return 0;
}

```


If the number is odd, the **continue** statement causes the **cout** statement to be skipped.

continue isn't used very often. You have to think the problem through to see when a **continue** or a **break** is appropriate. The use of **continue** inside the **while** loop in the space shuttle program would not be a good idea, since not only would the countdown not be displayed (the **cout** statement in the loop would be skipped), but also *count* would continue down to 0, since the decrement is part of the condition (*count-- > 1*). The shuttle would be launched even if a warning had been received.

Nested loops

One of the statements in the body of a loop can be another loop — this is called *nesting*. In this example, a **for** loop accepts strings that you enter, and the nested **for** loop prints hyphens under the string in order to underline it. We also introduce the function **getline()** which is an inherited member function of **cin**, declared in *iostream.h*. **getline()** reads characters until encountering the '\n' character and places the resulting string in the buffer specified by the argument. In the program *INTRO25.CPP* we have used the **getline()** function twice in the initial **for** statement, the first usage initializes the loop with the first string you enter, the second **getline()** is accessed each time through the loop until you type the string "end".

To try out this code, load and run *INTRO25.CPP*.

```
//INTRO25.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>
#include <string.h>
#include <conio.h>

int main()
{
    int i;
    char text[80];

    cout << "Type \"end\" to quit\n";

    for(cin.getline(text,80);strcmp(text,"end") !=0;
        cin.getline(text,80))
    {
        for(i = 1; i <= strlen(text); i++)
            cout << "-";
        cout << '\n';
    }
}
```

```
    return 0;
}
```

The first **for** loop gets the input string with the function `cin.getline(text, 80)`, initializing the **for** loop, and storing the string in the character array `text`. Then, it compares this string to the string "end" with the **strcmp** function. If the comparison yields a 0, the strings are identical, and the loop exits. If the comparison yields a value other than 0, the nested **for** loop is executed. The nested **for** loop then prints a number of hyphens equal to the length of the string that was originally entered, using the **strlen** function to find out how long it was. The **cout** statement, which ends the body of the initial **for** loop, positions the cursor for entry of the next line.

Choosing appropriate loops

You have now seen three different ways to code loops (**while**, **do while**, and **for**). Use whichever form of statement expresses the idea of the program most clearly, keeping in mind the following guidelines:

- If you don't want the body of the loop to be executed at all if the condition is false, use a **while** loop.
- If you want the body of the loop to always be executed at least once, use a **do while** loop.
- If the number of times the loop is to be executed is determined by the value of a variable or constant, it is usually best to use a **for** loop.
- If the loop is to be executed as long as some externally determined condition is true (for example, as long as there is data left in the file), use a **while** loop.

Program design with functions and macros

Now that you know how to control the execution of a program, you can do some useful work in C++. We encourage you to modify and elaborate upon the example programs. Small, simple programs such as the ones we've covered so far don't need a lot of structure. However, as your programs grow larger and more complex, you need to break them down into smaller, more manageable logical pieces, or functions.

Defining your own functions

The programs you have already seen perform divisions of labor. When you call **getline**, **strlen**, or **strcmp**, you don't have to worry about how the innards of these functions work. These and about 400 other functions are already defined and compiled for you in the Turbo C++ library. To use them, you need only include the appropriate header file in your program and check the online help to make sure you understand how to call the function, and what value (if any) it returns.

But you'll need to write your own functions. To do so, you need to break your code into discrete sections (functions) that each perform a single, understandable task for your program. Once you have declared and defined your own functions, you can call them throughout your program in the same way that you call Turbo C++ library functions.

The function prototype

Function prototypes are a key feature of the new ANSI standard for the C language, and since the C language is a subset of C++, function prototyping is a key feature of C++. A function *prototype* is a declaration that takes this form:

```
return_type function_name(parameter_type [parameter_name] ...);
```

Here are some examples:

```
float tax(float purchase);
char get_employee_type(int employee_num);
char get_choice(void);
int getch(void);
void show_menu(void);
```

By looking at the prototype, you can tell exactly what type of information the function expects, and what type it returns.

Let's look at **tax**. When you call it, you need to give it a floating-point number. The result will also be floating point. The prototype informs Turbo C++ about all this.

Some of the other example prototypes shown earlier use the keyword **void**. **void** means empty, or none. When the word **void** appears in the parentheses in place of the parameter list, this indicates that the function has no parameters. Thus if you tried to use the statement `show_menu(10)`, you would be informed that **show_menu** doesn't take any parameters. When **void** appears as

the function's return type, it means that the function does not return a value—so you shouldn't try to assign the function call's result to a variable.

Function declarations under Kernighan and Ritchie

We recommend that you use prototypes in your code.

In the old Kernighan and Ritchie style, the return type of a function was given only if it wasn't `int`. Similarly, function parameters were declared within the body of the function definition, rather than in the parameter list. `get_employee_type` would be declared in this old style as

```
char get_employee_type();
```

And the function's actual code (its definition) would look like this

```
char get_employee_type(employee_num)
int employee_num;
{
    /* body of function code */
}
```

Old-style functions compile correctly under Turbo C++. However, they have less information about the function's parameters, so errors involving the wrong type of parameters in function calls won't be caught automatically. This is one of the many good reasons why this older style is being discarded. The Kernighan and Ritchie style of function declaration predates ANSI C and C++, and is out of date. The information given here is to aid you in recognizing the obsolete function declaration style in older C source code.

The function definition The function definition contains the actual code for the function. The preferred form for the function header starts out exactly the same as the function declaration, except that it doesn't end in a semicolon. It is followed by local variable declarations and the code to be executed, enclosed in braces.

```
float tax(float purchase)
{
    float tax_rate = 0.065;
    return(purchase * tax_rate);
}
```

Processing within the function

The function sees its parameters as though they had been declared as variables of the indicated types. The **tax** function thus has access to two values: its **float** parameter, *purchase*, and its own **float** variable, *tax_rate*. If you make the function call `tax(amount)`, it is a *copy* of the value of *amount* that the function **tax** receives through its parameter *purchase*. The function refers to this value under the name *purchase* and can change the value of *purchase*. This doesn't change the value of the original variable *amount*, however. We'll show you later how a function can use pointers to change the values of variables used to call it.

The function return value

A function doesn't have to return a value—in that case, you should declare its return type to be **void**. To return a value to the caller, a function uses the **return** statement (as in the function **tax**, where the value returned is *purchase * tax_rate*). The next figure summarizes how information flows to the **tax** function and then flows back again.

Figure 3.3
Information flow to and from
the tax function

The value of amount is assigned to corresponding parameter purchase

```
/*function declaration*/  
float tax(float purchase)  
...  
/*calling statement*/  
tax_amt = tax(amount);  
...
```

The value is now available within function tax under the name purchase

```
return(purchase * tax_rate);
```

The return statement sends the calculated value back to the calling statement, where it replaces the function call tax(amount)

```
tax_amt = tax(amount);
```

In calling statement, returned value is assigned to the variable tax_amt

```
tax_amt = (value returned)
```

Using the return value

The returned value of a function can be treated like any other value in an expression. It can be combined with other variables and arithmetic operators in an expression; it can be part of the

condition for an **if** statement or loop; it can be assigned to a variable, and so on. Here are some examples of the use of function return values that you have already seen:

```
tax_amt = tax(purchase);  
if (get_status() == warning)  
for (cin.getline(text); strcmp(text,"end") != 0)
```

In the first example, the value returned by the **tax** function is assigned to the variable *tax_amt*. In the second example, the value returned by the call to **get_status** is compared with the value of the variable *warning*; the result determines the truth value of the **if** statement. In the last example, the value returned by **cin.getline** is passed to the **strcmp** function (along with the string "end"), and the result of the call to **strcmp** in turn is compared with zero. That result becomes the value of the **for** statement test.

Multifunction programs

The following program draws a graphic representation of part of the solar system. It illustrates the use of several user-defined functions as well some features of Turbo C++'s graphics library. (For more on how to use Turbo C++ graphics, see Chapter 20, "Video functions.")

After you have loaded the PLANETS.CPP file, go to the MENU BAR and select OPTIONS | LINKER, and then select LIBRARIES. You will then be presented with the libraries dialog box, select GRAPHICS LIBRARY and toggle the checkbox to on with the spacebar or by clicking the left mouse button. This will ensure that GRAPHICS.LIB will be linked in when you compile and run PLANETS.CPP. Be sure to set your path argument in calls to **initgraph** to the directory where your .BGI files are. When setting the path use the \\ symbol instead of the usual \ so as to comply with C++ escape sequences, for example "c:\\tc\\bgi", or the relative path ".\\bgi".

As written, the program requires EGA or VGA graphics hardware, but because it scales itself to the capabilities of the adapter, you could run a cruder version in CGA if you change the color constants used.

The program listing has function prototypes, global declarations, and then the definitions of the various other functions. This program is included on your disk as PLANETS.CPP.

After reading through the discussion of this program, feel free to modify it. Study the graphics library. Try different colors and fill styles. Experiment with various scaling values for distances and radii.

```
// PLANETS.CPP--Example from chapter 3
#include <stdio.h>
#include <graphics.h> // For graphics library functions
#include <stdlib.h> // For exit()
#include <iostream.h>
#include <conio.h>

int set_graph(void); // Initialize graphics
void calc_coords(void); // Scale distances onscreen
void draw_planets(void); // Draw and fill planet circles

// Draw one planet circle
void draw_planet(float x_pos, float radius, int color, int fill_style);
void get_key(void); // Display text on graphics screen, // Wait for key

// Global variables -- set by calc_coords()
int max_x, max_y; // Maximum x- and y-coordinates
int y_org; // Y-coordinate for all drawings
int au1; // One astronomical unit in pixels // (inner planets)

int au2; // One astronomical unit in pixels // (outer planets)
int erad; // One earth radius in pixels

int main()
{
    // Exit if not EGA or VGA
    // Find out if they have what it takes
    if (set_graph() != 1) {
        cout << "This program requires EGA or VGA graphics\n";
        exit(0);
    }
    calc_coords(); // Scale to graphics resolution in use
    draw_planets(); // Sun through Uranus
    get_key(); // Display message and wait for key press
    closegraph(); // Close graphics system

    return 0;
}

int set_graph(void)
{
    int graphdriver = DETECT, graphmode, error_code;

    //Initialize graphics system; must be EGA or VGA
    initgraph(&graphdriver, &graphmode, ".\\bgi");
    error_code = graphresult();
    if (error_code != grOk)
```

```

        return(-1);           // No graphics hardware found
    if ((graphdriver != EGA) && (graphdriver != VGA))
    {
        closegraph();
        return 0;
    }
    return(1);               // Graphics OK, so return "true"
}

void calc_coords(void)
{
    // Set global variables for drawing
    max_x = getmaxx();       // Returns maximum x-coordinate
    max_y = getmaxy();       // Returns maximum y-coordinate
    y_org = max_y / 2;      // Set Y coord for all objects
    erad = max_x / 200;     // One earth radius in pixels
    au1 = erad * 20;        // Scale for inner planets
    au2 = erad * 10;        // scale for outer planets
}

void draw_planets()
{
    // Each call specifies x-coordinate in au, radius, and color
    // arc of Sun
    draw_planet(-90, 100, EGA_YELLOW, EMPTY_FILL);
    // Mercury
    draw_planet(0.4 * au1, 0.4 * erad, EGA_BROWN, LTBKSLASH_FILL);
    // Venus
    draw_planet(0.7 * au1, 1.0 * erad, EGA_WHITE, SOLID_FILL);
    // Earth
    draw_planet(1.0 * au1, 1.0 * erad, EGA_LIGHTBLUE, SOLID_FILL);
    // Mars
    draw_planet(1.5 * au1, 0.4 * erad, EGA_LIGHTRED, CLOSE_DOT_FILL);
    // Jupiter
    draw_planet(5.2 * au2, 11.2 * erad, EGA_WHITE, LINE_FILL);
    // Saturn
    draw_planet(9.5 * au2, 9.4 * erad, EGA_LIGHTGREEN, LINE_FILL);
    // Uranus
    draw_planet(19.2 * au2, 4.2 * erad, EGA_GREEN, LINE_FILL);
}

void draw_planet(float x_pos, float radius, int color, int
fill_style)
{
    setcolor (color);        // This becomes drawing color
    circle(x_pos, y_org, radius); // Draw the circle
    setfillstyle(fill_style, color); // Set pattern to fill interior
    floodfill(x_pos, y_org, color); // Fill the circle
}

```



```

void get_key(void)
{
    outtextxy(50, max_y - 20, "Press any key to exit");
    getch();
}

```

Function prototypes and global declarations

This program calls five programmer-defined functions. Their declarations appear right after the **#include** statements. These declarations could appear elsewhere, but then there wouldn't be any type checking until the prototypes are reached. It is easiest to have them right at the start. The prototypes of **void** and non-**int** functions *must* appear before the first call to those functions.

The global variables hold information needed for drawing the planets. Their values are calculated by the **calc_coords** function, and these values are used by **draw_planets**. Making these variables global (rather than declaring them inside a function) makes them accessible to both functions that need them. Later, we'll show other ways to share variables between two functions. (Object-oriented programming, which we'll introduce in the next chapter, generally shuns the use of global variables).

Setting up the graphics display

main's first call is to **set_graph**, which "packages" a number of operations involving the Turbo C++ graphics library. **set_graph** uses the library function **initgraph** with the DETECT mode to automatically determine what kind of graphics hardware is present. Notice the multiple **return** statements—a function can have as many **return** statements as needed. The first **if** determines whether there was an error initializing the graphics system (the code returned isn't equal to *grOk* ("graphics OK"))—the function exits and returns an error code in that case. The second test returns an error code if *neither* EGA nor VGA capability is present. The second test is made only if the first was successful. The use of multiple **return** statements avoids the need for **else** statements.

The identifiers DETECT and *grOk* appear not to have been declared anywhere. In fact, these are defined constants that are part of the graphics.h header file. In addition to function prototypes, header files frequently make useful definitions available to your programs. We recommend that you browse through graphics.h and other header files that you frequently use in your programs, so that you become familiar with their contents. All of the colors and pattern fill styles you'll encounter later are also defined in graphics.h. Each identifier has an integer

value associated with it, but the use of symbolic names makes it much easier to see what is going on. (BLACK is much more meaningful than 0.)

main checks the value returned by **set_graph**. If anything other than 1 was returned, the program displays a message and exits.

Calculating the graphics coordinates

If everything checks out, **calc_coords** is called next. Many beginning programmers are used to thinking of the x - y dimensions of the graphics screen as being fixed by those provided by the graphics adapter they use—for example, 640x350 for EGA. The temptation is to hard-code these exact dimensions into your program. But this leads to trouble when the program is run on a machine that has a different graphics resolution and set of available colors. As discussed further in Chapter 20, “Video functions”, the graphics library helps you write programs that run on a wide range of graphics adapters.

To help make this possible, the library functions **getmaxx** and **getmaxy** functions return the highest x - and y -coordinates, respectively, for the graphics mode currently set. (This in turn was set by **initgraph**.) The remaining statements

- set the center for the planetary circles by taking half of the maximum y value.
- scale the Earth’s planetary radius (which is the unit used to express the radii of the other planets) to 1/200th of the width of the screen (the maximum x value).
- set two distance measurements for placing the centers of the circles on the x -axis. Because distances in the solar system increase rapidly once past Mars, different scales are used for the inner and outer planets. As a result, the drawing won’t be accurate in distances, though it will accurately show the relative sizes of the planets. (Due to the limited size of the screen, you can’t have both.) For the same reason, Neptune and Pluto had to be omitted.

Drawing the planets

The function **draw_planets** gathers together a series of calls to the function **draw_planet**, which does the actual work. **draw_planet** takes four parameters: x_pos , $radius$, $color$, and $fill_style$.

- x_pos is the x -coordinate for the center of the planetary circle. It is obtained by multiplying the distance unit ($au1$ or $au2$) by the mean distance of the planet’s orbit from the sun, expressed in

astronomical units. (An astronomical unit is the distance of the Earth from the Sun, approximately 93 million miles.)

- *radius* is the radius for the planetary circle. This is obtained by multiplying the actual planet's radius in terms of Earth's radius (about 4,000 miles) by *erad*, the number of pixels per Earth radius.
- *color* is a constant from `graphics.h` that gives the color to be used for drawing the circle (and for filling it in) from the default EGA palette (which also works for VGA).
- *fill_style* is a constant from `graphics.h` that gives the style to be used for filling in the circle.

draw_planet takes these parameters and calls Turbo C++ graphics library routines to draw the circle and fill it in. Notice that the y-coordinate needed by **circle** and **floodfill** didn't have to be supplied as a parameter. Since it is fixed, the quantity *y_org* previously calculated by **calc_coords** is used.

Finally, **get_key** uses the **outtext** library function to display a message, then calls **getch**, which waits for a key to be pressed to exit the program.

Header files, functions, and libraries

In a small program, you will probably declare and define your functions in the same file, together with your **main** function that ties everything together. Such a structure is shown in the next figure.

Figure 3.4
Simple program structure (all
in one)

*Describe
your
functions*

*Call
functions*

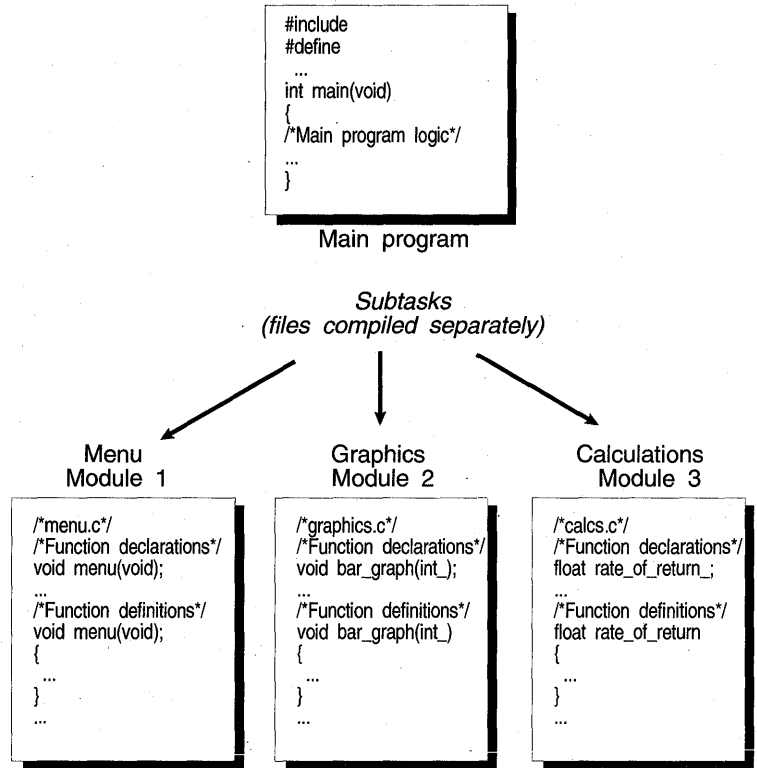
*Define tasks
performed
by your
functions*

```
#include
#define
/*Function declarations*/
void draw_menu(void);
...
...
/*Global data*/
...
main ()
{
    /*main program logic,*/
    /*including function calls*/
}
/*Function definitions (code)*/
void draw_menu(void)
{
    /*code for draw_menu*/
}
...
}
/*Definitions for other functions*/
...
```

*Header files usually contain
function declarations so they
can be included in many
different source or module
files.*

As programs get larger, however, it becomes desirable to group related function definitions in a separate file. For example, the functions dealing with the user interface might go into one file, the functions dealing with data processing in another, and the functions dealing with presentation graphics in yet a third file. Turbo C++ can compile all three files together to create the final executable program. This kind of structure is shown in the next figure.

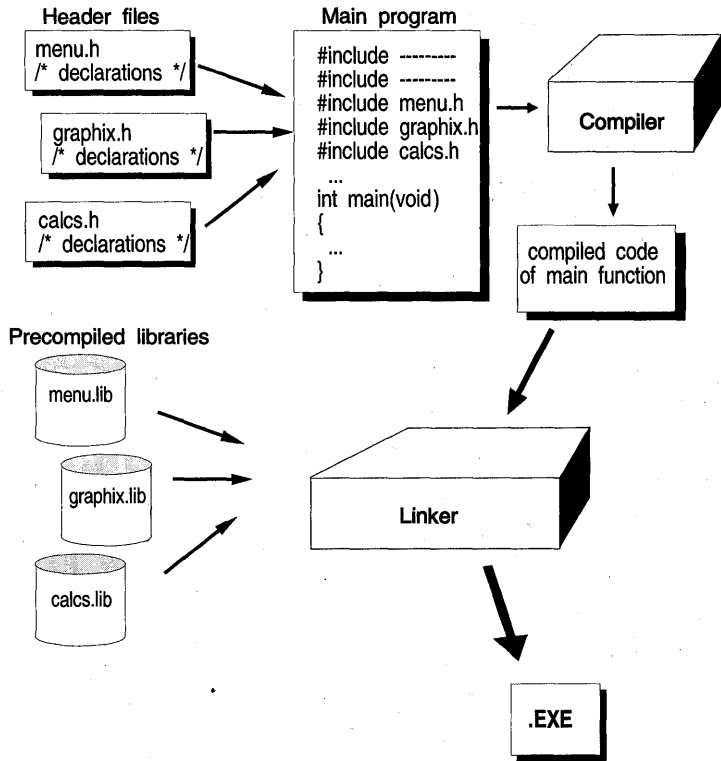
Figure 3.5
Program built from several files



You could substitute an `#include "header.h"` command for the function declarations.

Once parts of your program are stable, you can compile groups of functions into libraries. The declarations for the functions in each library can be put into a header file like those you use to access Turbo C++'s own libraries. Your main program includes the header files, thus inserting the function declarations into your program text. After compilation, the linker links the libraries into your program's object code. This process is shown in the next figure.

Figure 3.6
Program using custom
libraries



Scope and duration of variables

As programs get more complicated, the question of access to variables used in other parts of the program arises. Every variable has two characteristics: scope and duration. *Scope* (sometimes called *visibility*) defines what parts of a program can access the variable. *Duration* specifies how long the variable remains accessible.

Scope

Scope is determined by *where* you declare the variable. A variable defined within a function definition is by default *local*—it can only be accessed by code within the same function. For example, in this program,

Refer to chapter 5 for a more detailed discussion of scope with classes and struct.

You can load and run this program: *INTRO26.CPP*.

```

// INTRO26.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

void showval(void);
  
```

```

int main()
{
    int mainvar = 100;
    showval();
    cout << funcvar << '\n';

    return 0;
}

void showval(void)
{
    int funcvar = 10;
    cout << funcvar << '\n';
    cout << mainvar << '\n';
}

```

the function **showval** first uses a **cout** statement to display the value of *funcvar*. This is fine, since *funcvar* is declared and defined within the **showval** function. The next statement, which attempts to display the value of *mainvar*, causes the compiler to complain (by way of an error message) that *mainvar* is *undefined*. This is because *mainvar* is defined *within* another function, namely **main**. It cannot be accessed from within **showval**.

Even if you fix this, upon return from the call to **showval**, the **main** function tries to display the value of the variable *funcvar*. This, too, will cause an error because *funcvar* is defined within **showval**.

To make a variable visible from within *any* function in the current source file, define it *outside* any function definition. It will be visible after the position in the source file where it is declared, so the usual place to define such *global* variables is before the start of the definition of **main**. If the preceding example starts out like this,

```

void showval(void);
int mainvar, funcvar;

```

and if you remove the **int** declaration from

```

int mainvar = 100 and
int funcvar = 10,

```

there will be no complaint. Remember, however, that a variable that is accessible from anywhere is also changeable from anywhere, which can lead to bugs that are hard to track down. Changes in value caused this way are sometimes called *side effects*.

By default, global variables are accessible from any file. (Although, without **extern** references in other files, a global variable has file scope.) If you have a program that uses more than one source file, and you need to make a variable visible in a different source file, declare it in the current file by adding the keyword **extern** ("external"). Thus, if the file main.c defines `int xscale`, you can "see" this variable from within another file (such as stars.c) by declaring

```
extern int xscale;
```

there. The variable is assigned its memory address when it is originally defined. The **extern** declarations merely inform the compiler that an external variable will be referenced.

Duration It would be inefficient to reserve memory permanently for all of the variables in a large program. After all, a particular function may only be called once. By default, variables declared within a function definition are **auto** (automatic) variables. Their memory is allocated when their function begins to execute. When the function returns to its caller, the memory is freed up for use by other variables.

Occasionally you may wish to override this default behavior and have a variable stored permanently, even when the function it is declared in isn't running. The keyword **static** accomplishes this. For example, a function could count how many times it was called:

To try out this code, load and run INTRO27.CPP.

```
// INTRO27.CPP--Example from Chapter 3, "An Introduction to C++"  
#include <iostream.h>  
#include <conio.h>  
  
void tally(void);  
  
int main()  
{  
    cout << "Enter q to quit.\n";  
    while ( getch() != 'q')  
        tally();  
    return 0;  
}  
  
void tally(void)  
{  
    static int called = 0;  
    called++;  
}
```



```

    cout << "Function tally called " << called << " times\n";
}

```

Each time the **while** loop in **main** receives a character other than *q* from **getch**, it calls **tally**. **tally** increments the static variable *called* on each call. C++ initializes static numeric variables to 0 when they are declared.

Another declaration keyword that is occasionally used is **register**. When it is added to a variable declaration, **register** asks the compiler to generate code that uses one of the microprocessor's fast internal registers to hold the value, rather than using the slower random access memory. **register** can only profitably be used with data types small enough to fit in a register, such as **char** or **int**.

Since modern compilers such as Turbo C++ optimize so that they take advantage of machine resources efficiently, only experienced programmers know when to take advantage of this feature. Using it inappropriately can slow down your program. It is usually best to let the compiler figure out whether to use a machine register for a variable. Also, using the keyword **register** doesn't guarantee that the variable will be saved in a register. It is only a suggestion to the compiler that it attempt to do so.

Using constant values

A *constant* is a value that is fixed — it doesn't change during the execution of your program. There are two ways to define constants: the **const** keyword and the **#define** directive.

1. One way to create constants is to use the keyword **const** in a declaration. For example, if you declare

```
const float cm_per_inch = 2.54
```

you are telling the compiler that this value will never change. If you attempt to assign a new value to it anywhere in your program (including by incrementing or decrementing it with **++** or **--**, you will receive an error message. Using this keyword therefore helps Turbo C++ to catch programming slips.

2. Another way to include constant values in a program is by using the **#define** directive. Such as

```
#define RATE 0.065
```

This is not a declaration, but an instruction to the preprocessor, a part of Turbo C++ that will make requested changes to your source code before it is compiled. Here, the change is

equivalent to using a word processor to find all instances of RATE and replace them with the characters 0.065.

There are times when either **const** or **#define** is more appropriate. In both cases, you can change the value by simply changing the definition and recompiling. **const** gives the important advantage that Turbo C++ knows what data type the value should have (for example, **const int**). On the other hand, a **#define** can appear in multiple modules without problems, but a **const** can't. The potential for problems with the use of the **#define** directive result from the absence of any type declaration, therefore the compiler cannot perform type checking to ensure data type compatibility.

The advantage of using constants is that when the value changes, you need change only this one statement. You don't have to search for every instance of a particular number—a process which, besides being tedious, is prone to error.

Building data structures

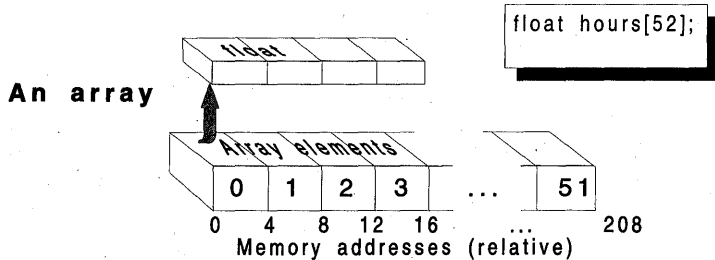
Data tends to come in bunches rather than single pieces. For example, you may need to keep track of the number of hours an employee has worked each week during the current year. Here you have a set of related data items of the same type (total hours, probably a **float** to allow for fractional hours). C++ allows you to declare an *array* to store such a set of homogeneous data. But your business also has to keep track of a variety of information about each employee, such as name, years worked, salary, department, and so on. These items are certainly related (they all refer to an employee), but they are of different types. Names are character strings (arrays of characters), years worked can be an **int** or a **float** depending on whether fractions are allowed, the salary is probably a **double** (to allow for well-paid employees), and the department can be a numeric code or a character string. The C++ structure type lets you handle complex data structures almost as easily as single variables.

*Structures are declared using the keyword **struct**.*

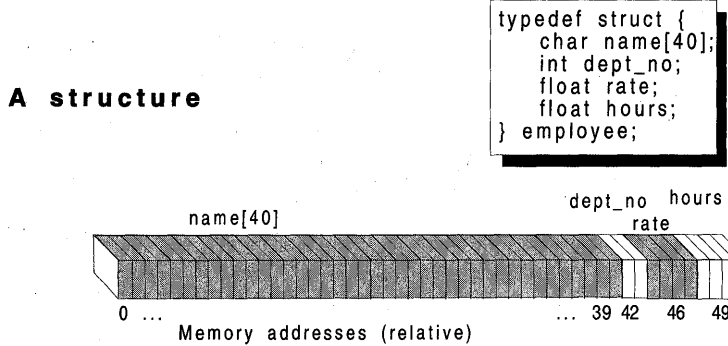
The **struct** type is used in ANSI C, but it does not have the added features provided for it in C++. The key difference is the treatment of a **struct** in C++ as being somewhat similar to a *class* (see Chapter five for more on structures and classes). A **struct**, by default, allows access to all of its members. In C++ a structure can include *member functions* allowing for the manipulation of the data

members. The examples and discussion in this section will be limited to the use of the **struct** as way to collect and organize your data efficiently.

Figure 3.7
Two ways to deal with sets of data



Note that the different members of a structure may not always be contiguous in memory.



In addition to organizing your data, you need a way that you can easily find the particular item you want to work with. Since all this data is actually stored in blocks of memory addresses (something like house addresses on a street), you can point to the data you want by using addresses. In this section, you'll learn how to use pointers to access data structures.

Declaring and initializing an array

An array is a chunk of memory that is used to hold a group of data items of the same type. For example, an array of **int** will hold the specified number of integers in consecutive memory locations. You specify an array by giving the type of data to be stored, the array name, and the number of items to be stored; put brackets around the number of items to be stored. For example, here is how you might declare an array that will hold the total hours worked per week for an employee for one year:

```
float hours[52];
```

This can be read as "hours, an array of 52 **float** values."

Arrays start at index 0 and end at a position one less than their assigned size.

A particular item, called an *element*, of the array can be referred to by giving the array name followed by the position of the item, in brackets. The first item is stored at the address pointed to by the array name itself. This can also be referred to as position 0. Thus the total for the first week in the *hours* array can be referred to as *hours[0]*. The total for the tenth week would be *hours[9]*, and the total for the 52nd week would be *hours[51]*.

The following program initializes the array *hours* to 0, assigns values for the first four positions, and then displays the value, showing how they are accessed:

To try this one out, load and run INTRO28.CPP.

```
// INTRO28.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    float hours[52];
    int week;

    // Initialize the array
    for (week = 0; week < 52; week++)
        hours[week] = 0;

    // Store four values in array
    hours[0] = 32.5;
    hours[1] = 44.0;
    hours[2] = 40.5;
    hours[3] = 38.0;

    // Retrieve values and show their addresses
    cout << "Element " << "\tValue " << "\tAddress\n";
    for (week = 0; week < 4; week++)
        cout << "hours[" << week << "]" << '\t'
            << hours[week] << '\t' << &hours[week] << '\n';

    return 0;
}
```

The output of the program will be:

The addresses will vary from one time to the next, and from one machine to another.

Element	Value	Address
hours[0]	32.5	0x8f43ff26
hours[1]	44	0x8f43ff2a
hours[2]	40.5	0x8f43ff2e
hours[3]	38	0x8f43ff32

Notice that the elements are stored in consecutive addresses, 4 bytes apart (which is, after all, the size of a single **float**). The address operator **&** retrieves the address of the element referenced. The **cout** statement `...<< &hours[week]...` displays the address as a hexadecimal number. The **for** statement is quite convenient for stepping through the elements of an array.

You can also explicitly initialize an array at the time it is declared. To do so, place the values you want to assign between braces, separating them with commas. For example,

```
int quarters[4] = {3, 10, 7, 14};
```

might hold the amount of points scored by a team in each quarter of a football game. (Of course data like this would probably come from being entered at the keyboard or being read from a file. But you can use explicit assignments to an array to provide data for testing your program.)

When you assign character constants, put each character in single quotes:

```
char grades[5] = { 'A', 'B', 'C', 'D', 'F' };
```

If you specify fewer values than the size of the array will accommodate, and the array is global or static, Turbo C++ sets the remaining elements to 0 (if the array is of the numeric type) or to null characters (in the case of an array of characters). If you specify *more* values than specified in the size of the array, you will receive the error message, "Too many initializers."

Arrays can be more complex than this (for example, you can omit the array size, or initialize a character array with a string). However, such topics are beyond the scope of this chapter.

Arrays with multiple dimensions

Sometimes it is useful to have a set of sets of values—for example, if you want to store the total hours worked by 12 employees during 52 weeks, you can declare

```
float hours[12][52];
```

Read this as "an array containing 12 arrays of 52 values each, of type **float**." You can think of this layout as being like a spreadsheet with 12 rows and 52 columns.

This next game generates a fictitious baseball score, by using an array called `scoreboard[2][9]` representing two teams and their respective scores for nine innings.

Load and run `GAME.CPP`

```
// GAME.CPP--Example from Chapter 3 "An Introduction to C++"

#include <stdlib.h>
#include <iostream.h>
#include <conio.h>

const DODGERS = 0;
const GIANTS = 1;

void main(void)
{
    int scoreboard [2][9];    // An array two rows by nine columns
    int team, inning;
    int score, total;

    randomize();              // Initialize random number generator

    // Generate the scores
    for (team = DODGERS; team <= GIANTS; team++) {
        for (inning = 0; inning < 9; inning++) {
            score = random(3);
            if (score == 2)    // 1/3 chance to score at least a run
                score = random(3) + 1;    // 1 to 3 runs
            if (score == 3)
                score = random(7) + 1;    // Simulates chance of a big
                // inning of 1 to 7 runs
            scoreboard[team][inning] = score;
        }
    }

    // Print the scores
    cout << "\nInning\t1  2  3  4  5  6  7  8  9  Total\n";
    cout << "Dodgers\t";
    total = 0;
    for (inning = 0; inning <= 8; inning++) {
        score = scoreboard[DODGERS][inning];
        total += score;
        cout << score << "  ";
    }
    cout << total << "\n";

    cout << "Giants\t";
    total = 0;
    for (inning = 0; inning < 9; inning++) {
        score = scoreboard[GIANTS][inning];
        total += score;
        cout << score << "  ";
    }
}
```

```
cout << total << "\n" ;
}
```

Not surprisingly, when two array dimensions are involved, two nested **for** loops are often used to access the array elements. The inner loop, using *inning* as the counter variable, steps through the nine innings, while the outer switches from team 0 (Dodgers) to team 1 (Giants).

Within the body of the loop, the **random** function (defined in the header file `stdlib.h`) generates the score. When **random** is first called as `random(3)`, there is a 1/3 chance of *score* getting the value 2. (**random** returns a value between 0 and one less than its parameter). If *score* has the value 2, it is recalculated as a random number between 1 and 3 runs. Finally, if *score* now is 3, a final random score of 1 to 7 runs is generated. The series of **if** statements thus attempts to simulate the occasional big inning.

Two **for** loops then print out the scores, totaling them as they go. Each one prints out one team's scores by using the team name as a constant value for the first dimension of the array, and varying the *inning*. Here's a sample run:

Inning	1	2	3	4	5	6	7	8	9	Total
Dodgers	0	1	0	1	0	1	1	2	0	6
Giants	1	1	0	2	0	0	4	1	0	9

Arrays and strings

Strings and arrays are very similar. In fact, a *string* is simply an array of **char** values with a null character stuck on the end. The following program declares a character array (string), then lets you store a value in it and extract a "substring" from the full string (load and run `INTRO29.CPP`):

Remember that string arrays need an extra element for the ending null character.

```
//INTRO29.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    char string[80];           // Has 79 usable elements
    int pos, num_chars;

    cout << "Enter a string for the character array: ";
    cin.get(string,80,'\n');
    cout << "How many characters do you want to extract? ";
```

```

cin >> num_chars;

for (pos = 0; pos < num_chars; pos++)
    cout << string[pos];
    cout << '\n';

return 0;
}

```

Here's a sample run:

```

Enter a value for the character array: The quick brown fox
How many characters do you want to extract? 9
The quick

```

➔ It is usually more convenient to use the library routines that manipulate strings such as **strcat()** or **strtok()** to deal with strings because they automatically take care of putting a null character at the end of the string. If you create a string with an array declaration and want to use these routines with it, you must supply the null character yourself. (The array must be large enough to hold the desired string *plus* the null character, which you must put at the end of the string.)

Renaming types

As you get into more complex data structures, it is helpful to be able to assign a meaningful name to them. You can do so using the **typedef** keyword. **typedef** gives a name to some combination of the standard C++ data types. Here are some examples:

```

#include <iostream.h>

typedef unsigned char uchar;

int main()
{
    uchar greek_alpha = 224, greek_beta = 225;
    cout << '\n' << greek_alpha << greek_beta;
    return 0;
}

```

The **typedef** declaration gives the new name **uchar** to the type **unsigned char**. (This is a character type that can hold all 256 characters of the extended PC character set.) The first statement inside **main()** declares *greek_alpha* and *greek_beta* to be variables of type **uchar**, and the **cout** statement displays their values.

typedef doesn't actually create new data types. It just makes it easier to remember what kind of data you are dealing with later in your program. As you will see, it is particularly useful for giving names to more complex data types, such as enumerations and structures.

Enumerated types

Data sometimes fits logically into an ordered series where one item follows another—for example, the days of the week. It is convenient to use a loop to step through such values.

```
for (day = mon; day <= fri; day++)
    /* add hours worked that day to total for week */
```

Since a loop steps through numeric values, you need to give *mon* an integer value, *tues* the next integer, and so on. You could do this with **#define** statements:

```
#define mon 0
#define tues 1
#define wed 2
#define thurs 3
#define fri 4
```

However, the **enum** (enumerated) type offers a more compact way to do this, as shown in this example (load and run **INTRO31.CPP**):

```
//INTRO31.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

int main()
{
    enum {mon, tues, wed, thurs, fri};
    int day;

    for (day = mon; day <= fri; day++)
        cout << day << '\n';

    return 0;
}
```

The first declaration automatically assigns the value 0 to *mon*, 1 to *tues*, 2 to *wed*, and so on. These names can now be used to specify the starting value and limit for a **for** loop, as shown.

Note that the numbers used in an enumeration don't have to be consecutive—you can override the default order by assigning values as follows:

```
enum {touchdown = 6, field_goal = 3, safety = 2, point_after = 1};
```

The statement

```
cout << "\n" << touchdown + point_after + field_goal;
```

displays the result 10.

Combining data into structures

Arrays and enumerations give two powerful ways to handle sets of data values of the same type. A *structure* bundles together a set of data values of different types (or at least values that have different meanings). For example, information about an employee could be stored in the following structure:

```
struct employee {  
    char last_name[30];  
    char first_name[20];  
    char initial;  
    double employee_no;  
    char dept_code[3];  
    float annual_salary;  
};
```

You have now defined a new data type **employee** and specified the list of data items (*members*) that a variable of **employee** type will have. As you can see, a variety of data types can be used—in this example, character arrays, single characters, doubles, and floats.

Using parts of a structure

The next example bundles together the information that the earlier program PLANETS.CPP needed to draw a planet. We'll use it later to illustrate how parts of a structure can be initialized and accessed. Here they are displayed (load and run INTRO32.CPP):

A field of a structure is referred to by using the structure name followed by a period and the member name. Thus mars.distance is the member containing the distance of Mars from the Sun in astronomical units.

```
//INTRO32.CPP--Example from Chapter 3, "An Introduction to C++"  
  
#include <iostream.h>  
#include <string.h>  
  
struct planet {  
    char name[10];  
    float distance;  
    float radius;
```

```

};

planet mars;

int main()
{
    strcpy(mars.name, "Mars");
    mars.distance = 1.5;
    mars.radius = 0.4;

    cout << "Planetary statistics:\n";
    cout << "Name: " << mars.name << '\n';
    cout << "Distance from Sun in AU: " << mars.distance << '\n';
    cout << "Radius in Earth radii: " << mars.radius;

    return 0;
}

```

The **struct** keyword gives the name *planet* to a **struct** (structure) consisting of three members, or fields: The planet name is an array of characters, while the distance and radius are floating-point values. The declaration `planet mars;` creates a variable *mars* whose type is **planet** (in other words, it makes a copy of the structure defined earlier). In **main**, values are assigned to these three members.

As shown here, the library function **strcpy** is handy for copying a string value into structure fields that hold arrays of characters. The numeric values are simply assigned as usual.

The **cout** statements at the end of the program use the *name.member* notation to reference and print out the values that were just assigned.

Pointers

Every variable has a unique memory address that indicates the beginning of the memory area occupied by its value. The amount of memory used depends on the type of data involved. In the case of an **int**, this area is 2 bytes, while a **float** uses 4 bytes. For an array, the area occupied is equal to the number of elements times the size needed for one value of the declared data type. For a structure, the area used is equal to the sum of the areas needed for the structure's members, plus some padding if needed. Because in all cases data is stored in an orderly, predictable way, it is possible to access data by using a variable that contains the relevant address. Such a variable is called a *pointer*.

Why are pointers useful? First, they allow you to access and manipulate structured data easily, without having to move the data itself around in memory. For example, by being set to the addresses of consecutive elements in an array, a pointer can be used to initialize the array or to retrieve data from it. By adding to or subtracting from the pointer, you point to different data items.

Pointers can also be used to allow a function to receive and change the value of a variable. This can avoid the need for declaring global variables.

Also, pointers are needed for allocating memory while your program is running: In essence, you ask for some free memory (via a function such as **malloc**), and get back a pointer to the first available address.

Memory allocation is discussed further in Chapter 18, "Memory management."

Declaring and using a pointer

A pointer declaration takes this form:

```
type *name
```

where *type* is any data type. Here are some example pointer declarations:

```
int *intptr;           // Points to an integer
float *fltpr;         // Points to a floating-point value
char *string;         // Points to a character value
```

Most programmers use names that include an abbreviation of the word pointer; for example, intptr.

You can declare a pointer to any object in memory, including arrays, structures, functions, and even other pointers.

To access the value pointed to by a pointer, precede the pointer name with an asterisk. For example, **intptr* yields the value stored at the address stored in the pointer *intptr*. Because this value is reached indirectly (rather than the case of a regular variable, where the value is stored at the variable's own address), this process is called *indirection* or *dereferencing*.

The following program declares a pointer and uses it to retrieve the value of a variable:

Load and run INTRO33.CPP.

```
//INTRO33.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>
int main()
{
    int intvar = 10;
    int *intptr;
```

```

    intptr = &intvar;

    cout << "\nLocation of intvar: " << &intvar;
    cout << "\nContents of intvar: " << intvar;
    cout << "\nLocation of intptr: " << &intptr;
    cout << "\nContents of intptr: " << intptr;
    cout << "\nThe value that intptr points to: " << *intptr;

    return 0;
}

```

Here's the output:

*The values of the addresses
may vary.*

```

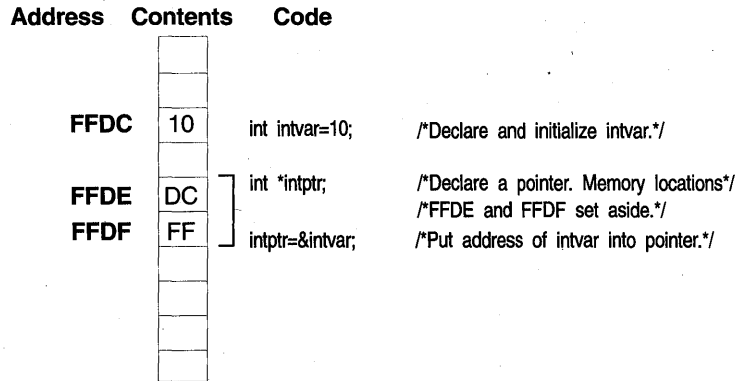
Location of intvar: 0x8f8efff4
Contents of intvar: 10
Location of intptr: 0x8f8efff2
Contents of intptr: 0x8f8efff4
The value that intptr points to: 10

```

First, the **int** variable *intvar* is declared and assigned a value of 10. The next declaration declares an **int *** variable called *intptr*. (Read this declaration as “*intptr*, a pointer to an integer value.”) The next statement assigns *intptr* the address of the variable *intvar* (notice the **&**, or address-of operator). A pointer must always be assigned the address of the object it is intended to point to. If you neglect this, the pointer will contain a random address. If you try to store something at such an address by means of the pointer, you risk destroying part of your program or data, or even halting the system.

As you can see from the output of the **cout** statements, *intvar* and *intptr* are stored at different addresses, since they are different variables. (Their addresses happen to be close together, but that has nothing to do with how pointers work.) *intptr* contains the address of *intvar* (which had been assigned to it previously). The last **cout** statement prints the value pointed to by *intptr*; in other words, the contents of the address stored there. Since that address is that of *intvar*, the value pointed to is the contents of *intvar*. The next figure may help you visualize all this.

Figure 3.8
How pointers point (and
what they point to)



Pointers and strings

You've seen that you can access individual characters from a string by using indexing. For example, if you declare a string `char name[20]` and store the string "Thomas" in it, the value of `name[2]` is the character `o` (remembering that the count starts at 0: `name[0]` is `T`). An alternative way to handle strings is to declare a pointer to character and use it to manipulate the string.

This example is based on the earlier example of a string as a character array, but has been rewritten to use a pointer. To try it out, load and run `INTRO34.CPP`.

```
//INTRO34.CPP-Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>
int main()
{
    char name[40];
    char *str_ptr = name;
    int pos, num_chars;

    cout << "Enter a string for the character array: ";
    cin.get(name,40,'\n');
    cout << "How many characters do you want to extract? ";
    cin >> num_chars;

    for (pos = 0; pos < num_chars; pos++)
        cout << *str_ptr++;
    cout << '\n';

    return 0;
}
```

Notice that `str_ptr` is declared to be a pointer to character, and then assigned the address of the character array `name`. This could have been written in two separate statements:

```
char *str_ptr;  
str_ptr = name;
```

but by now you know that C++ programmers seldom use two statements when one will do. Also notice that the address assigned is *name*, not *&name*. Referring to the name of an array (or structure) gets you the first address used to store the array's values. This is equivalent to saying *&name[0]* (the address of the first element of the array *name*); you will sometimes see the latter notation.

The rest of the program works in the same way as before, until you get to the **for** loop that extracts the requested substring. In the old version, the character being retrieved each time the loop executes was indexed from the array by using the expression *name[pos]*. Here, however, the pointer is used, so the reference is to the value currently being pointed at: **str_ptr++*. (The pointer is incremented each time so that it points to the next value.)

Pointer arithmetic

Noncharacter arrays are handled with pointers in the same way as strings are. You increment the pointer to point to the next element in the array; you decrement it to point to the previous item. You have to check, of course, to make sure that you aren't pointing to a location outside the bounds of the array. You usually do this by setting the appropriate limit for the loop statement used. In *INTRO34.CPP*, for instance, if the variable *num_chars* is set to a number greater than 39, you will exceed the bounds of the array and get garbage out (try it).

It is important to remember that when the pointer *ptr* is incremented, it does *not* necessarily point to the next address—in fact, it usually doesn't. The distance between addresses pointed to is equal to the size of the data type to which the pointer points. An **int** pointer points two addresses ahead when it's incremented; a **double** pointer points eight addresses ahead. C++ handles this automatically.

Pointers, structures, and lists

You may remember that you get the value of a member field of a structure by using the notation *structure_name.member_name*, so that the salary field in the employee structure named *jim* would be *jim.salary*. How do you access structures and parts of structures

with a pointer? The following example shows how (load and run SOLAR.CPP):

Figure 3.9 illustrates how this program works.

```
//SOLAR.CPP--Example from Chapter 3, "An Introduction to C++"
#include <graphics.h>
#include <iostream.h>
#include <string.h>

struct planet {
    char name[10];
    float distance;
    float radius;
    int color;
    int fill_type;
};

planet solar_system[9];
planet *planet_ptr;
int planet_num;

int main()
{
    strcpy(solar_system[0].name,"Mercury");
    solar_system[0].distance = 0.4;
    solar_system[0].radius = 0.4;
    solar_system[0].color = EGA_YELLOW;
    solar_system[0].fill_type = EMPTY_FILL;

    planet_ptr = solar_system;
    planet_ptr++;          // Point to second planet structure
    strcpy (planet_ptr->name,"Venus");
    planet_ptr->distance = 0.7;
    planet_ptr->radius = 1.0;
    planet_ptr->color = EGA_BROWN;
    planet_ptr->fill_type = SOLID_FILL;

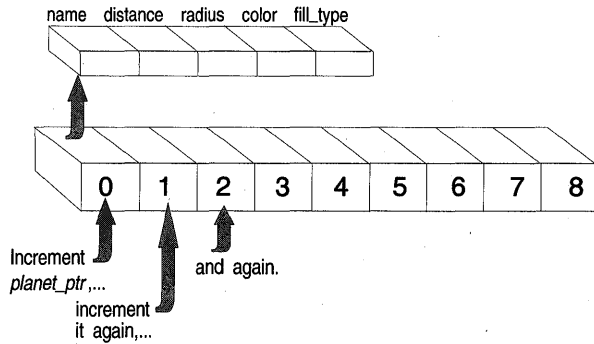
    planet_ptr = solar_system;          // Reset to first element
    for (planet_num = 0; planet_num < 2; planet_num++, planet_ptr++)
    {
        cout << " \nPlanetary statistics: ";
        cout << " \nName: "<< planet_ptr->name;
        cout << " \nDistance from Sun in AU: "<< planet_ptr->distance;
        cout << " \nRadius in Earth radii: "<< planet_ptr->radius;
        cout << " \nColor constant value "<< planet_ptr->color;
        cout << " \nFill pattern constant value "
            << planet_ptr->fill_type;
    }

    return 0;
}
```

The → notation is explained on page 119.

The following figure illustrates how the previous code might “look” during the first few steps through its **for** loop:

Figure 3.9
Using pointers to access an
array of structures



planet solar_system[9]
Array of nine planet structures

The *planet* structure is an expanded version of the one shown earlier, with members added for the color and fill type. The next declarations

```
planet solar_system[9];
planet *planet_ptr;
```

specify an array, *solar_system*, whose nine members are copies of the *planet* structure and *planet_ptr*, a pointer to a *planet* structure.

The first group of statements in **main** initialize the members of the first planet structure, using array and structure notation. To refer to a member of a structure in an array of structures, use the form

```
array_name[index].member_name
```

Thus the distance of the fourth planet in *solar_system* can be referred to as *solar_system*[3].distance.

Since planet_ptr was declared as a pointer to structure type planet, the built-in pointer arithmetic takes care of moving the pointed-to address enough bytes to reach the next element of solar_system.

The second block of statements initialize a *planet* structure using pointers. Before you can use a pointer, you must assign it a valid address. The statement `planet_ptr = solar_system` sets *planet_ptr* to point to the first element of the array *solar_system*. This is the element that has just been initialized, containing information for the planet Mercury. Therefore, the statement `planet_ptr++` points to the next (second) element.

With the pointer properly pointed, information about the second planet (Venus) is assigned to the second element of *solar_system*. Here, however, rather than array index notation, you see the use

of the pointer *planet_ptr* to access the members of the *planet* structure. This takes the form

```
pointer_name->member_name
```

so that, for example, the *distance* member for the *planet* structure currently being pointed to is `planet_ptr->distance`.

The rest of the program displays the two planet structures that have been initialized. First, *planet_ptr* is set back to the first element by being assigned the address *solar_system*. The **for** loop increments *planet_ptr* and obtains the structure's member values using the notation just discussed. Notice that the reference `planet_ptr->distance` is a bit less cumbersome than

```
solar_system[index].distance
```

where *index* is the current element number.

In fact, for any array, the index of an array is actually a pointer to the address of the array plus the index value, which internal pointer arithmetic converts to the array address plus

```
index * sizeof(type)
```

where *type* is the declared type of the array, and **sizeof** is a C++ operator that returns the number of bytes used by a type or variable.

Using pointers to return values from functions

Pointers also allow you to change the actual value of the variable or variables used in calling a function. So far, functions have been called only with constant values or the *names* of variables. When you make a call such as `draw(x_cor, y_cor, size, color)`, you are passing the *values* of the specified variables to the function **draw**. The function actually gets *copies* of these values and can refer to them by name and manipulate them, but this has no effect on the actual variables used by the caller.

Sometimes it is useful to be able to call a function using variable names and have the function actually change the values of the variables themselves. This function is an example:

```
void swap(int *a, int *b)
```

swap swaps the values of the two variables with which the function is called. In order for the function to access the variables themselves, however, it must have their *addresses*, so it can write the new values back to their location in memory. Therefore, the

parameters are pointers to the appropriate variables, not the variables themselves. Since a pointer refers to an address, you would call this function to swap the variables *x* and *y* using the statement `swap(&x, &y)`.

Here is the definition for the **swap** program, together with a **main** function that tests it (load and run `INTRO35.CPP`).

```
//INTRO35.CPP--Example from Chapter 3, "An Introduction to C++"
#include <iostream.h>

void swap(int *, int *); // This is swap's prototype

int main()
{
    int x = 5, y = 7;
    cout << "\n x is " << x << " and y is " << y << '\n';
    swap(&x, &y);
    cout << "\n x is now " << x << " and y is now " << y << '\n';

    return 0;
}

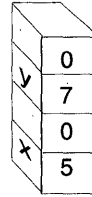
void swap(int *a, int *b) // swap is actually defined here
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Notice that the **swap** function uses indirection (the ***** notation) to refer to the *values* contained in the variables *x* and *y*. The value of *x* is first stored in a temporary variable, then the value of *y* is stored in *x*. The former value of *x* is then obtained from the temporary variable and stored in *y*.

Figure 3.10
Using pointers in a function

Initialize variables:

```
int x = 5, y = 7;
```

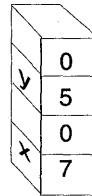


Call the function:

```
swap(&x, &y);
```

Results:

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```



When you use pointers, a function is not limited to just returning one value via a **return** statement. A function can change the values of any variables it is given access to. While this could also be achieved (in this example) by making *a* and *b* global variables (declaring them outside of a function definition), it is easy to accidentally change a global variable. Pointers keep the transaction private.

Using system resources

Thus far, the example programs have operated nearly in a vacuum. Some programs read data that you typed at the keyboard, and every program displayed something on the screen. None of the data was stored in permanent form—if you want the data back, you must run the program again. In real applications, programs usually have to read in the bulk of their data from the disk drive, a communications port, or some other source. When the data has been processed, the program may need to send it to

the printer or write it to a disk file for later use. This is true of word processors, spreadsheets, and databases, for example.

You have actually been using files all along. Every C++ program has automatic access to five streams, as shown in the next table.

Table 3.8
Preopened streams in Turbo
C++

Name	Function	Connected to
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error	Screen
clog	Buffered cerr	
cprn	Printer (DOS only)	Printer port

Except for *clog* each of the objects has a similar predefined file in ANSI C.

All of the default streams can be redirected in various ways. Due to their default connections, your Turbo C++ programs expect to get input from the keyboard and send their output to the screen.

This section shows you how you can use Turbo C++ to read data from a disk file and copy that data to another disk file. Conceptually, Turbo C++ sets up something called a *stream* through which the data moves. There are lower-level ways to work with MS-DOS files, and library functions for dealing with them, but the stream features are recommended for portability. They allow you to use files without worrying about the target machine's operating system.

A stream represents a file on the disk or some other device from which data can be read or to which it could be sent. (Many devices are used for input or output but not both — you can't usually read data from a printer, and data sent to a keyboard won't accomplish much.) In the following example program we don't manipulate the stream directly. Instead, the library file `fstream.h` provides access to its member classes **ifstream** and **ofstream** which contain the constructors and functions for creating and handling the files.

To open a stream, you use a predefined function contained in a library file. In the following example we use **get()** from `fstream.h`. In the case of the program example, `INTRO36.CPP`, **get(ch)** extracts characters from the file `OLDFILE.TXT` into the variable **char ch**; until an end-of-file is encountered. The **put(ch)** function inserts the same characters into the newly created file `NEWFILE.TXT` it also encounters the end-of-file symbol.

If OLDFILE.TXT does not exist, or is not in the current directory, the **cerr** statement will terminate the program and display "Cannot open OLDFILE.TXT for input". If the program is unable to create, or find and overwrite NEWFILE.TXT the program will terminate with the statement "Cannot open NEWFILE.TXT for output". The work of the program is done in the **while** loop:

```
while (f2 && f1.get(ch))
    f2.put(ch);
```

The **while** loop is where the actual copying of OLDFILE.TXT to NEWFILE.TXT is accomplished. To check the successful execution of the program first check the user window for any error statements as indicated above. If there are no error statements press F10, select the FILE window, and then select the DOS shell option. When at the DOS command line you can display the file on the screen with the DOS command "TYPE NEWFILE.TXT". NEWFILE.TXT will contain an exact copy of OLDFILE.TXT. In the event that the program cannot find and open the file OLDFILE.TXT it will still create the file NEWFILE.TXT, but the file will be empty.

Text streams are used for normal DOS files. Therefore, files are opened in text mode by default when using the file-stream classes. When the file is extracted all of the DOS carriage return/line feed sequences are converted to the '\n' character. When the file is output to the target file (in this case, newfile.txt) all '\n' characters are translated back to DOS carriage return/line feed sequences. This maintains compatibility with DOS. The file-stream classes (**ifstream** and **ofstream**) allow you to set additional parameters to override this default procedure, giving you the flexibility to open files in binary mode, delete files, append files, and so on.

The following program copies the contents of OLDFILE.TXT to NEWFILE.TXT:

Load and run INTRO36.CPP.

```
//INTRO36.CPP--Example from Chapter 3, "An Introduction to C++"
#include <fstream.h>

int main()
{
    char ch;
    ifstream f1 ("OLDFILE.TXT");
    ofstream f2 ("NEWFILE.TXT");

    if (!f1) cerr << "Cannot open OLDFILE.TXT for input";
    if (!f2) cerr << "Cannot open NEWFILE.TXT for output";
```

```
while (f2 && f1.get(ch))
    f2.put(ch);
return 0;
}
```

Opening a stream

This program is a simple demonstration of using streams to work with disk files. Here a file is opened for reading by **ifstream**, and another file is created and opened for writing by **ofstream**. By default the **ifstream** is opened for reading, and the **ofstream** is opened for writing. As mentioned above both can take a second argument specifying alternative modes of opening. It is also possible to open a file for both input and output. We will leave a more detailed discussion of the **iostreams** library and streams to the following chapters. The purpose of this example program is to introduce you to the basics of stream operations. A comprehensive discussion of streams requires some groundwork in the concept of object-oriented programming (OOP).

We have covered a wide range of information in this chapter with the goal of giving you a chance to navigate the headwaters of the C++ programming language as quickly and easily as possible. Beyond this point, C++ begins to expand beyond the procedural programming style we have focused on in this chapter.

In Chapter 4 you will be introduced to the concepts of classes, inheritance, encapsulation and other topics to help you on your way towards gaining a comfortable and confident mastery of object-oriented programming with Turbo C++.

Object-oriented programming with C++

This chapter covers the basic ideas of object-oriented programming with C++; Chapter 5, "Hands-on C++," gives you practical examples of OOP programming with Turbo C++.

In the previous chapter, we introduced you to the "procedural" side of C++: the basic syntax and control structures of the language. In this chapter we'll give you the feel and flavor of *object-oriented* programming in C++, and introduce the important new design concepts that make C++ a genuinely new and better language rather than just an "improved C." We'll demystify some of the jargon and combine a little theory with simple, illustrative programs. The source code for these examples is provided on your distribution diskettes so you can study, edit, compile, and run them. (The graphics examples, of course, will run only if you have a graphics adapter and monitor. Any CGA, EGA, VGA, or Hercules setup will do.)

Turbo C++ provides all the features of AT&T's C++ version 2.1. C++ is an extension of the popular C language, adding special features for object-oriented programming (OOP).

OOP is a method of programming that seeks to mimic the way we form models of the world. To cope with the complexities of life, we have evolved a wonderful capacity to generalize, classify, and generate abstractions. Almost every noun in our vocabulary represents a class of objects sharing some set of attributes or behavioral traits. From a world full of individual dogs, we distill an abstract class called *dog*. This allows us to develop and process ideas about canines without being distracted by the details concerning any particular dog. The OOP extensions in C++ exploit

this natural tendency we have to classify and abstract things—in fact, C++ was originally called “C with Classes.”

Three main properties characterize an OOP language:

- *Encapsulation*: Combining a data structure with the functions (actions or *methods*) dedicated to manipulating the data. Encapsulation is achieved by means of a new structuring and data-typing mechanism—the *class*.
- *Inheritance*: Building new, *derived* classes that inherit the data and functions from one or more previously-defined *base* classes, while possibly redefining or adding new data and actions. This creates a *hierarchy* of classes.
- *Polymorphism*: Giving an action one name or symbol that is shared up and down a class hierarchy, with each class in the hierarchy implementing the action in a way appropriate to itself.

Borland’s C++ gives you the full power of object-oriented programming:

- more control over your program’s structure and modularity
- the ability to create new data types with their own specialized operators
- and the tools to help you create reusable code

All these features add up to code that can be more structured, extensible, and easier to maintain than that produced with non-object-oriented languages.

To achieve these important benefits of C++, you may need to modify ways of thinking about programming that have been considered standard for many years. Once you do that, however, C++ is a simple, straightforward, and superior tool for solving many of the problems that plague traditional software.

Your background may affect the way you look at C++:

If you are new to C and C++. You may at first have some difficulty with the new concepts discussed in this chapter, but working through (and experimenting with) the examples will help make the ideas concrete. Before you begin, you should make sure you understand the basic elements of the C language (you may wish to review Chapter 3 before continuing here). As a beginner, you have one very real advantage: You probably have fewer old programming habits to unlearn.

If you are an experienced C programmer. C++ builds upon the existing syntax and capabilities of C. This makes the learning curve much gentler than if you had to learn a whole new language. It also allows you to port existing C programs to C++ with a minimum of recoding. You aren't losing C's power and efficiency: You're adding the representational power of classes and the security of controlling access to internal data.

If you program in Turbo Pascal 5.5 or greater. Turbo Pascal 5.5 and greater embody many of the same object-oriented features found in C++. While you will have to deal with basic syntax differences between the two languages, you will find that Turbo Pascal's *objects* and Turbo C++'s *classes* are structured similarly. You will recognize C++ *member functions* as being like Turbo Pascal's *methods*, and may note many other similarities. The main difference you will observe is that C++ has tighter control over data access.

If you are experienced in another object-oriented programming language. You will find some differences in C++:

- First, the syntax of C++ is that of a traditional, procedural language, as covered in chapter 3, "An introduction to C++."
- Second, the way C++ and Smalltalk actually deal with objects during compilation is different. Smalltalk's binding is done completely at run time (*late binding*); C++ allows both compile-time (early) binding and late binding.

In this chapter, we begin by describing the three key OOP ideas — encapsulation, inheritance, and polymorphism — in more detail. The first listings show fragments of code to illustrate each topic. Later, we present complete, compilable programs. The main example develops object-oriented representations useful for graphics, but occasional side tours show how C++ works with strings and other data structures.

Encapsulation

How does C++ change the way you work with code and data? One important way is *encapsulation*: the welding of code and data together into a single class-type object. For example, you might have developed a data structure, such as an array holding the information needed to draw a character font on the screen, and

code (functions) for displaying, scaling and rotating, highlighting, and coloring your font characters.

In traditional C, the usual solution is to put the data structures and related functions into a single separately compiled source file in an attempt to treat code and data as a single module. While this is a step in the right direction, it isn't good enough. There is no explicit relationship between the data and the code, and you or another programmer can still access the data directly without using the functions provided. This can lead to problems. For example, suppose that you decide to replace the array of font information with a linked list? Another programmer working on the same project may decide that she has a better way to access the character data, so she writes some functions of her own that manipulate the array directly. The problem is that the array isn't there any more!

C++ comes to the rescue by extending the power of C's **struct** and **union** keywords, and by adding a keyword not found in C: **class**. All three keywords are used in C++ to define *classes*.

*In this manual, we use bold type to distinguish the keyword **class** from the generic word "class."*

In C++, a single class entity (defined with **struct**, **union**, or **class**) combines functions (known as *member functions*) and data (known as *data members*). You usually give a class a useful name, such as **Font**. This name becomes a new type identifier that you can use to declare *instances* or *objects* of that class type:

```
class Font {  
    // here you declare your members: both data and functions;  
    // don't worry how for the moment.  
};  
Font Tiffany;           // declares Tiffany to be of type class  
                        // Font.
```

Note that in Turbo C++ you can now use two slashes (*//*) to introduce a single-line comment in both C and C++. You can still use the */* */* comment characters if you prefer them; in fact, they are especially useful for long comments.

Warning! Use of the *//* comments is not usually portable to other C compilers. However, it is portable to other C++ compilers.

The variable *Tiffany* is an *instance* (sometimes called an *instantiation*) of the class **Font**. You can use the class name **Font** very much like a normal C data type. For example, you can declare arrays and pointers:

```
Font Times[10];    // declare an array of 10 Fonts
Font* font_ptr;    // declare a pointer to Font
```

A major difference between C++ classes and C structures concerns the accessibility of members. The members of a C structure are freely available to any expression or function within their scope. With C++, you can control access to **struct** and **class** members (code and data) by declaring individual members as **public**, **private**, or **protected**. (A C++ union is more like a C union, with all members **public**.) We'll explain these three access levels in more detail later on.

*C++ **structs** and **unions** are not quite the same as the C versions.*

C++ structures and unions offer more than their C counterparts: they can hold function declarations and definitions as well as data members. In C++, the keywords **struct**, **union**, and **class** can all be used to define classes.

- A class defined with **struct** is simply a class in which all the members are **public** by default (but you can vary this arrangement if you wish).
- A class defined with **union** has all its members **public** (this access level cannot be changed).
- In a class defined with **class**, the members are **private** by default (but there are ways of changing their access levels).

So, when we talk about classes in C++, we include structures and unions, as well as types defined with the keyword **class**.

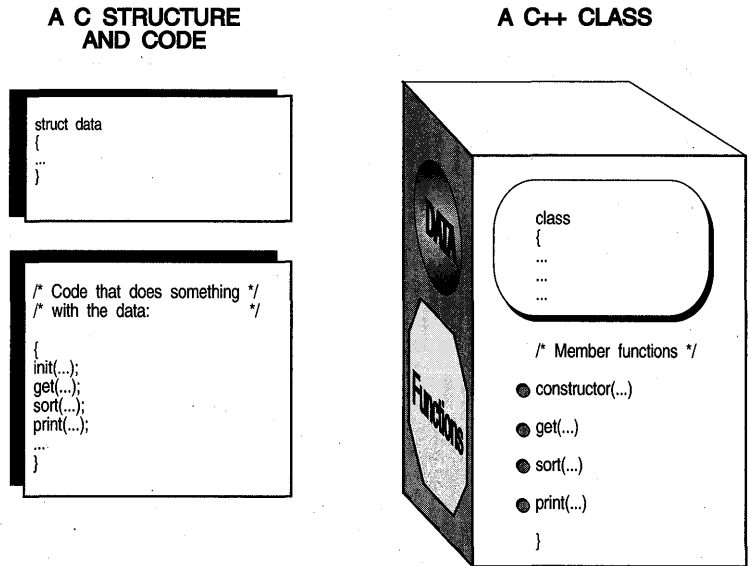
Typically, you restrict member-data access to member functions: you usually make the member data **private** and the member functions **public**.

Returning to the problem of handling fonts, how does the C++ class concept help?

By creating a suitable **Font** class, you can ensure that the private font data can be accessed and manipulated *only* through the public **Font** member functions that you have created for that purpose. You are now free at any time to change the font data structure from an array to a linked list, or whatever. You would, of course, need to recode the member functions to handle the new font data structure, but if the function names and arguments are unchanged, programs (and programmers) in other parts of your system will be unaffected by your improvements!

The next figure compares the ways C and C++ provide access to a font.

Figure 4.1
Traditional C versus
encapsulated C++



Thus the technique of encapsulation in classes helps provide the very real benefit of *modularity*, as found in languages such as Ada and Modula-2. The C++ class establishes a well-defined interface that helps you design, implement, maintain, and reuse programs. Debugging a C++ program is often simpler since many errors can be quickly traced to one particular class.

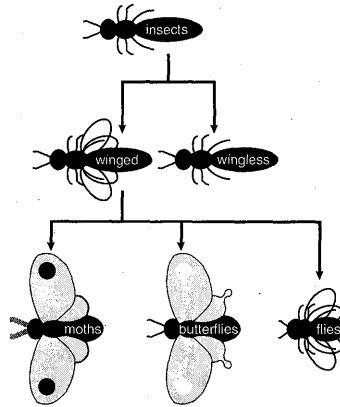
The class concept leads to the idea of *data abstraction*. Our font data structure is no longer tied to any particular physical implementation; rather, it is defined in terms of the operations (member functions) allowed on it. At the same time, the traditional C philosophy that views a program as a collection of *functions*, with data as second-class citizens, has also shifted. The C++ class weds data and function as equal, interdependent partners.

Inheritance

The descriptive branches of science (required before the explanatory and predictive aims of science can bear fruit) spend much time classifying objects according to certain traits. It often helps to organize your classification as a family tree with a single overall category at the root, with subcategories branching out into subsubcategories, and so on.

Entomologists, for example, classify insects as shown in Figure 4.2. Within the phylum *insect* there are two divisions: winged and wingless. Under winged insects is a larger number of categories: moths, butterflies, flies, and so on.

Figure 4.2
A partial taxonomy chart of
insects



This classification process is called *taxonomy*. It's a good starting metaphor for OOP's inheritance mechanism.

The questions we ask in trying to classify some new animal or object are these: *How is it similar to the others of its general class? How is it different?* Each different class has a set of behaviors and characteristics that define it. We begin at the top of a specimen's family tree and start descending the branches, asking those questions along the way. The highest levels are the most general, and the questions the simplest: Wings or no wings? Each level is more specific than the one before it, and less general.

Once a characteristic is defined, all the categories *beneath* that definition *include* that characteristic. So once you identify an insect as a member of the order *diptera* (flies), you needn't make the point that a fly has one pair of wings. The species *fly* inherits that characteristic from its order.

OOP is the process of building class hierarchies. One of the important things C++ adds to C is a mechanism by which class types can inherit characteristics from simpler, more general types. This mechanism is called *inheritance*. Inheritance provides for commonality of function while allowing as much specialization as needed. If a class **D** inherits from class **B**, we say that **D** is the *derived* class and **B** is the *base* class.

It is by no means a trivial task, though, to establish the ideal class hierarchy for a particular application. The insect taxonomy took hundreds of years to develop, and is still subject to change and acrimonious debate. Before you write a line of C++ code, you must think hard about which classes are needed at which level. As the application develops, you may find that new classes are required that fundamentally alter the whole class hierarchy. Remember also that a growing number of vendors are supplying Turbo C++ compatible libraries of classes. So don't reinvent too many wheels.

Occasionally, you encounter a class that combines the properties of more than one previously established class. C++ version 2.0 offers a mechanism (not found in earlier C++ versions) known as *multiple inheritance*, whereby a derived class can inherit from two or more base classes. You'll see later how this is achieved as a logical extension of the single inheritance mechanism.

Polymorphism

The word *polymorphism* comes from the Greek: "having many shapes." Polymorphism in C++ is accomplished with *virtual* functions. Virtual functions let you use many versions of the same function throughout a class hierarchy, with the particular version to be executed being determined at run time (this is called *late binding*).

Overloading

In C, you can only have one function with a given name. For example, if you declare and define the function

```
int cube (int number);
```

you can now get the cube of an integer. But suppose you want to cube a **float** or a **double**? You can of course declare functions for these purposes, but they can't use the name **cube**:

```
float fcube (float float_number);  
double dcube (double double_number);
```

In C++, however, you can *overload* functions. This means that you can have several functions that have the same name but work with different types of data. Thus you can declare:

```
int cube (int number);  
float cube (float float_number);  
double cube (double double_number);
```

As long as the argument lists are all different, C++ takes care of calling the correct function for the argument given. If you have the call **cube**(10); the **int** version of **cube** is called, while if you call **cube**(2.5); the **double** version will be called. If you call **cube**(2.5F), then you are passing a floating-point literal rather than a **double**, and the **float** version will be called. Even operators such as + can be overloaded and redefined so they work not only with numbers, but with graphic objects, strings, or whatever is appropriate for a given class.

Modeling the real world with classes

The C++ class provides a natural way of building computer models of real-world systems—indeed, Bjarne Stroustrup devised the language at AT&T Bell Labs in order to model a large telephone switching system.

There have been many C++ applications in the motor industry. When modeling vehicles, for instance, you would be interested in both the physical description (the number of tires, engine power, weight, and so on) and the behavior (acceleration, breaking, steering, fuel consumption). A **Car** class could encapsulate the physical parameters (data) and their behavior (functions) in a very general way. Using inheritance, you might then derive specialized **Sports_car** and **Station_wagon** classes, adding new data types and functions, as well as modifying (overriding) some of the functions of the base class. Much of the coding you have done for the base class(es) is reused or at least recycled.

Building classes: a graphics example

In a graphics environment, a reasonable place to start would be a class that models the physical pixels on a screen with the abstract points of plane geometry. A first try might be a **struct** class called

Point that brings together the *X* and *Y* coordinates as data members:

```
struct Point { // defines a struct class called Point
    int X;      // struct member data are public by default
    int Y;
};
```

When you define a class, you add a new data type to C++. The language treats your new data type in the same way that it treats built-in data types.

You can now declare several particular variables of type **struct Point** (for brevity, we often loosely refer to such variables as being of type **Point**). In C, you would use declarations such as

```
struct Point Origin, Center, Cur_Pos, AnyPoint;
```

but in C++, all you need is

```
Point Origin, Center, Cur_Pos, AnyPoint;
```

The terms object and class instance are used interchangeably in C++.

A variable of type **Point** (such as *Origin*) is one of many possible instances of type **Point**. Note carefully that you assign values (particular coordinates) to *instances* of the class **Point**, not to **Point** itself. Beginners often confuse the data type **Point** with the instance variables of type **Point**. You can write `Center = Origin` (assign *Origin's* coordinates to *Center*), but `Point = Origin` is meaningless.

When you need to think of the *X* and *Y* coordinates separately, you can think of them as independent members (fields) *X* and *Y* of the structure. On the other hand, when you need to think of the *X* and *Y* coordinates working together to fix a place on the screen, you can think of them collectively as **Point**.

Suppose you want to display a point of light at a position described on the screen. In addition to the *X* and *Y* location members you have already seen, you'll want to add a member that specifies whether there is an illuminated pixel at that location. Here's a new **struct** type that includes all three members:

The Boolean type will be familiar to Turbo Pascal programmers.

```
enum Boolean {false, true}; // false = 0; true = 1
struct Point {
    int X;
    int Y;
    Boolean Visible;
};
```

This code uses an enumerated type (**enum**) to create a true/false test. Since the values of enumerated types start at 0, *Boolean* can have one of two values: 0 or 1 (false or true).

Declaring objects

As with other data types, you can have pointers to classes and arrays of classes:

```
Point Origin;           // declare object Origin of type Point
Point Row[80];         // declare an array of 80 objects of type Point
Point *point_ptr;      // declare a 'pointer to type Point'
point_ptr = &Origin;   // point it to the object Origin
point_ptr = Row;       // then point it to Row[0]
```

Member functions

As you saw earlier, C++ classes can contain functions as well as data members. A *member function* is a function declared *within* the class definition and tightly bonded to that class type. (Member functions are known as *methods* in other object-oriented languages, such as Turbo Pascal and Smalltalk.)

Data members are what the class knows; its member functions are what the class does.

Let's add a simple member function, **GetX**, to the class *Point*. There are two ways of adding a member function to a class:

- Define the function inside the class
- Declare it inside the class, then define it outside the class

The two methods have different syntaxes and technical implications.

The first method looks like this:

```
struct Point {
    int X, Y;
    Boolean Visible;
    int GetX() { return X;} // inline member function defined
};
```

Inline functions are discussed in more detail on pages 143 and 179.

This form of definition makes **GetX** an *inline* function by default. Briefly, inline functions are functions “small” enough to be usefully compiled *in situ*, rather like a macro, avoiding the overhead of normal function calls.

Note that the inline member function definition follows the usual C syntax for a function definition: the function **GetX** returns an **int** and takes no arguments. The body of the function, between { and }, contains the statements defining the function—in our case, the single statement, `return X;`.

In the second method, you simply *declare* the member function within **struct Point**, (using normal C function declaration syntax), then provide its full *definition* (complete with the body statements) elsewhere, outside the body of the class definition.

```
struct Point {
    int X, Y;
    Boolean Visible;
    int GetX();    // member function declared
};

int Point::GetX() { // member function defined
    return X;      // outside the class
}
```

The :: is known as the scope resolution operator; it tells the compiler where the function belongs.

Member functions defined outside the class definition still can be made inline (if certain conditions are met), but you have to request this explicitly with the keyword **inline**.

Note carefully the use of the scope resolution operator in **Point::GetX** in the function definition. The class name **Point** is needed to tell the compiler which class **GetX** belongs to (there may be other versions of **GetX** around belonging to other classes). The inside definition did not need the **Point::** modifier, of course, since that **GetX** clearly belongs to **Point**.

Chapter 13 explains class scope in more detail.

The **Point::** in front of **GetX** also serves another purpose. Its influence extends into the function definition, so that the X in return X; is taken as a reference to the X member of the class **Point**. Note also that the body of **Point::GetX** is within the scope of **Point** regardless of its physical location.

Whichever defining method we use, the important point is that we now have a member function **GetX** tied to the class **Point**. Since it is a member function, it can access all the data variables that belong to **Point**. In our simple case, **GetX** just accesses X, and returns its value.

Calling a member function

Now member functions represent operations on objects of their class, so when we call **GetX** we must somehow indicate which **Point** object is being operated on. If **GetX** were a normal C function (or a C++ nonmember function), this problem would not arise—you would simply invoke the function with the expression, **GetX()**. With member functions, you must supply the name of the target object. The syntax used is a natural extension of that used

in C to reference structure members. Just as you would refer to *Origin.X* for the *X* component of the object *Origin*, or to *Endpoint.Y* for the *Y* component of the object *Endpoint*, you can invoke **GetX** with **Origin.GetX()** or **Endpoint.GetX()**. The **."** operator serves as the class component selector for both data and function members. The general calling syntax is

```
class-object-name.member-function-name(argument-list)
```

In the same way, if you had a pointer to a **Point** object, you would use the pointer member selector, **"->"**: `Point_pointer->GetX()`. You'll see many examples of such member function calls in the examples in this chapter.

Constructors and destructors

There are two special types of member functions, *constructors* and *destructors*, that play a key role in C++. To appreciate their importance, a short detour is needed. A common problem with traditional languages is *initialization*: Before using a data structure, you must initialize it and allocate memory for it. Consider the task of initializing the structure defined earlier:

```
struct Point {
    int X;
    int Y;
    Boolean Visible;
};
```

Inexperienced programmers might try to assign initial values to the *X*, *Y*, and *Visible* members in the following way:

```
Point ThisPoint;
ThisPoint.X = 17;
ThisPoint.Y = 42;
ThisPoint.Visible = false;
```

This works, but it's tightly bound to one specific object, *ThisPoint*. If more than one *Point* object needs to be initialized, you'll need more assignment statements that do essentially the same thing. The natural next step is to build an initialization function that generalizes the assignment statements to handle any **Point** object passed as an argument:

```
void InitPoint(Point *Target, Int NewX, Int NewY)
{
    Target->X = NewX;
    Target->Y = NewY;
```

```
        Target->Visible = false;
    }
}
```

This function takes a pointer to a **Point** object and uses it to assign the given values to its members (note again the `->` operator when using pointers to refer to class members). You've correctly designed the function **InitPoint** specifically to serve the structure **Point**. Why, then, must you keep specifying the class type and the particular object that **InitPoint** acts upon? The answer is that **InitPoint** is not a member function. What we really need for true object-oriented bliss is a member function that will initialize any **Point** object. This is one of the roles of the constructor.

C++ aims to make user-defined data types as integral to the language (and as easy to use) as built-in types. Therefore, C++ provides a special type of member function called a *constructor*. A constructor specifies how a new object of a class type will be created, that is, allocated memory and initialized. Its definition can include code for memory allocation, assignment of values to members, conversion from one type to another, and anything else that might be useful. Constructors can be user-defined, or C++ can generate default constructors. Constructors can either be called explicitly or implicitly. The C++ compiler automatically calls the appropriate constructor whenever you define a new object of the class. This can happen in a data declaration, when copying an object, or through the dynamic allocation of a new object using the operator **new**.

Destructors, as the name indicates, destroy the class objects previously created by a constructor by clearing values and deallocating memory. As with constructors, destructors can be called explicitly (using the C++ operator **delete**) or implicitly (when an object goes out of scope, for example). If you don't define a destructor for a given class, C++ generates a default version for you. Later on, we'll be looking at the syntax for defining destructors. First, though, let's see how constructors are made.

The following version of **Point** adds a constructor:

```
struct Point {
    int X;
    int Y;
    Boolean Visible;
    int GetX() {return X;}
    Point(int NewX, int NewY); // constructor declaration
};
```

`Point::Point` indicates that we are defining a constructor for the class `Point`.

```
Point::Point(int NewX, int NewY) // constructor definition
{
    X = NewX;
    Y = NewY;
    Visible = false;
};
```

The constructor definition here is made *outside* the class definition. Constructors can also be legally defined *inside* the class, as inline functions. Or they can be defined outside the class definition and made inline with the keyword **inline**. However, some care is needed: The amount of code generated by a constructor is not always proportional to the visible source code in its definition.

Notice that the name of a constructor is the same as the name of the class: **Point**. That's how the compiler knows that it is dealing with a constructor. Also note that a constructor can have arguments as with any other kind of function. Here the arguments are `NewX` and `NewY`. The constructor body is built just like the body of any member function, so a constructor can call any member functions of its class or access any member data. A constructor, though, *never* has a return type—not even **void**.

Now you can declare a new **Point** object like this:

```
Point Origin(1,1);
```

This declaration invokes the previously defined **Point** constructor for you. As you'll see later, you can have more than one constructor for a class—and, as with other C++ overloaded functions, the appropriate version will be automatically invoked according to the argument lists involved. You'll also see that if you do not define a constructor, C++ generates a default constructor with no arguments.

Another useful trick in C++ is that you can have default values for function arguments:

```
Point::Point(int NewX=0, int NewY=0) // revised constructor definition
{
    // as before
}
```

The declaration,

```
Point Origin(5);
```

would initialize `X` to 5 and `Y` to 0 by default.

Code and data together

One of the most important tenets of object-oriented programming is that the programmer should think of code and data *together* during program design. Neither code nor data exist in a vacuum. Data directs the flow of code, and code manipulates the shape and values of data.

When your data and code are separate entities, there's always the danger of calling the right function with the wrong data or the wrong function with the right data. Matching the two is the programmer's job, and while ANSI C, unlike traditional C, provides good type-checking, at best it can only say what *doesn't* go together.

By bundling code and data declarations together, C++ classes help keep them in sync. Typically, to get the value of one of a class's data members, you call a member function belonging to that class which returns the value of the desired member. To set the value of a field, you call a member function that assigns a new value to that field.

Member access control: **private**, **public**, and **protected**

While the enhanced **struct** in C++ allows bundling of data and functions, it is not as encapsulated or modular as it could be. As we mentioned earlier, access to all data members and member functions of a **struct** is **public** by default—that is, any statement within the same scope can read or change the internal data of a **struct** class. As noted earlier, this isn't desirable and can lead to serious problems. Good C++ design practices *data hiding* or *information hiding* — keeping member data private or protected, and providing an authorized interface for accessing it. The general rule is to make all data private so that it can be accessed only through public member functions. There are only a few situations where public rather than private or protected data members are needed. Also, some member functions involved only in internal operations can be made private or protected rather than public.

Three keywords provide access control to structure or class members. The appropriate keyword (with a colon) is placed before the member declarations to be affected:

- private:** Members following this keyword can be accessed only by member functions declared within the same class.
- protected:** Members following this keyword can be accessed by member functions within the same class, and by member functions of classes that are derived from this class (see the discussion on page 144).
- public:** Members following this keyword can be accessed from anywhere within the same scope as the class definition.

For example, here is how to redefine the **Point** structure so that the data members are private and the member functions are public:

```
struct Point {
private:
    int X;
    int Y;

public:
    int GetX();
    Point(int NewX, int NewY);
};
```

The **class: private** by default

A **struct** class is public by default, so you have to use **private:** to specify the private part, and then **public:** for the part to be made available for general access. Since good C++ practice makes things private by default and carefully specifies what should be public, C++ programmers generally favor the **class** over the **struct**. The only difference between a **class** and a **struct** is this matter of default privacy.

Point redefined as a class looks like this:

```
class Point {
    int X;        // private by default
    int Y;

public:          // needed to override the private default
    int GetX();
    Point(int NewX, int NewY);
};
```

No **private** modifier is needed for the data members—they're **private** by default. The member functions, however, must be

declared **public** so that they can be used outside of the class to initialize and retrieve values of **Point** objects.

You can repeat access control specifications as often as needed:

```
enum Boolean {false, true};

class Employee {
    double salary;           // private by default
    Boolean permanent;
    Boolean professional;

public:
    char name[50];
    char dept_code[3];

private:
    int Error_check(void);

public:
    Employee(double salary, Boolean permanent, Boolean professional,
             char *name, char *dept_code);
};
```

*Data members are usually **private**, while member functions are usually **public**. Allow public access only where it is truly needed.*

Here the data members *salary*, *permanent*, and *professional* are **private** by default; the data members *name* and *dept_code* are declared to be **public**; the member function **Error_check** is declared to be **private** (intended for internal use); and the constructor **Employee** is declared to be **public**.

Running a C++ program

It's time to put everything you've learned so far together into a complete compilable program. To compile a C++ program in the IDE, enter or load your text into the editor as usual. You can run C++ programs from the IDE in either of two ways. First, by default, any file with the .CPP extension will be compiled assuming C++ syntax, and any files with the .C extension will be compiled assuming C syntax. However, you can select the C++ Always button in the Source Options dialog box to have all files treated as C++ source files, regardless of extension.

To compile a C++ program with the command-line compiler, just give your file the extension .CPP. Or you can use the command-line option **-P**, in which case Turbo C++ will assume that the file has an extension of .CPP. If the file has a different extension, you must give the extension along with the file name. Life will be easier for you (and your next-of-kin) if you give all C++ programs a .CPP extension and all C programs a .C extension.

The program POINT.CPP defines the *Point* class and manipulates its data values:

This code is available to load and run: POINT.CPP.

```
/* POINT.CPP illustrates a simple Point class */
#include <iostream.h>           // needed for C++ I/O

class Point {                  // define Point class
    int X;                     // X and Y are private by default
    int Y;
public:
    Point(int InitX, int InitY) {X = InitX; Y = InitY;}
    int GetX() {return X;} // public member functions
    int GetY() {return Y;}
};

int main()
{
    int YourX, YourY;

    cout << "Set X coordinate: "; // screen prompt
    cin >> YourX;                 // keyboard input to YourX

    cout << "Set Y coordinate: "; // another prompt
    cin >> YourY;                 // key value for YourY

    Point YourPoint(YourX, YourY); // declaration calls constructor

    cout << "X is " << YourPoint.GetX(); // call member function
    cout << '\n';                     // newline
    cout << "Y is " << YourPoint.GetY(); // call member function
    cout << '\n';
    return 0;
}
```

The class **Point** now contains a new member function, **GetY**. This function works just like the **GetX** defined earlier, but accesses the private data member *Y* rather than *X*. Both are “short” functions and good candidates for the inline form of definition within the class body.

As with a macro using the **#define** directive, the code for an inline function is substituted directly into your file each time the function is used, thereby avoiding the function call overhead at the expense of code size. This is the classic “space versus time” dilemma found in many programming situations. As a general rule you should only use inline definitions for “short” functions, say one to three statements. Note that, unlike a macro, an inline function doesn’t sacrifice the type checking that helps prevent errors in function calls. The number of arguments in a function is also relevant to your decision whether to “inline” or not, since the

argument structure affects the function call overhead. The case for inlining is strongest when the total code for the function body is smaller than the code it takes to call the function out of line. You may need to try both methods and examine the assembly code output before deciding which approach is best for your needs.

Whether to inline a constructor or not can depend on whether base constructors are involved. A derived class constructor, especially where there are virtual functions (see page 160) in the hierarchy, can generate a lot of "hidden" code.

In the above example, the **Point** constructor has been defined as out-of-line, following the end of the class declaration. While you can put definitions in any order (and even put them elsewhere in the current file), it makes sense with smaller, single-file programs to put those definitions that aren't inline right after the class definition, in the order in which they were declared.

As your code gets larger, you'll probably have your class declarations in header files, and your class function definitions (implementation code) in separately compiled C++ source files. Inline function definitions, however, should always be in the header file.

This program also uses the C++ `iostreams` library introduced in chapter 3 (note the statement `#include <iostream.h>` at the beginning of the program).

Once the *X* and *Y* values have been received from the keyboard, the **Point** object *YourPoint* is declared with the received values as arguments. Recall that this declaration automatically invokes the constructor for the **Point** class, which creates and initializes *YourPoint*.

Try running the program. The result should look like this:

```
Set X coordinate: 50
Set Y coordinate: 100
X is 50.
Y is 100
```

Inheritance

Classes don't usually exist in a vacuum. A program often has to work with several different but related data structures. For exam-

ple, you might have a simple memory buffer in which you can store and from which you can retrieve data. Later, you may need to create more specialized buffers: A file buffer that holds data being moved to and from a file, and perhaps a buffer to hold data for a printer, and another to hold data coming from or going to a modem. These specialized buffers clearly have many characteristics in common, but each has some differences caused by the fact that disk files, printers, and modems involve devices that work differently.

The C++ solution to this “similar but different” situation is to allow classes to *inherit* characteristics and behavior from one or more *base* classes. This is an intuitive leap; inheritance is perhaps the single biggest difference between C++ and C. Classes that inherit from base classes are called *derived* classes. And a derived class may itself be the base class from which other classes are derived (recall the insect family tree).

Rethinking the Point class

The fundamental unit of graphics is the single point on the screen (one pixel). So far we’ve devised several variants of a **Point** class that define a point by its X and Y locations, a constructor that creates and initializes a point’s location, and other member functions that can return the point’s current X and Y coordinates. Before you can draw anything, however, you have to distinguish between pixels that are “on” (drawn in some visible color) and pixels that are “off” (have the background color). Later, of course, you may want to define which of many colors a given point should have, and perhaps other attributes (such as blinking). Pretty soon you can end up with a complicated class that has many data members.

Let’s rethink our strategy. What are the two fundamental *kinds* of information about points? One kind of information describes *where* the point is (location) and the other kind of information describes *how* the point is (the point’s state of being: You can either see it, or you can’t, and if you can see it, it is in some color). Of the two, the *location* is most fundamental: Without a location, you can’t have a point at all.

Because all points must contain a location, you can make the class **Point** a derived class of a more fundamental base class, **Location**, which contains the information about X and Y coordinates. **Point**

inherits everything that **Location** has, and adds whatever is new about **Point** to make **Point** what it must be.

These two related classes can be defined this way:

*This code is available as
point.h.*

```
/* point.h--Example from Object-oriented programming with C++ */
// point.h contains two classes:
// class Location describes screen locations in X and Y coordinates
// class Point describes whether a point is hidden or visible

enum Boolean {false, true};

class Location {
protected:      // allows derived class to access private data
    int X;
    int Y;

public:          // these functions can be accessed from outside
    Location(int InitX, int InitY);
    int GetX();
    int GetY();
};

class Point : public Location {      // derived from class Location
// public derivation means that X and Y are protected within Point
protected:
    Boolean Visible; // classes derived from Point will need access

public:
    Point(int InitX, int InitY);    // constructor
    void Show();
    void Hide();
    Boolean IsVisible();
    void MoveTo(int NewX, int NewY);
};
```

Here, **Location** is the base class, and **Point** is the derived class. The process can continue indefinitely: You can define other classes derived from **Location**, other classes derived from **Point**, yet more classes derived from **Point's** derived class, and so on. You can even have a class derived from more than one base class: This is called *multiple inheritance*, and will be discussed later. A large part of designing a C++ application lies in building this class hierarchy and expressing the family tree of the classes in the application.

Inheritance and
access control

Before we discuss the various member functions point.h, let's review the inheritance and access control mechanisms of C++.

The data members of the **Location** class are declared to be **protected**—recall that this means that member functions in both the **Location** class and the derived class **Point** will be able to access them, but the “public at large” won’t be able to do so.

You declare a derived class as follows:

```
class D : access_modifier B { // default is private
    ...
}
```

or

```
struct D : access_modifier B { // default is public
    ...
}
```

D is the name of the derived class, *access_modifier* is optional (either **public** or **private**), and **B** is the name of the base class.

With **class**, the default *access_modifier* is **private**; with **struct**, the default is **public**. (Note that **unions** can be neither base nor derived classes.)

The *access_modifier* is used to modify the accessibility of inherited members, as shown in the following table:

Table 4.1
Class access

*In a derived class, access to the elements of its base class can be made **more** restrictive but never **less** restrictive.*

Access in base class	Access modifier	Inherited access in base
public	public	public
private	public	not accessible
protected	public	protected
public	private	private
private	private	not accessible
protected	private	private

When writing new classes that rely on existing classes, make sure you understand the relationship between base and derived classes. A vital part of this is understanding the access levels conferred by the specifiers **private**, **protected**, and **public**. Access rights must be passed on carefully (or withheld) from parents to children to grandchildren. C++ lets you do this without “exposing” your data to non-family and non-friends. The access level of a base class member, as viewed by the base class, need not be the same as its access level as viewed by its derived class. In other words, when members are inherited, you have some control over how their access levels are inherited.

See Chapter 13 for more advanced technical details.

A class can be derived privately or publicly from its base class. **private** derivation (the default for **class** type classes) converts **public** and **protected** members in the base class into **private** members of the derived class, while **private** members remain **private**. (Although **private** derivation is the default for classes, it is by no means the most commonly used method of derivation—so we have a rare situation where the default is not the norm.)

A **public** derivation leaves the access level unchanged.

A derived class inherits all members of its base class, but can only use the **protected** and **public** members of its base class. **private** members of the base class are not directly available through the members of the derived class.

The particular definitions of **Location** and **Point** adopted here will allow us later on to derive further classes from **Point** for more complex graphics applications.

*Base class members that you want to use in a derived class must be either **protected** or **public**. **private** base class members can't be accessed except by their own member functions or through **friend** functions.*

If you use **public** derivation, **protected** members of the base class remain **protected** in the derived class, and thus won't be available from outside except to other publicly derived classes and friends. It's a good idea to always specify **public** or **private**, whatever the default, to avoid confusion. Good comments, too, will improve your source code legibility.

Packaging classes into modules

Classes such as **Location** and **Point** can be packaged together for use in further program development. With its built-in data, member functions, and access control, a class is inherently modular. In developing a program, it often makes sense to put the declarations for each class or group of related classes in a separate header file, and the definitions for its non-inline member functions in a separate source file.

You can also combine several class object files into a library using TLIB. (See the online document UTIL.DOC to learn how to create libraries.)

There are further advantages to modularizing classes: You can distribute your classes in object form to other programmers. The other programmers can derive new, specialized classes from the ones you made available, without needing access to your source code.

We can now develop a separately compiled "module" containing the **Location** and **Point** classes. First, the declarations for the two classes (including their member functions) as listed on page 146 are put in the file `point.h` (on your distribution diskettes).

Note again how the class **Point** is derived from the class **Location**:

```
class Point : public Location { ...
```

The keyword **public** is needed before **Location** to ensure that the member functions of the derived class, **Point**, can access the protected members *X* and *Y* in the base class, **Location**. In addition to the *X* and *Y* location members, **Point** inherits the member functions **GetX** and **GetY** from **Location**. The class **Point** also adds the **protected** data member *Visible* (of the enumerated type *Boolean*), and five public member functions, including the constructor **Point::Point**. Note again that we have used **protected** rather than **private** access for certain elements so that `point.h` can be used in later examples that have further classes derived from **Location** and **Point**.

The file `POINT2.CPP` contains the definitions for all of the member functions of these two classes:

*This code is available as
POINT2.CPP.*

```
/* POINT2.CPP--Example from Object-oriented programming with C++ */
// POINT2.CPP contains the definitions for the Point and Location
// classes that are declared in the file point.h

#include "point.h"
#include <graphics.h>

// member functions for the Location class
Location::Location(int InitX, int InitY) {
    X = InitX;
    Y = InitY;
};

int Location::GetX(void) {
    return X;
};

int Location::GetY(void) {
    return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Point::Point(int InitX, int InitY) : Location(InitX,InitY) {
    Visible = false;           // make invisible by default
};
```



```

void Point::Show(void) {
    Visible = true;
    putpixel(X, Y, getcolor());    // uses default color
};

void Point::Hide(void) {
    Visible = false;
    putpixel(X, Y, getbkcolor()); // uses background color to erase
};

Boolean Point::IsVisible(void) {
    return Visible;
};

void Point::MoveTo(int NewX, int NewY) {
    Hide();    // make current point invisible
    X = NewX;    // change X and Y coordinates to new location
    Y = NewY;
    Show();    // show point at new location
};

```

A base constructor is invoked before the body of the derived class constructor.

This example introduces the important concept of base-class constructors. When a **Point** object is defined, we want to make use of the fact that its base class, **Location**, already has its own user-defined constructor. The definition of the constructor **Point::Point** begins with a colon and a reference to the base constructor *Location(InitX,InitY)*. This specifies that the **Point** constructor will first call the **Location** constructor with the arguments *InitX* and *InitY*, thereby creating and initializing data members *X* and *Y*. Then the **Point** constructor body is invoked, creating and initializing the data member *Visible*. By explicitly specifying a base constructor, we have saved ourselves some coding (in larger examples, of course, the savings may be more significant).

In fact, derived-class constructors *always* call a constructor of the base class first to ensure that inherited data members are correctly created and initialized. If the base class is itself derived, the process of calling base constructors continues down the hierarchy. If you don't define a constructor for a particular class **X**, C++ will generate a default constructor of the form **X::X()**; that is, a constructor with no arguments.

If the derived-class constructor does not explicitly invoke one of its base-class constructors, or if you have not defined a base-class constructor, the default base class constructor (with no arguments) will be invoked. (There's more on base class constructors in chapter 13, "C++ specifics.")

Notice that the reference to the base class constructor, *Location(InitX,InitY)* appears in the definition, not the declaration, of the derived class constructor.

Here's a main program (available on your distribution disks as *PIXEL.CPP*) that demonstrates the capabilities of the **Point** and **Location** classes.

You'll need to compile and link *POINT2.CPP*, *PIXEL.CPP*, and *GRAPHICS.LIB*, using the *PIXEL.PRJ* project file supplied on your distribution diskettes. (Read Chapter 7, "Managing multi-file projects," for information on how to use project files.)

```
/* PIXEL.CPP--Example from Chapter 5 of Getting Started */
// PIXEL.CPP demonstrates the Point and Location classes
// compile with POINT2.CPP and link with GRAPHICS.LIB
#include <graphics.h> // declarations for graphics library
#include <conio.h> // for getch() function
#include "point.h" // declarations for Point and Location

int main()
{
    // initialize the graphics system
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "c:\\\\bgi");

    // move a point across the screen
    Point APoint(100, 50); // Initial X, Y at 100, 50
    APoint.Show(); // APoint turns itself on
    getch(); // Wait for keypress
    APoint.MoveTo(300, 150); // APoint moves to 300,150
    getch(); // Wait for keypress
    APoint.Hide(); // APoint turns itself off
    getch(); // Wait for keypress
    closegraph(); // Restore original screen
    return 0;
}
```

Extending classes

One of the beauties of classes is the way that new objects can be accommodated and given appropriate functionality. The next example takes the already defined **Location** and **Point** classes and derives a new class, **Circle**, along with functions to show, hide, expand, move, and contract circles.

This code is on your disks:
CIRCLE.CPP.

```
/* CIRCLE.CPP--Example from Object-oriented programming with C++ */
// CIRCLE.CPP A Circle class derived from Point
#include <graphics.h> // graphics library declarations
#include "point.h" // Location and Point class declarations
#include <conio.h> // for getch() function
```

```

// link with point2.obj and graphics.lib
class Circle : Point { // derived privately from class Point
                        // and ultimately from class Location
    int Radius;        // private by default

public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show(void);
    void Hide(void);
    void Expand(int ExpandBy);
    void MoveTo(int NewX, int NewY);
    void Contract(int ContractBy);
};

Circle::Circle(int InitX, int InitY, int InitRadius) :
Point(InitX, InitY)
{
    Radius = InitRadius;
};

void Circle::Show(void)
{
    Visible = true;
    circle(X, Y, Radius); // draw the circle
}

void Circle::Hide(void)
{
    unsigned int TempColor; // to save current color
    TempColor = getcolor(); // set to current color
    setcolor(getbkcolor()); // set drawing color to background
    Visible = false;
    circle(X, Y, Radius); // draw in background color to erase
    setcolor(TempColor); // set color back to current color
};

void Circle::Expand(int ExpandBy)
{
    Hide(); // erase old circle
    Radius += ExpandBy; // expand radius
    if (Radius < 0) // avoid negative radius
        Radius = 0;
    Show(); // draw new circle
};

void Circle::Contract(int ContractBy)
{
    Expand(-ContractBy); // redraws with (Radius - ContractBy)
};

void Circle::MoveTo(int NewX, int NewY)
{

```

```

Hide(); // erase old circle
X = NewX; // set new location
Y = NewY;
Show(); // draw in new location
};

main() // test the functions
{
    // initialize the graphics system
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "c:\\\\bgi");

    Circle MyCircle(100, 200, 50); // declare a circle object
    MyCircle.Show(); // show it
    getch(); // wait for keypress
    MyCircle.MoveTo(200, 250); // move the circle (tests hide
    // and show also)

    getch();
    MyCircle.Expand(50); // make it bigger
    getch();
    MyCircle.Contract(75); // make it smaller
    getch();
    closegraph();
    return 0;
}

```

To see how this works for the **Circle** class, you need to examine the member functions in the listing **CIRCLE.CPP** and refresh yourself on the class declarations in **point.h**.

Note first that the member functions of **Circle** need to access various data members in the classes **Circle**, **Point**, and **Location**.

Consider **Circle::Expand**. It needs access to **int Radius**. No problem. *Radius* is defined as **private** (by default) in **Circle** itself. So, *Radius* is accessible to **Circle::Expand**—indeed, it is accessible *only* to member functions of **Circle**. (Later, you'll see that the **private** members of a class can also be accessed by functions that have been specially defined as **friends** of that class.)

Next, look at the member function **Circle::Hide**. This needs to access **Boolean Visible** from its base class **Point**. Now *Visible* is protected in **Point**, and **Circle** is derived privately (by default) from **Point**. So, from the rules outlined above, *Visible* is **private within Circle**, and is accessible just like *Radius*. Note that if *Visible* had been defined as **private** in **Point**, it would have been inaccessible to the member functions of **Circle**. So, you might be tempted to make *Visible* **public**. However, this is overkill: *Visible* would become accessible to non-member functions. You might say that

protected is **private** with a dash of **public** for derived classes: member functions of a derived class can access a **protected** member without exposing that member to public abuse.

Finally, consider **Circle::Show**. **Circle::Show** needs to access **Location**'s members *X* and *Y* in order to draw the circle. How is this achieved? **Circle** is not directly derived from **Location**, so the access rights are not immediately obvious. **Circle** derives from **Point** which derives from **Location**. Let's trace the access declarations.

1. Members *X* and *Y* are declared **protected** in **Location**.
2. **Point** specifies **public** derivation from **Location**, so **Point** also inherits the *X* and *Y* members as **protected**.
3. **Circle** is derived from **Point** using the default **private** derivation.
4. **Circle** therefore inherits *X* and *Y* as **private**. **Circle::Show** can access *X* and *Y*. Note that *X* and *Y* are still **protected** within **Location**.

Having digested this chain of access rights, you might want to consider the situation if a derived class of **Circle**, such as **PieChart** or **Arc**, was needed. Yes, you would need to change the derivation of **Circle** from **Point**—it would need to be a **public** derivation and *Radius* would need to become **protected**.

It should now be pretty easy to see what is going on in **CIRCLE.CPP**. A circle, in a sense, is a fat point: It has everything a point has (an *X,Y* location and a visible/invisible state) plus a radius. Class **Circle** appears to have only the single member *Radius*, but don't forget about all the members that **Circle** inherits by being a derived class of **Point**. **Circle** has *X*, *Y*, and *Visible* as well, even if you don't see them in the class definition for **Circle**.

Compile and link **CIRCLE.CPP**, **POINT2.CPP**, and **GRAPHICS.LIB**. The project file **CIRCLE.PRJ** on your distribution diskettes will help you do this. As you press a key, you should see a circle. Press a key again and the circle moves. Again, and the circle expands, and again and the circle contracts.

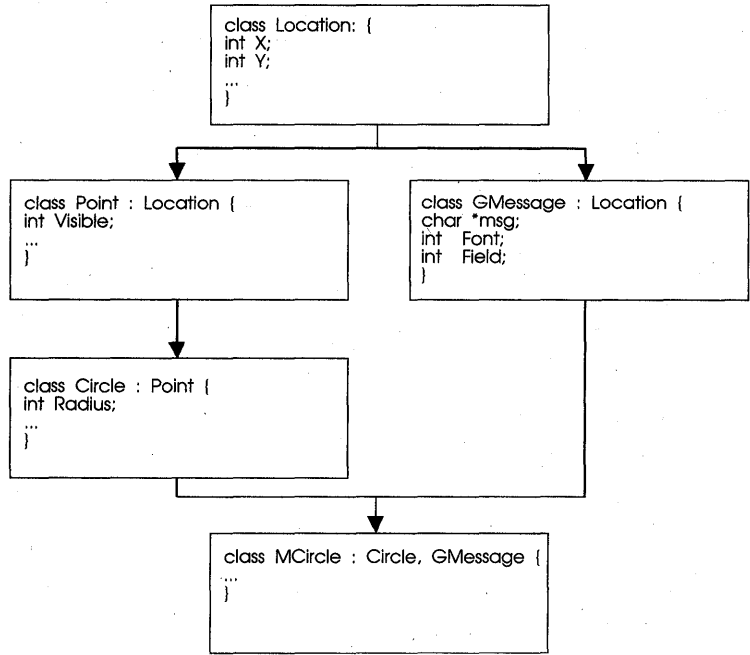
Multiple inheritance

As we mentioned earlier, a class can inherit from more than one base class. This *multiple inheritance* mechanism was one of the main features added to C++ release 2.0. To see a practical example, the next program lets you display text inside a circle.

Your first thought might be to simply add a string data member to the *Circle* class and then add code to **Circle::Show** so that it displays the text with the circle drawn around it. But text and circles are really quite different things: When you think of text you think of fonts, character size, and possibly other attributes, none of which really has anything to do with circles. You could, of course, derive a new class directly from *Circle* and give it text capabilities. When dealing with fundamentally different functionalities, however, it is often better to create new “fundamental” base classes, and then derive specialized classes that combine the appropriate features. The next listing, *MCIRCLE.CPP*, illustrates this approach.

We’ll define a new class called *GMessage* that displays a string on the screen starting at specified *X* and *Y* coordinates. This class will be *MCircle*’s other parent. *MCircle* will inherit **GMessage::Show** and use it to draw the text. The relationships of all of the classes involved is shown in the next figure.

Figure 4.3
Multiple inheritance



This code is available on your disks: MCIRCLE.CPP. You need to run it using MCIRCLE.PRJ.

```

/* MCIRCLE.CPP--Example for Object-oriented programming with C++ */
// MCIRCLE.CPP      Illustrates multiple inheritance

#include <graphics.h> // Graphics library declarations
#include "point.h"    // Location and Point class declarations
#include <string.h>   // for string functions
#include <conio.h>    // for console I/O

// link with point2.obj and graphics.lib

// The class hierarchy:
// Location->Point->Circle
// (Circle and GMessage)->MCircle

class Circle : public Point { // Derived from class Point and
                             // ultimately from class Location
protected:
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show(void);
};

class GMessage : public Location {

```

```

// display a message on graphics screen
char *msg;           // message to be displayed
int Font;           // BGI font to use
int Field;          // size of field for text scaling

public:
    // Initialize message
    GMessage(int msgX, int msgY, int MsgFont, int FieldSize,
             char *text);
    void Show(void); // show message
};

class MCircle : Circle, GMessage { // inherits from both classes
public:
    MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
            char *msg);
    void Show(void); // show circle with message
};

// Member functions for Circle class
// Circle constructor
Circle::Circle(int InitX, int InitY, int InitRadius) :
    Point(InitX, InitY) // initialize inherited members
// also invokes Location constructor
{
    Radius = InitRadius;
};

void Circle::Show(void)
{
    Visible = true;
    circle(X, Y, Radius); // draw the circle
}

// Member functions for GMessage class
// GMessage constructor
GMessage::GMessage(int msgX, int msgY, int MsgFont,
                  int FieldSize, char *text) :
    Location(msgX, msgY)
// X and Y coordinates for centering message
{
    Font = MsgFont; // standard fonts defined in graph.h
    Field = FieldSize; // width of area in which to fit text
    msg = text; // point at message
};

void GMessage::Show(void)
{
    int size = Field / (8 * strlen(msg)); // 8 pixels per char.

```



```

        settextrjust(CENTER_TEXT, CENTER_TEXT); // centers in circle
        settextrstyle(Font, HORIZ_DIR, size); // if size > 1, magnifies
        outtextxy(X, Y, msg); // display the text
    }

    // Member functions for MCircle class

    // MCircle constructor
    MCircle::MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
                    char *msg) : Circle (mcircX, mcircY, mcircRadius),
                               GMessage(mcircX,mcircY,Font,2*mcircRadius,msg)
    {
    }

    void MCircle::Show(void)
    {
        Circle::Show();
        GMessage::Show();
    }

    main() // draws some circles with text
    {
        int graphdriver = DETECT, graphmode;
        initgraph(&graphdriver, &graphmode, "c:\\bgi");
        MCircle Small(250, 100, 25, SANS_SERIF_FONT, "You");
        Small.Show();
        MCircle Medium(250, 150, 100, TRIPLEX_FONT, "World");
        Medium.Show();
        MCircle Large(250, 250, 225, GOTHIC_FONT, "Universe");
        Large.Show();
        getch();
        closegraph();
        return 0;
    }

```

As you read the listing, check the class declarations and note which data members and member functions are inherited by each class. You may also want to look at `point.h` again, since the *Location* and *Point* classes are defined there. Notice that both *MCircle* and *GMessage* have *Location* as their ultimate base class: *MCircle* by way of *Point* and *Circle*, and *GMessage* directly.

The :: operator is used to specify a function from another scope rather than (by default) using the function of that name in the current scope.

In the body of the definition of **MCircle::Show**, you will see the two function calls **Circle::Show()**; and **GMessage::Show()**; This syntax shows another common use of **::** (the scope resolution operator). When you want to call an inherited function, such as **Show**, the compiler may need some help: which **Show** is required? Without the scope resolution "override," **Show()** would refer to the **Show()** in the current scope, namely **MCircle::Show()**. To call the **Show()** of another scope (assuming, of course, that you

have access permission), you must supply the appropriate class name followed by `::` and the function name (with arguments, if any). What if there happened to be a *nonmember* function called **Show** that you wanted to call? You would use `::Show()` with no preceding class name.

A member function of a given name in the derived class *overrides* the member function of the same name in the base class, but you can still get at the latter by using `::`. The scoping rules for C++ are slightly different from those for C.

Before leaving MCIRCLE.CPP, a brief word about the constructor for **MCircle**. You saw earlier how the **Point** constructor explicitly invoked its base constructor in **Location**. Since **MCircle** inherits from *both* **Circle** and **GMessage**, the **MCircle** constructor can conveniently initialize by calling *both* base constructors:

```
MCircle::MCircle
    (int mcircX, int mcircY, int mcircRadius, int font, char *msg) :
    Circle(mcircX, mcircY, mcircRadius),
    GMessage(mcircX, mcircY, 2*mcircRadius,msg) {
}
```

The constructor body is empty here because all the necessary work is accomplished in the *member initialization list* (after the `:` you enter a list of initializing expressions separated by commas. You met a simpler version of this syntax in the single base class constructors used in the **Point** and **Circle** class definitions). When the **MCircle** constructor is invoked (by declaring an **MCircle** object, for example), quite a spate of activity is triggered behind the scenes.

See Chapter 13 for details on
constructor calling
sequences.

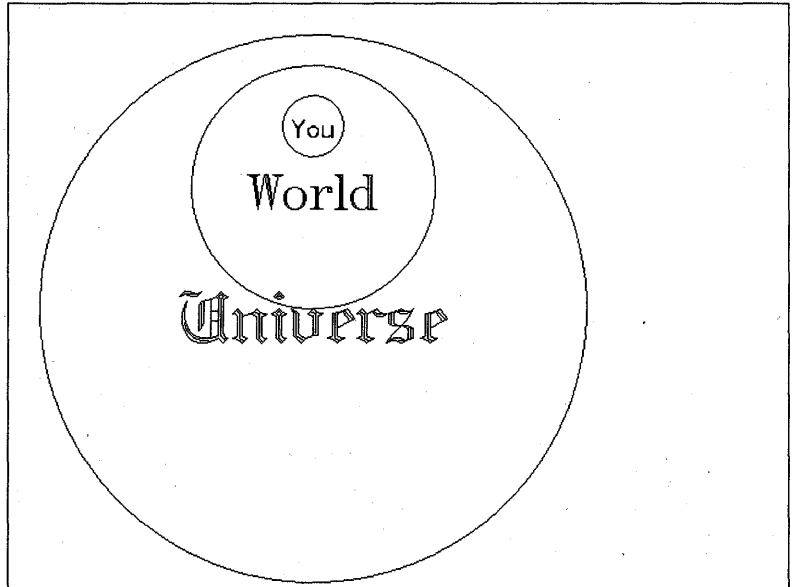
First, the **Circle** constructor is called. This constructor then calls the **Point** constructor, which in turn calls the **Location** constructor. Finally, the **GMessage** constructor is called, which calls the **Location** constructor for its own copy of its base class X and Y. The arguments given in the **MCircle** constructor are passed on to initialize the appropriate data members of the base classes.

When destructors are called (when an object goes out of scope, for example), the deallocation sequence is the reverse of that used during construction. (Virtual base class constructors and destructors have some sequencing quirks beyond the scope of this chapter.)

In passing, recall the point made earlier: if you don't supply your own constructors or destructors, C++ will generate and invoke default versions behind the scenes.

Figure 4.4 shows the output of MCIRCLE:

Figure 4.4
Circles with messages



virtual functions

Each class type in our graphics hierarchy represents a different type of figure onscreen: a point or a circle. It certainly makes sense to say that you can show a point on the screen, or show a circle. Later on, if you were to define classes to represent other figures such as lines, squares, arcs, and so on, you could write a member function for each that would display that object onscreen. In the new way of object-oriented thinking, you could say that all these graphic figure types had the ability to show themselves on the screen.

What is different for each object type is the *way* it must show itself onscreen. A point is drawn with a point-plotting routine that needs only an X, Y location and perhaps a color value. A circle needs a more complex graphics routine to display itself, taking into account not only X and Y , but a radius as well. Still further, an arc needs a start angle and an end angle, and a different

drawing algorithm. The same situation, of course, applies to hiding, dragging, and other basic shape manipulations.

The ordinary member functions you have seen so far certainly allow us to define a **Show** function for each shape class. But they lack an essential ingredient. Graphics modules based on our existing classes and member functions would need source code changes and recompilations each time a new shape class was introduced with its own member function **Show**. The reason is that the C++ mechanisms revealed so far allow essentially only three ways to resolve the question: which **Show** is being referenced?:

1. There's the distinction by argument signature—**Show(int,char)** is not the same function as **Show(char*,float)**, for example.
2. There's the use of the scope resolution operator, whereby **Circle::Show** is distinguished from **Point::Show** and **::Show**.
3. There's the resolution by class object: *ACircle.Show* invokes **Circle::Show**, while *Apoint.Show* invokes **Point::Show**. Similarly with pointers to objects: *APoint_pointer->Show* invokes **Point::Show**.

All these function resolutions, so far, have been made at compile time—a mechanism called *early* or *static* binding.

A typical graphics toolbox would provide the user with class definitions in .H source files together with the precompiled .OBJ or .LIB code for the member functions. With the early binding restrictions, the user cannot easily add new class shapes, and even the developer faces extra chores in extending the package. C++ offers a flexible mechanism to solve these problems: *late* (or *dynamic*) binding by means of special member functions called *virtual* functions.

The key concept is that virtual function calls are resolved at run time (hence the term, late binding). In practical terms, it means that the decision as to which **Show** function is called can be deferred until the object type involved is known during execution. A virtual function **Show**, "hidden" in a class **B** in the precompiled toolbox library, is not *bound* to the objects of **B** in the way that ordinary member functions of **B** are. You are free to create a class **D** derived from **B** for your own favorite shape, and write appropriate functions (putting on your **Show**, as it were). You then compile and link your OBJ or LIB code to that of the toolbox. Calls made on **Show**, whether from existing member functions of

B or from the new functions you have written for **D**, will automatically reference the correct **Show**. This resolution is made entirely on the object type involved in the call. Let's look at virtual functions in action. We have a potential candidate in the earlier code given for **CIRCLE.CPP**

virtual functions in action

Consider the member function **Circle::MoveTo** in **CIRCLE.CPP**:

```
void Circle::MoveTo(int NewX, int NewY)
{
    Boolean vis = Visible;
    if (vis) Hide(); // hide only if visible
    X = NewX; Y = NewY; // set new location
    if (vis) Show(); // draw at new location if previously
                    // visible
}
```

Notice how similar this definition is to **Point::MoveTo** found in the **Circle's** base class **Point**. In fact, the return value, function name, number and types of formal arguments (known as the function *signature*), and even the function body itself, all appear to be identical! If C++ encounters two function calls using the same function name but differing in signatures, we have already seen that the C++ compiler is smart enough to resolve the potential ambiguities caused by function-name *overloading*. (Recall that C, unlike C++, demands unique function names.) In C++, member functions with different signatures are really different functions, even if they share the same name.

But, our two **MoveTos** do not, at first sight, offer any distinguishing clues to the compiler—so will it know which one you intended to call? The answer, as you've seen, with ordinary member functions is that the compiler determines the target function from the class type of the object involved in the call.

So, why not let **Circle** inherit **Point's MoveTo**, just as **Circle** inherits **Point's GetX** and **GetY** (via **Location**)? The reason, of course, is that the **Hide** and **Show** called in **Circle::MoveTo** are not the same **Hide** and **Show** called in **Point::MoveTo**. Only the names and signatures are the same. Inheriting **MoveTo** from **Point** would lead to the wrong **Hide** and **Show** being called when trying to move a circle. Why? Because **Point's** versions of these two functions would be bound to **Point's** (and hence also to **Circle's**) **MoveTo** at compile time (early binding). As you may have

guessed already, the answer is to declare **Hide** and **Show** as virtual functions. This will delay the binding so that the correct versions **Hide** and **Show** can be invoked when **MoveTo** is actually called to move a point or a circle (or whatever).

Note again that if we wanted to precompile our class definitions and member functions for **Location**, **Point**, and **Circle** in a neat standalone library (with the implementation source locked up with our other trade secrets), we certainly could not know in advance the objects that **MoveTo** may be asked to move. Virtual functions not only provide this technical advantage; they also provide a conceptual gain that lies at the heart of OOP. We can concentrate on developing reusable classes and methods with less anxiety about name clashes.

While it is true that add-on library extensions are available for most languages, the use of virtual functions and multiple inheritance in C++ makes extensibility more natural. You inherit everything that all your base classes have, and then you add the new capabilities you need to make new objects work in familiar ways. The classes you define and their versions of the virtual functions become a true extension of an orderly hierarchy of capabilities. Because this is part of the language design rather than an afterthought, there is very little penalty in performance.

Having sold you on the merits of virtual functions, let's see how you can implement them, and some of the rules you have to follow.

Defining **virtual** functions

The syntax is straightforward: add the qualifier **virtual** in the member function's first declaration:

```
virtual void Show();  
virtual void Hide();
```



Important! Only member functions can be declared as **virtual**. Once a function is declared **virtual**, it must not be redeclared in any derived class with the *same* formal argument signature but with a *different* return type. If you redeclare **Show** with the same formal argument signature and same return type, the new **Show** automatically becomes virtual, whether you use the **virtual** qualifier or not. This new, virtual **Show** is said to override the **Show** in its base class.

You are free to redeclare **Show** with a different formal argument signature (whether you change the return type or not)—but the virtual mechanism is inoperable for this version of **Show**. Beginners should avoid rash overloading—there are situations where a non-virtual function can hide a virtual function declared in its base.

The particular **Show** called will depend only on the class of the object for which **Show** is invoked, even if the call is invoked via a pointer (or reference) to the base class. For example,

```
Circle ACircle;
Point* APoint_pointer = &ACircle; // pointer to Circle assigned to
                                   // pointer to base class, Point
APoint_pointer->Show();           // calls Circle::Show!
```

vpoint.h and VCIRC.CPP (available on your distribution disks) are versions of point.h and CIRCLE.CPP with **Show** and **Hide** made virtual. Compile VCIRC.CPP with POINT2.CPP using VCIRC.PRJ. It will run exactly like CIRCLE.CPP. We don't list the virtual versions in full here since the differences can be summed up simply as follows:

- In vpoint.h, **Point's Show** and **Hide** have been declared with the keyword **virtual**. The **Show** and **Hide** in the VCIRC's derived class **Circle** have the same argument signature and return values as the base versions in **Point**; this implies that they are also virtual, even though the keyword **virtual** is not used in their declarations.
- In VCIRC.CPP, **Circle** no longer has its own **MoveTo** member function.
- We now derive **Circle** publicly from **Point** to allow access to **MoveTo**

To recap the significance of these changes:

Circle objects can now safely call the **MoveTo** inherited from **Point**. The **Show** and **Hide** called by **MoveTo** will be bound at run time to **Circle's** own **Show** and **Hide**. Any **Point** objects calling **MoveTo** will invoke the **Point** versions.

Developing a complete graphics module

As a more complete and realistic example of virtual functions, let's create a module that defines some shape classes and a generalized means of dragging them around the screen. This module,

figures.h and FIGURES.CPP (on your distribution diskettes), is a simple implementation of the graphics toolbox discussed earlier.

A major goal in designing the FIGURES module is to allow users of the module to extend the classes defined in the module—and still make use of all the module's features. It is an interesting challenge to create some means of dragging an arbitrary graphics figure around the screen in response to user input.

As a first approach, we might consider a function that takes an object as an argument, and then drags that object around the screen:

```
void Drag(Point& AnyFigure, int DragBy)
{
    int DeltaX,DeltaY;
    int FigureX,FigureY;
    AnyFigure.Show();           // Display figure to be dragged
    FigureX = AnyFigure.GetX(); // Get the initial X,Y of figure
    FigureY = AnyFigure.GetY();

    // This is the drag loop
    while (GetDelta(DeltaX, DeltaY))
    {
        // Apply delta to figure X,Y
        FigureX = FigureX + (DeltaX * DragBy);
        FigureY = FigureY + (DeltaY * DragBy);
        // And tell the figure to move
        AnyFigure.MoveTo(FigureX, FigureY);
    }
};
```

Reference types Notice that *AnyFigure* is declared to be of type **Point&**. This means “a reference to an object of type **Point**” and is a new feature of C++. As you know, C ordinarily passes arguments by value, not by reference. In C, if you want to act directly on a variable being passed to a function, you have to pass a pointer to the variable, which can lead to awkward syntax, since you have to remember to dereference the pointer. C++ lets you pass and modify the actual variable by using a reference. To declare a reference, simply follow the data type with an ampersand (&) in the variable declaration.

Drag calls an auxiliary function not shown here, **GetDelta**, that obtains some sort of change in X and Y from the user. It could be from the keyboard, or from a mouse, or a joystick. (For

simplicity's sake, our example obtains input from the arrow keys on the keyboard.)

An important point to notice about **Drag** is that any object of type **Point**, or any type derived from **Point**, can be passed in the *AnyFigure* reference argument. Objects of **Point** or **Circle** type, or any type defined in the future that inherits from **Point** or **Circle**, can be passed without complication in *AnyFigure*.

Adding a new member function to an existing class hierarchy involves a little thought. How far up the hierarchy should the member function be placed? Think about the utility provided by the function and decide how broadly applicable that utility is. Dragging a figure involves changing the location of the figure in response to input from the user. In terms of inheritability, it sits right beside **MoveTo**—any object to which **MoveTo** is appropriate should also inherit **Drag**. Therefore **Drag** should be a member of **Point**, so that all of **Point**'s derived types can share it.

Having resolved the place of **Drag** in the hierarchy, we can take a closer look at its definition. As a member function of the base class **Point**, there is no need for the explicit reference to the *Point* & *AnyFigure* argument. We can rewrite **Drag** so that the functions it calls, such as **GetX**, **Show**, **MoveTo**, and **Hide**, will correctly reference the versions appropriate to the type of the object being dragged. As we saw earlier, the functions **Show** and **Hide** that require special shape-related code can be made virtual. We can then redefine them for any future classes without disturbing the **FIGURES** module. This also takes care of **MoveTo**, since **MoveTo** calls the correct **Show** and **Hide** (you'll recall that that was our original motivation for making **Show** and **Hide** virtual). **GetX** and **GetY** present no problem: as ordinary member functions inherited from **Point** via **Location**, they simply return the X and Y data members of the calling object of any derived class, present or future. Remember, though, that X and Y are protected in **Location**, so we must use public derivation as shown.

The next design decision is whether to make **Drag** virtual. The litmus test for making any function virtual is whether its functionality is expected to change somewhere down the hierarchy. There is no golden rule here, but later on we'll discuss the various tradeoffs: extensibility versus performance overhead (virtual functions require slightly more memory and a few more memory-access cycles). We have taken the view that some future class in, say, a CAD (Computer Aided Design) application might conceivably need a special dragging action. Perhaps dragging an

isometric drawing will require some scaling actions, and so on. In our new **Point** class definition in `figures.h`, we have therefore made **Drag** virtual.

Remember to recompile everything that uses this header file.

```
class Point : public Location {
protected:
    Boolean Visible;
public:
    Point(int InitX, int InitY);
    virtual void Show();          // Show and Hide are virtual
    virtual void Hide();
    Boolean IsVisible() {return Visible;}
    void MoveTo(int NewX, int NewY);
    virtual void Drag(int DragBy);
};
```

Here is the header file `figures.h` containing the class declarations for the FIGURES module. This is the only part of the package that needs to be distributed in source code form:

*This code is on your disks:
figures.h.*

```
// figures.h contains three classes.
//
// Class Location describes screen locations in X and Y
// coordinates.
//
// Class Point describes whether a point is hidden or visible.
//
// Class Circle describes the radius of a circle around a point.
//
// To use this module, put #include <figures.h> in your main
// source file and compile the source file FIGURES.CPP together
// with your main source file.

enum Boolean {false, true};

class Location {
protected:
    int X;
    int Y;
public:
    Location(int InitX, int InitY) {X = InitX; Y = InitY;}
    int GetX() {return X;}
    int GetY() {return Y;}
};

class Point : public Location {
protected:
    Boolean Visible;
public:
    Point(int InitX, int InitY);
```

```

virtual void Show();          // Show and Hide are virtual
virtual void Hide();
virtual void Drag(int DragBy); // new virtual drag function
Boolean IsVisible() {return Visible;}
void MoveTo(int NewX, int NewY);
};

class Circle : public Point { // Derived from class Point and
                             // ultimately from class Location
protected:
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show();
    void Hide();
    void Expand(int ExpandBy);
    void Contract(int ContractBy);
};

// prototype of general-purpose, non-member function
// defined in FIGURES.CPP

Boolean GetDelta(int& DeltaX, int& DeltaY);

```

Here is the file FIGURES.CPP containing the member function definitions. This is what would be distributed in object or library form commercially. Note that we have defined the **Circle** constructor outside the class since it invokes base constructors. You may wish to experiment by making it an inline function (see the discussion on page 179). The nonmember function **GetDelta** will repay some study if you are new to C. Note the use of reference arguments, which is a C++ touch; the rest of the code is traditional.

This code is on your disks: FIGURES.CPP. You should compile this code and link it to GRAPHICS.LIB to get FIGURES.OBJ. You'll need FIGURES.OBJ for the next exercise.

```

// FIGURES.CPP: This file contains the definitions for the Point
// class (declared in figures.h). Member functions for the
// Location class appear as inline functions in figures.h.

#include "figures.h"
#include <graphics.h>
#include <conio.h>

// member functions for the Point class

//constructor
Point::Point(int InitX, int InitY) : Location (InitX, InitY)
{
    Visible = false;    // make invisible by default
}

void Point::Show()
{

```

```

    Visible = true;
    putpixel(X, Y, getcolor()); // uses default color
}

void Point::Hide()
{
    Visible = false;
    putpixel(X, Y, getbkcolor()); // uses background color to erase
}

void Point::MoveTo(int NewX, int NewY)
{
    Hide();           // make current point invisible
    X = NewX;        // change X and Y coordinates to new location
    Y = NewY;
    Show();          // show point at new location
}

// a general-purpose function for getting keyboard
// cursor movement keys (not a member function)
Boolean GetDelta(int& DeltaX, int& DeltaY)
{
    char KeyChar;
    Boolean Quit;
    DeltaX = 0;
    DeltaY = 0;

    do
    {
        KeyChar = getch(); // read the keystroke
        if (KeyChar == 13) // carriage return
            return(false);
        if (KeyChar == 0) // an extended keycode
        {
            Quit = true; // assume it is usable
            KeyChar = getch(); // get rest of keycode
            switch (KeyChar) {
                case 72: DeltaY = -1; break; // down arrow
                case 80: DeltaY = 1; break; // up arrow
                case 75: DeltaX = -1; break; // left arrow
                case 77: DeltaX = 1; break; // right arrow
                default: Quit = false; // bad key
            };
        };
    } while (!Quit);
    return(true);
}

void Point::Drag(int DragBy)
{
    int DeltaX, DeltaY;

```

```

int FigureX, FigureY;

Show();           // display figure to be dragged
FigureX = GetX(); // get initial position of figure
FigureY = GetY();

// This is the drag loop
while (GetDelta(DeltaX, DeltaY))
{
    // Apply delta to figure at X, Y
    FigureX += (DeltaX * DragBy);
    FigureY += (DeltaY * DragBy);
    MoveTo(FigureX, FigureY); // tell figure to move
};
}
// Member functions for the Circle class
//constructor
Circle::Circle(int InitX, int InitY, int InitRadius) : Point (InitX,
InitY)
{
    Radius = InitRadius;
}

void Circle::Show()
{
    Visible = true;
    circle(X, Y, Radius); // draw the circle
}

void Circle::Hide()
{
    unsigned int TempColor; // to save current color
    TempColor = getcolor(); // set to current color
    setcolor(getbkcolor()); // set drawing color to background
    Visible = false;
    circle(X, Y, Radius); // draw in background color to
    setcolor(TempColor); // set color back to current color
}

void Circle::Expand(int ExpandBy)
{
    Hide(); // erase old circle
    Radius += ExpandBy; // expand radius
    if (Radius < 0) // avoid negative radius
        Radius = 0;
    Show(); // draw new circle
}

void Circle::Contract(int ContractBy)
{

```

```

        Expand(-ContractBy);    // redraws with (Radius-ContractBy)
    }

```

We are now ready to test FIGURES by exposing it to a new shape class called **Arc** that is defined in FIGDEMO.CPP. **Arc** is (naturally) derived publicly from **Circle**. Recall that **Drag** is about to drag a shape it has never seen before!

This code is on your disks as FIGDEMO.CPP. You need to compile it and link it to FIGURES.OBJ.

```

// FIGDEMO.CPP -- Exercise for Object-oriented programming with C++
// demonstrates the Figures toolbox by extending it with
// a new type Arc.

// Link with FIGURES.OBJ and GRAPHICS.LIB

#include "figures.h"
#include <graphics.h>
#include <conio.h>

class Arc : public Circle {
    int StartAngle;
    int EndAngle;
public:
    // constructor
    Arc(int InitX, int InitY, int InitRadius, int InitStartAngle, int
        InitEndAngle) : Circle (InitX, InitY, InitRadius) {
        StartAngle = InitStartAngle; EndAngle = InitEndAngle;}
    void Show(); // these functions are virtual in Point
    void Hide();
};

// Member functions for Arc

void Arc::Show()
{
    Visible = true;
    arc(X, Y, Radius, StartAngle, EndAngle);
}

void Arc::Hide()
{
    int TempColor;
    TempColor = getcolor();
    setcolor (getbkcolor());
    Visible = false;
    // draw arc in background color to hide it
    arc(X, Y, StartAngle, EndAngle, Radius);
    setcolor(TempColor);
}

int main() // test the new Arc class
{

```

```

int graphdriver = DETECT, graphmode;
initgraph(&graphdriver, &graphmode, "c:\\bgi");
Circle ACircle(151, 82, 50);
Arc AnArc(151, 82, 25, 0, 190);

// you first drag an arc using arrow keys (5 pixels per key)
// press Enter when tired of this!
// Now drag a circle (10 pixels per arrow key)
// Press Enter to end FIGDEMO.

AnArc.Drag(5); // drag increment is 5 pixels
AnArc.Hide();
ACircle.Drag(10); // now each drag is 10 pixels
closegraph();
return 0;
}

```

Ordinary or virtual member functions?

In general, because calling a non-virtual member function is a little faster than calling a virtual one, we recommend that you use ordinary member functions when extensibility is not a consideration, but performance is. Use virtual functions otherwise.

To recap our earlier discussion, let's say you are declaring a class named **Base**, and within **Base** you are declaring a member function named **Action**. How do you decide whether **Action** should be virtual or ordinary? Here's the rule of thumb: Make **Action** virtual if there is a possibility that some future class derived from **Base** will override **Action**, and you want that future code to be accessible to **Base**. Make **Action** ordinary if it is evident that for derived types, **Action** will perform the same steps (even if this involves invoking other, virtual, functions); or the derived types will not make use of **Action**.

Dynamic objects

All the examples shown so far, except for the message array allocation in `MCIRCLE.CPP`, have had static or automatic objects of class types that were declared as usual with their memory being allocated by the compiler at compile time. In this section we look at objects that are created at run time, with their memory allocated from the system's *free memory store*. The creation of dynamic objects is an important technique for many programming applications where the amount of data to be stored in memory

cannot be known before the program is run. An example is a free-form database program that holds data records of various sizes in memory for rapid access.

C++ can use the dynamic memory allocation functions of C such as **malloc**. However, C++ includes some powerful extensions that make dynamic allocation and deallocation of objects easier and more reliable. More importantly, it ensures that constructors and destructors are called. For example,

To allocate an object from free store, declare a pointer to the object's type and assign the result of the expression `new object_type` to the pointer. You can now use the pointer to refer to the newly created object.

```
Circle *ACircle = new Circle(151,82,50);
```

Here *ACircle*, a pointer to type **Circle**, is given the address of a block of memory large enough to hold one object of type **Circle**. In other words, *ACircle* now points to a **Circle** object allocated from free store. A *Circle* constructor is then called to initialize the object according to the arguments supplied.

If you are allocating an array rather than a standard-length data type, use the optional syntax

```
new object [size]
```

For example, to dynamically allocate an array of 50 integers called *counts*, use

```
counts = new int [50];
```

If you wanted to create a dynamic **Point** class object, you might do it like this:

You can find this on your disks: DYNPOINT.CPP. Or use DYNPOINT.PRJ.

```
// DYNPOINT.CPP -- Exercise in Chapter 5, Getting Started
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "figures.h"

int main()
{
    // Assign pointer to dynamically allocated object; call constructor
    Point *APoint = new Point(50, 100);

    // initialize the graphics system
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "..\\bgi");

    // Demonstrate the new object
    APoint->Show();
    cout << "Note pixel at (50,100). Now, hit any key...";
    getch();
    delete APoint;
```



```
closegraph();  
return(0);  
}
```

Destructors and **delete**

Just as you can define a constructor that will be called whenever a new object of a class is created, you can define a *destructor* that will be called when it is time to destroy an object, that is to say, clear its value and deallocate its memory.

Space for static objects is allocated by the compiler; the constructor is called before **main** and the destructor is called after **main**. In the case of **auto** objects, deallocation occurs when the declaration goes out of scope (when the enclosing block terminates). Any destructor you define is called at the time the static or auto objects is destroyed. (If you haven't defined a destructor, C++ uses an implicit, or built-in one.)

If you create a dynamic object using the **new** operator, however, you are responsible for deallocating it, since C++ has no way of "knowing" when the object is no longer needed. You use the **delete** operator to deallocate the memory. Any destructor you have defined is called when **delete** is executed.

The **delete** operator has the syntax

```
delete pointer;
```

where *pointer* is the pointer that was used with **new** to allocate the memory.

You have seen that a constructor for the class **X** is identified by having the same name, viz **X::X()**. The name of a destructor for class **X** is **X::~~X()**. In addition to deallocating memory, destructors can also perform other appropriate actions, such as writing member field data to disk, closing files, and so on.

An example of dynamic object allocation

The next example program provides some practice in the use of objects allocated dynamically from free store, including the use of destructors for object deallocation. The program shows how a linked list of graphics objects might be created in memory and cleaned up using delete calls when the objects are no longer required.

Building a linked list of objects requires that each object contain a pointer to the next object in the list. Type **Point** contains no such pointer. The easy way out would be to add a pointer to **Point**, and in doing so ensure that all **Point**'s derived types also inherit the pointer. However, adding anything to **Point** requires that you have the source code for **Point**, and as noted earlier, one advantage of C++ is the ability to extend existing objects without necessarily being able to recompile them. So for this example we'll pretend that we don't have the source code to **Point** and show how you can extend the graphics tool kit anyway.

See the next listing for the declarations of *List* and *Node*.

One of the many solutions that requires no changes to **Point** is to create a new class not derived from **Point**. Type **List** is a very simple class whose purpose is to head up a list of **Point** objects. Because **Point** contains no pointer to the next object in the list, a simple **struct, Node**, provides that service. **Node** is even simpler than **List**, in that it has no member functions and contains no data except a pointer to type **Point** and a pointer to the next node in the list.

List has a member function that allows it to add new figures to its linked list of **Node** records by inserting a new **Node** object immediately after itself, as a referent to its *Nodes* pointer member. The **Add** member function takes a pointer to a **Point** object, rather than a **Point** object itself. Remember that rules for the class hierarchy in C++ allows pointers to any type publicly derived from **Point** to be passed in the *Item* argument to **List::Add**.

Program *ListDemo* declares a static variable, *AList*, of type **List**, and builds a linked list with three nodes. Each node points to a different graphics figure that is either a **Point** or one of its derived classes. The number of bytes of free storage space is reported before any of the dynamic objects are created, and then again after all have been created. Finally, the whole structure, including the three **Node** records and the three **Point** objects, is cleaned up and removed from memory, thanks to the destructor for the **List** class called automatically for its object *AList*.

This code is on your disks as *LISTDEMO.CPP*.

```
// LISTDEMO.CPP--Example from Object-oriented programming with C++
// LISTDEMO.CPP           Demonstrates dynamic objects
// Link with FIGURES.OBJ and GRAPHICS.LIB

#include <conio.h>           // for getch()
#include <alloc.h>          // for coreleft()
#include <stdlib.h>         // for itoa()
```

```

#include <string.h>           // for strcpy()
#include <graphics.h>
#include "figures.h"

class Arc : public Circle {
    int StartAngle, EndAngle;
public:
    // constructor
    Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
        int InitEndAngle);
    // virtual functions
    void Show();
    void Hide();
};

struct Node { // the list item
    Point *Item; // can be Point or any class derived from Point
    Node *Next; // point to next Node object
};

class List { // the list of objects pointed to by nodes
    Node *Nodes; // points to a node
public:
    // constructor
    List();
    // destructor
    ~List();
    // add an item to list
    void Add(Point *NewItem);
    // list the items
    void Report();
};

// definitions for standalone functions
void OutTextLn(char *TheText)
{
    outtext(TheText);
    moveto(0, gety() + 12); // move to equivalent of next line
}

void MemStatus(char *StatusMessage)
{
    unsigned long MemLeft; // to match type returned by
                          // coreleft()
    char CharString[12]; // temp string to send to outtext()
    outtext(StatusMessage);
    MemLeft = long (coreleft());

    // convert result to string with ltoa then copy into
    // temporary string
    ltoa(MemLeft, CharString, 10);
}

```

```

        OutTextLn(CharString);
    }

    // member functions for Arc class
Arc::Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
        int InitEndAngle) : Circle (InitX, InitY, InitRadius)
        // calls Circle
        // constructor
{
    StartAngle = InitStartAngle;
    EndAngle = InitEndAngle;
}

void Arc::Show()
{
    Visible = true;
    arc(X, Y, StartAngle, EndAngle, Radius);
}

void Arc::Hide()
{
    unsigned TempColor;
    TempColor = getcolor();
    setcolor(getbkcolor());
    Visible = false;
    arc(X, Y, StartAngle, EndAngle, Radius);
    setcolor(TempColor);
}

// member functions for List class
List::List () {
    Node *N;
    N = new Node;
    N->Item = NULL;
    N->Next = NULL;
    Nodes = NULL;
    // sets node pointer to "empty"
    // because nothing in list yet
}

List::~List() // destructor
{
    while (Nodes != NULL) { // until end of list
        Node *N = Nodes; // get node pointed to
        delete(N->Item); // delete item's memory
        Nodes = N->Next; // point to next node
        delete N; // delete pointer's memory
    };
}

void List::Add(Point *NewItem)
{

```

```

Node *N;           // N is pointer to a node
N = new Node;     // create a new node
N->Item = NewItem; // store pointer to object in node
N->Next = Nodes;  // next item points to current list pos
Nodes = N;        // last item in list now points
                  // to this node
}

void List::Report()
{
    char TempString[12];
    Node *Current = Nodes;
    while (Current != NULL)
    {
        // get X value of item in current node and convert to string
        itoa(Current->Item->GetX(), TempString, 10);
        outtext("X = ");
        OutTextLn(TempString);
        // do the same thing for the Y value
        itoa(Current->Item->GetY(), TempString, 10);
        outtext("Y = ");
        OutTextLn(TempString);
        // point to the next node
        Current = Current->Next;
    };
}

void setlist(void);

// Main program
main()
{
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "c:\\\\bgi");

    MemStatus("Free memory before list is allocated: ");
    setlist();
    MemStatus("Free memory after List destructor: ");
    getch();
    closegraph();
}

void setlist() {
    // declare a list (calls List constructor)
    List AList;

    // create and add several figures to the list
    Arc *Arc1 = new Arc(151, 82, 25, 200, 330);
    AList.Add(Arc1);
    MemStatus("Free memory after adding arc1: ");
    Circle *Circle1 = new Circle(200, 80, 40);
}

```

```

AList.Add(Circle1);
MemStatus("Free memory after adding circle1: ");
Circle *Circle2 = new Circle(305, 136, 35);
AList.Add(Circle2);
MemStatus("Free memory after adding circle2: ");
// traverse list and display X, Y of the list's figures
AList.Report();
// The 3 AList nodes and the Arc and Circle objects will be
// deallocated automatically by their destructors when they
// go out of scope in main(). Arc and Circle use implicit
// destructors in contrast to the explicit ~List destructor.
// However, you could delete explicitly here if you wish:
// delete Arc1; delete Circle1; delete Circle2;
getch(); // wait for a keypress
return;
}

```

Once you have mastered LISTDEMO.CPP, you might wish to develop a more satisfying solution based on the following idea: define a new class called **PointList** by multiple inheritance from classes **Point** and **List**.

More flexibility in C++

Although it will take you some time to master the nuances of this new style of programming, you have now learned the essential elements of C++. There are a number of additional features that we touch on briefly here so that you will know what they are and how to use them.

None of these features are essential to understanding C++, but they can add to its flexibility and power.

- Inline functions outside class definitions
- Default function arguments
- Overloading functions and multiple constructors
- Friend functions—another way of providing access to a class
- Overloading operators to provide new meanings
- More about C++ I/O and the streams library

Inline functions outside class definitions

You have already seen that you can include an *inline* definition of a member function within the class declaration as shown here with the **Point** class:

```

class Point: { // define Point class
    int X; // these are private by default
    int Y;
public: // public member functions
    Point(int InitX, int InitY) {X = InitX, Y = InitY;}
    int GetX(void) {return X;}
    int GetY(void) {return Y;}
};

```

All three member functions of the **Point** class are defined inline, so no separate definition is necessary. For functions with only a line or so of code, this provides a more compact, easier to read description of the class.

Remember that inline code is enclosed in braces.

Functions can also be declared as *inline*. The only difference is that you have to start the function declaration with the keyword **inline**. For example, in LISTDEMO.CPP, there is an operation that simply moves the output location for text in graphics mode down one line (it is used in the function *OutTextLn*). If this function were to be used in many other places in the code, it would be more efficient to declare it as a separate inline function:

```
inline void graphLn() { moveto(0, gety() + 12); }
```

If you wish, you can format your inline definitions to look more like a regular function definition:

```

inline void graphLn()
{
    moveto(0, gety() + 12);
}

```

Another advantage to using the **inline** keyword is that you can avoid revealing your implementation code in the distributed header files.

Functions with default arguments

If you plan to use certain values often for a function, use those values as default arguments for the function.

Default values must be specified the first time the function name is given.

You can define functions that you can call with fewer arguments than defined. The arguments that you don't supply are given default values. If you are going to be using these default values most of the time, such an "abbreviated" call saves typing. You don't lose flexibility, because when you want to override the defaults, you simply specify the values you want.

For example, the following version of the constructor for the *Circle* class gives a default circle of radius 50 pixels centered at (X = 200, Y = 200). A more portable program, of course, would have to

determine the graphics hardware available and adjust these values accordingly.

As with ANSI C, C++ allows functions to have a variable number of arguments, such as `float average(int number, ...)`, which can take one or more integer values. See Chapter 11 for details.

```
class Circle : public Point { // Derived from class Point and
                               // ultimately from class Location
protected:
    int Radius;
public:
    Circle(int InitX = 200, int InitY = 200, int InitRadius = 50);
    void Show(void);
    void Hide(void);
    void Expand(int ExpandBy);
    void Contract(int ContractBy);
};
```

Now the declaration

```
Circle ACircle;
```

gives you a circle with the default center at (200,200) and radius 50. The declaration

```
Circle ACircle(50, 100);
```

gives a circle with center at 50, 100, with the default radius of 50.

The declaration

```
Circle ACircle(300)
```

gives a circle at $X = 300$, with default $Y = 200$ and radius = 50.

Any default arguments must be in consecutive rightmost positions in the argument list. For example, you couldn't declare

```
void func(int a = 10, int b, int c)
```

because the compiler wouldn't know which values are being supplied. You *could* declare

```
void func(int a, int b, int c = 10)
```

More about overloading functions

Overloading is an important concept in C++. When several different functions (whether member functions or ordinary) are defined with the same name within the same scope, they are said to be overloaded. You have met several such cases; for example, the three functions called **cube()** on page 133. (Earlier versions of C++ required that such declarations be preceded by the keyword **overload**, but this is now obsolete.)

The basic idea is that overloaded function calls are distinguished by comparing the types of the actual arguments in the call and the formal argument signatures in the function definitions. The actual rules for disambiguation are beyond the scope of a primer and should rarely affect the beginner (who is hereby cautioned against the rash replication of function names). Among the possible complications are functions called with default actual arguments, or with a variable numbers of arguments; also, there are the normal C conversions of argument type to be considered, together with additional type conversions peculiar to C++. When faced with a call to a heavily overloaded function, the compiler tries to find a *best match*. If there is no best match, a compiler error results.

One of the most common cases is overloading a constructor so as to provide several different ways to create a new object of a class. To illustrate this, we will define a very simple *String* class.

You can load and run *STRING.CPP* from the IDE. After running it, you'll have to activate the User Screen to see the output. Use the hot key Alt-F5 or the Window | User Screen menu item.

```
//STRING.CPP--Example from Chapter 5 of Getting Started */
#include <iostream.h>
#include <string.h>

class String {
    char *char_ptr; // pointer to string contents
    int length; // length of string in characters
public:
    // three different constructors
    String(char *text); // constructor using existing string
    String(int size = 80); // creates default empty string
    String(String& Other_String); // for assignment from another
                                // object of this class
    ~String() {delete char_ptr;};
    int Get_len (void);
    void Show (void);
};

String::String (char *text)
{
    length = strlen(text); // get length of text
    char_ptr = new char[length + 1];
    strcpy(char_ptr, text);
};

String::String (int size)
{
    length = size;
    char_ptr = new char[length+1];
    *char_ptr = '\0';
};
```

```

String::String (String& Other_String)
{
    length = Other_String.length;    // length of other string
    char_ptr = new char [length + 1]; // allocate the memory
    strcpy (char_ptr, Other_String.char_ptr); // copy the text
};

int String::Get_len(void)
{
    return (length);
};

void String::Show(void)
{
    cout << char_ptr << "\n";
};

main ()                                // test the functions
{
    String AString ("Allocated from a constant string.");
    AString.Show();

    String BString;                    // uses default length
    cout << "\n" << BString.Get_len() << "\n" ; //display length
    BString = "This is BString";

    String CString(BString);          // invokes the third constructor
    CString.Show();                  // note its contents
}

```

When calling a constructor with no arguments (or when accepting all default arguments), don't put empty parentheses after the name of the object. For example, declare `String BString;` not `String BString();`.

The class **String** has three different constructors. The first takes an ordinary string constant such as "This is a string" and initializes a string with these contents. The second constructor uses a default length of 80, and allocates the string without storing any characters in it (this might be used to create a temporary buffer). Note that you can override the default simply by calling the constructor with a different length: Instead of declaring `String AString`, you could declare, for example, `String AString(40)`.

The third constructor takes a reference to another object of type **String** (recall that the ampersand after a type means a reference to that type, and is used to pass the address of a variable rather than a copy of its contents.) With this constructor you can now write statements such as these:

```

String AString("This is the first string"); // create and initialize
String BString = AString; // create then assign BString from AString

```

Note that constructors are involved in three related but separate aspects of an object's life story: creation, initialization, and assignment. The use of the = operator for class assignments leads us

nicely to our next topic, operator overloading. Unless you define a special = operator for a class, C++ defaults to a member-by-member assignment.

Overloading operators to provide new meanings

C++ has a special feature found in few other languages: existing operators such as + can be given new definitions to make them work in an appropriate, user-defined manner with your own class objects. Operators are a very concise way of doing business. If you didn't have them, an expression such as `line * width + pos` would have to be written something like this: `add(mult(line, width), pos)`. Fortunately, the arithmetic operators in C (and C++) already know how to work with all of the numeric data types — the same + that works with **int** values also works with **float**, for example. The same operator is used, but the code generated is clearly different, since integers and floating-point numbers are represented differently in memory. In other words, operators such as + are already overloaded, even in regular C. C++ simply extends this idea to allow user-defined versions of the existing operators.

*Whitespace is okay between the keyword **operator** and the operator symbol.*

To define an operator, you define a function that has as its name the keyword **operator** followed by the operator symbol. (So, for example, `operator+` names a new version of the + operator.) All operator functions are by definition overloaded: They use an operator that already has a meaning in C, but they redefine it for use with a new data type. The + operator, for example, already has the capability to add two values of any of the standard numeric types (**int**, **float**, **double**, and so on).

Now we can add a + operator to the **String** class. This operator will concatenate two string objects (as in BASIC) returning the result as a string object with the appropriate length and contents. Since concatenating is "adding together," the + symbol is the appropriate one to use. The BASIC lobby often criticizes C for not having such natural string operations. With C++, you can go far beyond the built-in BASIC string facilities.

The file XSTRING.CPP, available on your distribution disks, has the following additions to STRING.CPP to provide a simple operator +.

```
//XSTRING.CPP--Example from Object-oriented programming with C++  
// version of STRING.CPP with overloaded operator +  
  
#include <iostream.h>
```

```

#include <string.h>

class String {
    char *char_ptr; // pointer to string contents
    int length;     // length of string in characters
public:
    // three different constructors
    String(char *text);           // constructor using existing string
    String(int size = 80);       // creates default empty string
    String(String& Other_String); // for assignment from another
                                // object of this class
    ~String() {delete char_ptr;}; // inline destructor
    int Get_len (void);
    String operator+ (String& Arg);
    void Show (void);
};

String::String (char *text)
{
    length = strlen(text); // get length of text
    char_ptr = new char[length + 1];
    strcpy(char_ptr, text);
};

String::String (int size)
{
    length = size;
    char_ptr = new char[length+1];
    *char_ptr = '\0';
};

String::String (String& Other_String)
{
    length = Other_String.length; // length of other string
    char_ptr = new char [length + 1]; // allocate the memory
    strcpy (char_ptr, Other_String.char_ptr); // copy the text
};

String String::operator+ (String& Arg)
{
    String Temp( length + Arg.length );
    strcpy(Temp.char_ptr, char_ptr);
    strcat(Temp.char_ptr, Arg.char_ptr);
    return Temp;
}

int String::Get_len(void)
{
    return (length);
};

void String::Show(void)

```

```

{
    cout << char_ptr << "\n";
};

main ()
{
    String AString ("The Quick Brown fox");
    AString.Show();

    String BString(" jumps over Bill");
    String CString;
    CString = AString + BString;
    CString.Show();
}

```

When you run the program, *CString* is assigned the concatenation of the two strings *AString* and *BString*. So **CString.Show()** displays

To see this display from the IDE, press Alt-F5 or Window / User.

```
The Quick Brown Fox jumps over Bill
```

The overloaded **+** takes only one explicit argument, so you may wonder how it manages to concatenate two strings. Well, the compiler treats the expression *AString* + *BString* as

```
AString.(operator +(BString))
```

so the **+** operator does access two string objects. The first is the **String** object currently being referenced, and the other is a second string object. The operator function adds the lengths of the two strings together, then uses the **strcat** library function to combine the contents of the two strings, which is then returned. This remarkable trick makes use of a "hidden" pointer known as **this**. What is **this**?

Every call by a member function sets up a pointer to the object upon which the call is acting. This pointer can be referred via the keyword **this** (also known as "self" or rather "pointer-to-self" in OOP parlance), allowing functions to access the actual object. Now **this** is of type "pointer to String", so the return value must be ***this**, the actual current object, is exactly what is needed. Note, too, that individual members of the object involved in a function call can be referenced via the expression **this->member**. A further point to watch: **this** is available only to member functions, not to friend functions.

There are some restrictions when overloading operators:

- C++ can't distinguish between the prefix and postfix versions of **++** and **--**.

- The operator you wish to define must already exist in the language. For example, you can't define the operator #.
- You can't overload the following operators:
 - . * :: ?:
- Overloaded operators keep their original precedence.
- If @ stands for any unary operator, the expressions @x and x@ may be interpreted as either x.operator@() or as operator@(x). If both forms have been declared, the compiler will try to resolve the ambiguity by matching the arguments. Similarly, with an overloaded binary operator, @, x@y could mean either x.operator@(y) or operator@(x,y), and the compiler needs to look at the arguments if both forms have been defined. You saw an example of a binary operator in the string version of +, where AString + BString was interpreted as AString.(operator+(BString)).

friend functions

Normally, access to the private members of a class is restricted to member functions of that class. Occasionally it may be necessary to give outside functions access to the class's private data. The **friend** declaration within a class declaration lets you specify outside functions (or even outside classes) that will be granted access to the declared class's private members. You'll sometimes see an overloaded operator declared as a friend, but generally speaking friend functions are to be used sparingly—if their need persists in your project, it is often a sign that your class hierarchy needs revamping.

But, suppose that there is a fancy formatted printing function called *Fancy_Print* that you want to have access to the contents of your objects of class **String**. You can add the following line to the list of member function declarations:

The position of the declaration doesn't matter.

```
class String {
...
friend void Fancy_Print(String& AString);
...
}
```

In this admittedly artificial example, the *Fancy_Print* function can access the members *char_ptr* and *length* of objects of the **String** class. That is, if *AString* is a string object, *Fancy_Print* can access *AString.char_ptr* and *AString.length*.

If the *Fancy_Print* function is a member of another class (for example, the class **Format**), use the scope resolution operator in the friend declaration:

```
friend void Format::Fancy_Print(String& AString);
```

You can also make a whole class the friend of the declared class, by using the word **class** in the declaration:

```
friend class Format;
```

Now any member function of the **Format** class can access the private members of the **String** class. Note that in C++, as in life, friendship is not transitive: if **X** is a friend of **Y**, and **Y** is a friend of **Z**, it does not follow that **X** is a friend of **Z**.

The friend declaration should be used only when it is really necessary; when without it you would have to have a convoluted class hierarchy. By its nature, the friend declaration diminishes encapsulation and modularity. In particular, if you find yourself wanting to make a whole class the friend of another class, consider instead the possibility of deriving a common derived class and using it to access the needed members.

The C++ streams libraries

This section is intended merely to whet your appetite and point you in the right direction. We encourage you to study the examples in Chapter 16, "Using C++ streams," and experiment on your own.

While all the stdio library I/O functions (such as **printf** and **scanf**) are still available, C++ also provides a group of classes and functions for I/O defined in the **iostreams** library. To access these, your program must have the directive `#include <iostream.h>`, as you may have noticed in some of our examples.

There are many advantages in using **iostreams** rather than stdio. The syntax is simpler, more elegant, and more intuitive. The C++ stream mechanism is also more efficient and flexible. Formatting output, for example, is simplified by extensive use of overloading. The same operator can be used to output both predefined and user-defined data types, avoiding the complexities of the **printf** argument list.

Starting with the stream as an abstraction for modeling any flow of data from a source (or producer) to a sink (or consumer), **iostream** provides a rich hierarchy of classes for handling buffered and unbuffered I/O for files and devices.



Turbo C++ also supports the older (version 1.x) C++ **stream** library to assist programmers during the transition to the new **iostream** library of C++ release 2.1. If you have any C++ code that

uses the obsolete **stream** classes, you can still maintain and run it with Turbo C++. However, given a choice, you should convert to the more efficient **iostream** and avoid **stream** when writing new code. Chapter 16, "Using C++ streams," explains the differences between the **stream** and **iostream** libraries, and provides some hints on conversion. See also OLDSTR.DOC on your distribution disks.

In this section we cover only the simpler classes in **iostream**. For a more detailed account, you should read Chapter 16, "Using C++ streams." You can also browse through `iostream.h` on your distribution disks to see the many classes defined there and how they are derived using both single and multiple inheritance.

Standard I/O C++ provides four predefined stream objects defined as follows:

- **cin** Standard input, usually the keyboard, corresponding to **stdin** in C
- **cout** Standard output, usually the screen, corresponding to **stdout** in C
- **cerr** Standard error output, usually the screen, corresponding to **stderr** in C
- **clog** A fully-buffered version of **cerr** (no C equivalent)

You can redirect these standard streams from and to other devices and files. (In C, you can redirect only **stdin** and **stdout**.) You have already seen the most common of these, **cin** and **cout**, in some of the examples in this chapter.

A simplified view of the **iostream** hierarchy, from primitive to specialized, is as follows:

- **streambuf** Provides methods for memory buffers
- **ios** Handles stream state variables and errors
- **istream** Handles formatted and unformatted character conversions *from* a **streambuf**
- **ostream** Handles formatted and unformatted character conversions *to* a **streambuf**
- **iostream** Combines **istream** and **ostream** to handle bidirectional operations on a single stream
- **istream_withassign** Provides constructors and assignment operators for the **cin** stream

■ **ostream_withassign** Provides constructors and assignment operators for the **cout**, **cerr** and **clog** streams

<< used with streams is called the insertion or put to operator, while >> is called the extraction or get from operator.

The **istream** class includes overloaded definitions for the **>>** operator for the standard types [**int**, **long**, **double**, **float**, **char**, and **char*** (string)]. Thus the statement `cin >> x;` calls the appropriate **>>** operator function for the **istream** `cin` defined in `istream.h` and uses it to direct this input stream into the memory location represented by the variable `x`. Similarly, the **ostream** class has overloaded definitions for the **<<** operator, which allows the statement `cout << x;` to send the value of `x` to `ostream` `cout` for output.

These operator functions return a reference to the appropriate stream class type (for example, **ostream&**) in addition to moving the data. This allows you to chain several of these operators together to output or input sequences of characters:

```
int i=0, x=243; double d=0;
cout << "The value of x is " << x << '\n';
cin >> i >> d; // key an int, space, then a double
```

The second line would display "The value of x is 243" followed by a new line. The next statement would ignore whitespace, read and convert the keyed characters to an integer and place it in `i`, ignore following whitespace, read and convert the next keyed characters to a **double** and place it in `d`.

The following program simply copies **cin** to **cout**. In the absence of redirection, it copies your keyboard input to the screen:

This program simply stores each input character in the variable `ch` and then outputs the value of `ch` to the screen.

```
// COPYKBD.CPP      Copies keyboard input to screen
#include <iostream.h>
int main(void)
{
    char ch;
    while (cin >> ch)
        cout << ch;
}
```

Note how you can test (`cin >> ch`) as a normal Boolean expression. This useful trick is made possible by definitions in the class **ios**. Briefly, an expression such as (`cout`) or (`cin >> ch`) is cast as a pointer, the value of which depends on the error state of the stream. A null pointer (tested as false) indicates an error in the stream, while a non-null pointer (tested as true) means no errors.

You can also reverse the test using `!`, so that `(!cout)` is true for an error in the `cout` stream and false if all is well:

```
if (!cout) errmsg("Output error!");
```

Formatted output Simple I/O in C++ is efficient because only minimal conversion is done according to the data type involved. For integers, conversion is the same as the default for `printf`. The statements

```
int i=5; cout << i;
```

and

```
int i=5; printf("%d",i);
```

give the same result.

Formatting is determined by a set of format state flags enumerated in `ios`. These determine, for each active stream, the conversion base (decimal, octal, and hexadecimal), padding left or right, the floating-point format (scientific or fixed), and whether whitespace is to be skipped on input. Other parameters you can vary include field width (for output) and the character used for padding. These flags can be tested, set, and cleared by various member functions. The following snippet shows how the functions `ios::width` and `ios::fill` work:

```
int previous_width, i = 87;
previous_width = cout.width(7); // set field width to 7
                                // and save previous width
cout.fill('*');                // set fill character to *
cout << i << '\n';             // display *****87 <newline>
// after << the width is cleared to 0
// previous width may have been set without a subsequent <<
// so you may want to restore it with the following line.
cout.width(previous_width);
```

Setting *width* to zero (the default) means that the display will take as many screen positions as needed. If the given width is insufficient for the correct representation, a width of zero is assumed (that is, there is no truncation). Default padding gives right justification (left padding) for all types.

`setf` and `unsetf` are two general functions for setting and clearing format flags:

```
cout.setf(ios::left, ios::adjustfield);
```

This sets left padding. The first argument uses enumerated mnemonics for the various bit positions (possibly combined using `&`

and l), and the second argument is the target field in the format state. **unsetf** works the same way but clears the selected bits. (More on these in Chapter 16, "Using C++ streams.")

Manipulators

A rather more elegant way of setting the format flags (and performing other stream chores) uses special mechanisms known as *manipulators*. Like the << and >> operators, manipulators can be embedded in a chain of stream operations:

```
cout << setw(7) << dec << i << setw(6) << oct << j;
```

Without manipulators, this would take six separate statements.

The *parameterized manipulator* **setw** takes a single **int** argument to set the field width.

The non-parameterized manipulators, such as **dec**, **oct**, and **hex**, set the conversion base to decimal, octal, and hexadecimal. In the above example, `int i` would display in decimal on a field of width 7; `int j` would display in octal on a field of width 6.

Other simple parameterized manipulators include **setbase**, **setfill**, **setprecision**, **setiosflags**, and **resetiosflags**. To use any of the parameterized manipulators, your program must include both of these header files: **iomanip.h** and **iostream.h**. Non-parameterized manipulators do not require **iomanip.h**.

Useful non-parameterized manipulators include:

- **ws** (whitespace extractor): `istream >> ws`; discards any whitespace in **istream**.
- **endl** (endline and flush): `ostream << endl`; inserts a newline in **ostream**, then flush the **ostream**.
- **ends** (end string with null): `ostream << ends`; appends a null to **ostream**.
- **flush** (flush output stream): `ostream << flush`; flushes the **ostream**.

put, write, and get

Two general output functions are worthy of mention: **put** and **write**, declared in **ostream** as follows:

```
ostream& ostream::put(char ch);  
// send ch to ostream
```

```
ostream& ostream::write(const char* buff, int n);
// send n characters from buff to ostream; watch the size of n!
```

put and **write** let you output unformatted binary data to an **ostream** object. **put** outputs a single character, while **write** can send any number of characters from the indicated buffer. **write** is useful when you want to output raw data that may include nulls. (Note that writing binary data requires that the file be opened in binary mode.) The normal string extractor would not work since it terminates on a null.

The input version of **put** is called **get**:

```
char ch;
cin.get(ch);
// grab next char from cin whether whitespace or not
```

Another version of **get** lets you grab any number of raw, binary characters from an **istream**, up to a designated maximum, and place them in a designated buffer (as with **write**, files must be opened in binary mode):

```
istream& istream::get(char *buf, int max, int term='\n');
// read up to max chars from istream, and place them in buf. Stop if
// term char is read.
```

You can set *term* to a specific terminating character (the default is the newline character), at which **get** will stop if reached before *max* characters have been transferred to *buf*.

Disk I/O The **iostream** library includes many classes derived from **streambuf**, **ostream**, and **istream**, thereby allowing a wide choice of file I/O methods. The **filebuf** class, for example, supports I/O through file descriptors with member functions for opening, closing, and seeking. Contrast this with the class **stdiobuf** that supports I/O via **stdio** FILE structures, allowing some compatibility when you need to mix C and C++ code.

The most generally useful classes for the beginner are **ifstream** (derived from **istream**), **ofstream** (derived from **ostream**), and **fstream** (derived from **iostream**). These all support formatted file I/O using **filebuf** objects. Here's a simple example that copies an existing disk file to another specified file:

*This code is available as
DCOPY.CPP.*

```
// DCOPY.CPP -- Example from Object-oriented programming with C++
// DCOPY source-file destination-file
// copies existing source-file to destination-file
// If latter exists, it is overwritten; if it does not
```

```

// exist, DCOPY will create it if possible
*/

#include <iostream.h>
#include <process.h> // for exit()
#include <fstream.h> // for ifstream, ofstream

main(int argc, char* argv[]) // access command-line arguments
{
    char ch;
    if (argc != 3) // test number of arguments
    {
        cerr << "USAGE: dcopy file1 file2\n";
        exit(-1);
    }

    ifstream source; // declare input and output streams
    ofstream dest;

    source.open(argv[1], ios::nocreate); // source file must be there
    if (!source)
    {
        cerr << "Cannot open source file " << argv[1] <<
            " for input\n";
        exit(-1);
    }
    dest.open(argv[2]); // dest file will be created if not found
    // or cleared/overwritten if found
    if (!dest)
    {
        cerr << "Cannot open destination file " << argv[2] <<
            " for output\n";
        exit(-1);
    }

    while (dest && source.get(ch)) dest.put(ch);

    cout << "DCOPY completed\n";

    source.close(); // close both streams
    dest.close();
}

```

Note first that `#include <fstream>` also pulls in `iostream.h`. DCOPY uses the standard method of accessing command-line arguments to check whether the user specified the two files involved. When this argument list is used with the `main` function, the argument `argc` contains the number of command-line arguments (including the name of the program itself), and the strings `argv[1]` and `argv[2]` contain the two file names entered. A typical command-line invocation of this program would be

```
dcopy letter.spr letter.bak
```

To see how DCOPY works, examine the following lines:

```
ifstream source; // declare an input stream (ifstream object)
...
open.source(argv[1],ios::nocreate); // source file must be there
```

The declaration invokes a constructor of **ifstream** (the class for handling input file streams) to create a stream object called *source*. Before we can make use of *source*, we must create a file buffer and associate the stream and buffer with a real, physical file. Both tasks are performed by the member function **open** in **ifstream**. The **open** function needs a file name string and, optionally, one or two other arguments to specify the mode and protection rights. The file name here is given as *argv[1]*, namely, the source file supplied in the command line.

A neater alternative to the above declaration is:

```
ifstream source(argv[1],ios::nocreate); // source file must be there
// this creates source and opens the file as well
```

The mode argument **ios::nocreate** tells **open** not to create a file if the named file is not found. For DCOPY, we clearly want **open** to fail if the named source file is not on the disk. Later, you'll see the other mode arguments available. If the file *argv[1]* cannot be opened for any reason (usually because the file is not found), the value of *source* is effectively set to zero (false), so that `!source` tests true, giving us an error message, then exiting.

In fact, we could determine the possible reason for the failure to open the source file by examining the error bits set in the *stream state*. The member functions **eof**, **fail**, and **bad** test various error bits and return true if they are set. Alternatively, **rdstate** returns the error state in an **int**, and you can then test which bits are set. The **eof** (end of file) is not really an error *per se*, but it needs to be tested and acted upon since a stream cannot be usefully accessed beyond its final character. Note that once a stream is in an error state (including **eof**), no further I/O is permitted. The function **clear** is provided for clearing some or all error bits, allowing you to resume after clearing a nonfatal situation.

Back in DCOPY.CPP, if all is well with the source file, we then try to open the destination file with the **ofstream** object, *dest*. With output files, the default situation is that a file will be created if it does not exist; if it exists it will be cleared and recreated as an

empty file. You can modify this behavior by adding a second argument, *mode*, to the declaration of *dest*. For example,

```
ofstream dest(argv[2], ios::app | ios::nocreate);
```

will try to open *dest* in *append* mode, failing if *dest* is *not* found. In append mode, the data in the source file would be added to the end of *dest*, leaving the previous contents undisturbed. Other mode flags enumerated in class **ios** (note the scope operator in **ios::app**), are **ate** (seek to end of file); **in** (open for input, used with **fstreams**, since they can be opened for both input and output); **out** (open for output, also used with **fstreams**); **trunc** (discard contents if file exists); **n** (replace (fail if file exists)).

Once both files have been opened, the actual copying is achieved in typically condensed C fashion. Consider the Boolean expression tested by the **while** loop:

```
(dest && source.get(ch))
```

When C tests (x && y), it will not bother to test y if x proves false. Since dest is less likely to "fail" than source.get(ch), you might consider reversing the entries.

We have seen that *dest* will test true until an error occurs. Similarly the call *source.get(ch)* will test true until either a reading error occurs or until the end of the file is reached. In the absence of "hard" errors, then, the loop **gets** characters from *source* and **puts** them in *dest* until an end of file situation makes *source* false.

There are many more file I/O features in the **iostream** library. And **iostream** can also help you with in-memory formatting, where your streams are in RAM. Special classes, such as **stringstream**, are provided for in-memory stream manipulation.

I/O for user-defined data types

A real benefit with C++ streams is the ease with which you can overload **>>** and **<<** to handle I/O for your own personal data types. Consider a simple data structure that you may have declared:

```
struct emp {
    char *name;
    int dept;
    long sales;
};
```

To overload `<<` to output objects of type *emp*, you need the following definition:

```
ostream& operator << (ostream& str, emp& e)
{
    str << setw(25) << e.name << ": Department " << setw(6) << e.dept <<
    << tab << " Sales $" << e.sales << '\n';
    return str;
}
```

Note that the operator-function `<<` must return **ostream&**, a reference to **ostream**, so that you can chain your new `<<` just like the predefined insertion operator. You can now output objects of type *emp* as follows:

```
#include <iostream.h>
#include <iomanip.h>           // don't forget this!
...
emp jones = {"S. Jones", 25, 1000};
cout << jones;
```

giving the display

```
S. Jones: Department 25    Sales $1000
```

Did you spot the manipulator **tab** in the `<<` definition? This is not a standard manipulator—but a user-defined one:

```
ostream& tab(ostream& str) {
    return str << '\t';
}
```

This, of course, is trivial, but nevertheless makes for more legible code.

An input routine for *emp* can be similarly devised by overloading `>>`. This is left as an exercise for the reader.

Where to now?

A suggestion for your first C++ project is to take the FIGURES module shown on page 167 (you have it on disk) and extend it. Points, circles, and arcs are by no means enough. Create objects for lines, rectangles, and squares. When you're feeling more ambitious, create a pie-chart object using a linked list of individual pie-slice figures.

One more subtle challenge is to implement classes to handle relative position. A relative position is an offset from some base point, expressed as a positive or negative difference. A point at relative coordinates $-17,42$ is 17 pixels to the left of the base point, and 42 pixels down from that base point. Relative positions are necessary to combine figures effectively into single larger figures, since multiple-figure combinations cannot always be tied together at each figure's anchor point. Better to define an *RX* and *RY* field in addition to anchor point *X,Y*, and have the final position of the object onscreen be the sum of its anchor point and relative coordinates.

Once you feel comfortable with C++, start building its concepts into your everyday programming chores. Take some of your more useful existing utilities and rethink them in C++ terms. Try to see the classes in your hodgepodge of function libraries—then rewrite the functions in class form. You'll find that libraries of classes are much easier to reuse in future projects. Very little of your initial investment in programming effort will ever be wasted. You will rarely have to rewrite a class from scratch. If it will serve as is, use it. If it lacks something, extend it. But if it works well, there's no reason to throw away any of what's there.

Conclusion

C++ is a direct response to the complexity of modern applications, complexity that has often made many programmers throw up their hands in despair. Inheritance and encapsulation are extremely effective means for managing complexity. C++ imposes a rational order on software structures that, like a taxonomy chart, imposes order without imposing limits.

Add to that the promise of the extensibility and reusability of existing code, and you not only have a toolkit—you have tools to build new tools!

Hands-on C++

This chapter is a concise, hands-on tutorial for C++.

In order to give you a sense of how C++ looks and how to accomplish tasks in C++, this chapter moves quickly through a large number of concepts with a minimum of verbiage. It is intended to be used as you work at your computer; you can load and run each of these programs (which are in your EXAMPLES subdirectory, along with any header and other files that you'll need). If you want a more in-depth treatment of C++, especially of the concepts underlying object-oriented programming, read Chapter 4, "Object-oriented programming with C++." You might also want to refer to Chapter 13, "C++ specifics," for precise details about the syntax and use of C++.

Important!

In this chapter, we assume that you are familiar with the C language, and that you know how to compile, link, and execute a source program with Turbo C++. We start with simple examples that grow in complexity so that new concepts will stand out. It is reasonable that such examples will not be bulletproof (in other words, they don't check for memory failure and so on). This chapter is not a treatise on data structures or professional programming techniques; instead, it is a gentle introduction to a complicated language.

This chapter is divided into two sections. The first section provides C++ alternatives to C programming knowledge and habits you might have. The second section provides a swift introduction to the kernel of C++: Object-oriented programming using classes and inheritance.

A better C: Making the transition from C

When referring to line numbers, we've counted blank lines.

Although knowing C is helpful to learning C++, sometimes that knowledge can get in the way, particularly in the areas that aren't specifically object-oriented programming, yet where C++ does things differently from C. For that reason, this section shows how to accomplish in C++ many of the same kinds of actions you would perform in C: writing text to the screen, commenting your code, creating and using constants, working with stream I/O and inline functions, and so on.

Program 1

Source

```
// ex1.cpp: A First Glance
// from Hands-on C++
#include <iostream.h>

main()
{
    cout << "Frankly, my dear...\n";
    cout << "C++ is a better C.\n";
}
```

Output

```
Frankly, my dear...
C++ is a better C.
```

Note the new comment syntax in the first line of this program. All characters from the first occurrence of double slashes to the end of a line are considered a comment, although you can still use the traditional `/*...*/` style. File names that have a `.CPP` extension are assumed to be C++ files (or you could use the command-line compiler option `-P`).

The third line includes the standard header file `iostream.h`, which replaces much of the functionality of `stdio.h`. `cout` is an *output stream*, and is used to send characters to standard output (as `stdout` does in C). The `<<` operator (pronounced "put to") sends the data on its right to the stream on its left. The context of the `<<` operator here distinguishes it from the arithmetic shift-left operator, which uses the same symbol. (Such multiple use of operators and functions is quite common in C++ and is called *overloading*.)

Program 2

Source

```
// ex2.cpp: An interactive example
// from Hands-on C++
#include <iostream.h>

main()
{
    char name[16];
    int age;

    cout << "Enter your name: ";
    cin >> name;
    cout << "Enter your age: ";
    cin >> age;

    if (age < 21)
        cout << "You young whippersnapper, " << name << "!\n";
    else if (age < 40)
        cout << name << ", you're still in your prime!\n";
    else if (age < 60)
        cout << "You're over the hill, " << name << "!\n";
    else if (age < 80)
        cout << "I bow to your wisdom, " << name << "!\n";
    else
        cout << "Are you really " << age << ", " << name << "?\n";
}
```

Sample execution

```
Enter your name: Don
Enter your age: 40
You're over the hill, Don!
```

cin is an input stream connected to standard input. It can correctly process all the standard data types. You may have noticed in C that printing a prompt without a newline character to **stdout** required a call to **flush(stdout)** in order for the prompt to appear. In C++, whenever **cin** is used it flushes **cout** automatically (you can turn this automatic flushing off — it's on by default).

Program 3

Source

```
// ex3.cpp: Inline Functions
// from Hands-on C++
#include <iostream.h>

const float Pi = 3.1415926;

inline float area(const float r) {return Pi * r * r;}

main()
```

```

{
    float radius;

    cout << "Enter the radius of a circle: ";
    cin >> radius;
    cout << "The area is " << area(radius) << "\n";
}

```

Sample execution

```

Enter the radius of a circle: 3
The area is 28.274334

```

A constant identifier behaves like a normal variable (that is, its scope is the block that defined it, and it is subject to type checking) except that it cannot appear on the left-hand side of an assignment statement (or anywhere an lvalue is required). Using **#define** is *almost* obsolete in C++.

The keyword **inline** tells the compiler to insert code directly whenever possible, in order to avoid the overhead of a function call. In all other ways (scope, and so forth) an inline function behaves like a normal function. Its use is recommended over **#defined** macros (except, of course, where you depend on the macro-substitution tricks of the preprocessor). This feature is intended for simple, one-line functions.

Program 4

Source

```

// ex4.cpp: Default arguments and Pass-by-reference
// from Hands-on C++
#include <iostream.h>
#include <ctype.h>

int get_word(char *, int &, int start = 0);

main()
{
    int word_len;
    char *s = " These words will be printed one-per-line ";

    int word_idx = get_word(s, word_len);           // line 13
    while (word_len > 0)
    {
        cout.write(s+word_idx, word_len);
        cout << "\n";
        // cout << form("%.s\n", word_len, s+word_idx);
        word_idx = get_word(s, word_len, word_idx+word_len);
    }
}

```

In an important change from C, declarations can appear anywhere a statement can.

It's good programming style to make null loop bodies stand out.

```
int get_word(char *s, int& size, int start)
{
    // Skip initial whitespace
    for (int i = start; isspace(s[i]); ++i)
        ;
    int start_of_word = i;
    // Traverse word
    while (s[i] != '\0' && !isspace(s[i]))
        ++i;
    size = i - start_of_word;
    return start_of_word;
}
```

Output

```
These
words
will
be
printed
one-per-line
```

The prototype for the function **get_word** in the sixth line has two special features. The second argument is declared to be a *reference* parameter. This means that the value of that argument will be modified in the calling program (this is equivalent to **var** parameters in Pascal, and is accomplished through pointers in C). By this means, the variable **word_len** is updated in **main**, and yet we can still return another useful value with the function **get_word**.

One exciting feature of C++ is the **default argument**.

The third argument is a *default* argument. This means that it can be omitted (as in line 13), in which case the value of 0 is passed automatically. Note that the default value need only be specified in the first mention of the function. Only the trailing arguments of a function can supply default values.

Object support

The world is made up of things that both possess *attributes* and exhibit *behavior*. C++ provides a model for this by extending the notion of a structure to contain functions as well as data members. This way an object's complete identity is expressed through a single language construct. The notion of object-oriented support then is more than a notational convenience—it is a tool of thought.

Program 5

Suppose we want to have an online dictionary. A dictionary is made up of definitions for words. We will first model the notion of a definition.

You'll need to compile DEF.CPP to an OBJ file, then link it in with either EX6.CPP or EX7.CPP (or load EX5.PRJ).

You might also want to compile it with Debug Info checked so you can step through and watch the program flow.

```
// def.h: A word definition class
// from Hands-on C++
#include <string.h>

const int Maxmeans = 5;

class Definition
{
    char *word; // Word being defined
    char *meanings[Maxmeans]; // Various meanings of this word
    int nmeanings;

public:
    void put_word(char *);
    char *get_word(char *s) {return strcpy(s,word);}; // line 15
    void add_meaning(char *);
    char *get_meaning(int, char *);
};
```

In traditional C style, we put definitions in an include file. The keyword **class** introduces the object description. By default, members of a class are private (though you can explicitly use the keyword **private**), so in this case the fields in lines 9 through 11 can only be accessed by functions of the class. (In C++, class functions are called *member functions*.) To make these functions available as a user interface, they are preceded by the keyword **public**. Note that the **inline** keyword is not required inside class definitions (line 15).

In other object-oriented languages, classes are often called **objects**, and member functions are called **methods**.

The implementation is usually kept in a separate file:

```
// def.cpp: Implementation of the Definition class
// from Hands-on C++
#include <string.h>
#include "def.h"

void Definition::put_word(char *s)
{
    word = new char[strlen(s)+1];
    strcpy(word,s);
    nmeanings = 0;
}

void Definition::add_meaning(char *s)
```

```

    {
        if (nmeanings < Maxmeans)
        {
            meanings[nmeanings] = new char[strlen(s)+1];
            strcpy(meanings[nmeanings++],s);
        }
    }

char * Definition::get_meaning(int level, char *s)
{
    if (0 <= level && level < nmeanings)
        return strcpy(s,meanings[level]);
    else
        return 0; // line 27
}

```

The *scope resolution operator* (`::`) informs the compiler that we are defining member functions for the **Definition** class (it's good practice to capitalize the first letter of a class to avoid name conflicts with library functions). The keyword **new** in line 8 is a replacement for the dynamic memory allocation function **malloc**. In C++, by convention, zero is used instead of NULL for pointers (line 27). Although we didn't do so here, it is advisable to verify that **new** returns a nonzero value.

Source

```

// ex5.cpp: Using the Definition class
// from Hands-on C++
#include <iostream.h>
#include "def.h"

main()
{
    Definition d; // Declare a Definition object
    char s[81];

    // Assign the meanings
    d.put_word("class");
    d.add_meaning("a body of students meeting together to \
study the same subject");
    d.add_meaning("a group sharing the same economic status");
    d.add_meaning("a group, set or kind sharing the same attributes");

    // Print them
    cout << d.get_word(s) << ":\n\n";
    for (int i = 0; d.get_meaning(i,s) != 0; ++i)
        cout << i+1 << ": " << s << "\n";
}

```

Output

```

class:
1: a body of students meeting together to study the same subject

```


- 2: a group sharing the same economic status
- 3: a group, set, or kind sharing the same attributes

Program 6

We can now define a dictionary as a collection of definitions.

From the command line, build `DICTION.OBJ` and `DEF.OBJ` with `EX6.CPP`. From the IDE, use the `EX6.PRJ` project file.

```
// diction.h: The Dictionary class
// from Hands-on C++
#include "def.h"

const int Maxwords = 100;

class Dictionary
{
    Definition *words; // An array of definitions; line 9
    int nwords;

    int find_word(char *); // line 12

public:
    // The constructor is on the next line
    Dictionary(int n = Maxwords)
        {nwords = 0; words = new Definition[n];};
    ~Dictionary() {delete words;};
    void add_def(char *s, char **def); // The destructor; line 17
    int get_def(char *, char **);
};
```

The function **find_word** on line 14 is for internal use only by the **Dictionary** class and so is kept private. A function with the same name as the class is called a *constructor* (line 18). It is called once whenever an object is declared. It is used to perform initializations; here we are dynamically allocating space for an array of definitions. A *destructor* (line 19) is called whenever an object goes out of scope (in this case, the **delete** operator will free the memory previously allocated by the constructor). In order to have an array of member objects (line 11), the included class must either have a constructor with no arguments or no constructor at all (the **Definition** class has none).

```
// diction.cpp: Implementation of the Dictionary class
// from Hands-on C++
#include "diction.h"

int Dictionary::find_word(char *s)
{
    char word[81];
    for (int i = 0; i < nwords; ++i)
```

```

        if (stricmp(words[i].get_word(word),s) == 0)
            return i;
    }
    return -1;
}

void Dictionary::add_def(char *word, char **def)
{
    if (nwords < Maxwords)
    {
        words[nwords].put_word(word);
        while (*def != 0)
            words[nwords].add_meaning(*def++);
        ++nwords;
    }
}

int Dictionary::get_def(char *word, char **def)
{
    char meaning[81];
    int nw = 0;
    int word_idx = find_word(word);
    if (word_idx >= 0)
    {
        while (words[word_idx].get_meaning(nw,meaning) != 0)
        {
            def[nw] = new char[strlen(meaning)+1];
            strcpy(def[nw++],meaning);
        }
        def[nw] = 0;
    }
    return nw;
}

```

We can now use the **Dictionary** class without any reference to the **Definition** class (the output is the same as in the previous example).

Source

```

// ex6.cpp: Using the Dictionary class
// from Hands-on C++
#include <iostream.h>
#include "diction.h"

main()
{
    Dictionary d(5);
    char *word = "class";
    char *indef[4] =
        {"a body of students meeting together to study the same",
        "subject a group sharing the same economic status",

```

```

        "a group, set or kind sharing the same attributes",
        0);
char *outdef[4];

d.add_def(word, indef);
cout << word << ":\n\n";
int ndef = d.get_def(word, outdef);
for (int i = 0; i < ndef; ++i)
    cout << i+1 << ": " << outdef[i] << "\n";
}

```

In the **Dictionary** implementation, we specifically called the **Definition** member functions. Sometimes it is desirable to allow certain functions or even an entire class to have access to the private members of another. We could declare the **Dictionary** class to be a **friend** to the **Definition** class (line 18):

Build LIST.OBJ with EX7.CPP

```

// def2.h: A word definition class
// from Hands-on C++
#include <string.h>

const int Maxmeans = 5;

class Definition
{
    char *word; // Word being defined
    char *meanings[Maxmeans]; // Various meanings of this word
    int nmeanings;

public:
    void put_word(char *);
    char *get_word(char *s) {return strcpy(s, word);};
    void add_meaning(char *);
    char *get_meaning(int, char *);
    friend class Dictionary; // line 18
};

```

The implementation of **find_word** could then access **Definition** members directly (line 5 in the following code):

```

int Dictionary::find_word(char *s)
{
    char word[81];
    for (int i = 0; i < nwords; ++i)
        if (strcmp(words[i].word, s) == 0)
            return i;

    return -1;
}

```

Program 7

One of the key features of object-oriented programming is *inheritance*. A new class can inherit the data and member functions of an existing (“base”) class (the new class is said to be *derived* from the base class). In this program, we define **List**, a base class for processing a list of integers, then derive **Stack**, a class to handle a stack (which is a special kind of list). First, we create the header file:

To try these out, build
LIST.OBJ and EX7.CPP, or use
EX7.PRJ.

```
// list.h: A Integer List Class
// from Hands-on C++
const int Max_elem = 10;

class List
{
    int *list;        // An array of integers
    int nmax;        // The dimension of the array
    int nelelem;     // The number of elements

public:
    List(int n = Max_elem) {list = new int[n]; nmax = n; nelelem = 0;};
    ~List() {delete list;};
    int put_elem(int, int);
    int get_elem(int&, int);
    void setn(int n) {nelelem = n;};
    int getn() {return nelelem;};
    void incn() {if (nelelem < nmax) ++nelelem;};
    int getmax() {return nmax;};
    void print();
};
```

Then we create the source code:

```
// list.cpp: Implementation of the List Class
// from Hands-on C++
#include <iostream.h>
#include "list.h"

int List::put_elem(int elem, int pos)
{
    if (0 <= pos && pos < nmax)
    {
        list[pos] = elem;    // Put an element into the list
        return 0;
    }
    else
        return -1;        // Non-zero means error
}
```

```

int List::get_elem(int& elem, int pos)
{
    if (0 <= pos && pos < nmax)
    {
        elem = list[pos];    // Retrieve a list element
        return 0;
    }
    else
        return -1;          // non-zero means error
}

void List::print()
{
    for (int i = 0; i < nelem; ++i)
        cout << list[i] << "\n";
}

```

And finally we use the new class:

```

// ex7.cpp: Using the List class
// from Hands-on C++
#include "list.h"

main()
{
    List l(5);
    int i = 0;

    // Insert the numbers 1 through 5
    while (l.put_elem(i+1,i) == 0)
        ++i;
    l.setn(i);

    l.print();
}

```

Output

```

1
2
3
4
5

```

Program 8

*Build STACK.OBJ and LIST.OBJ
with EX8.CPP, or use EX8.PRJ.*

```

// stack.h: A Stack class derived from the List class
#include "list.h"

class Stack : public List
{
    int top;

public:

```

```

Stack() {top = 0;};
Stack(int n) : List(n) {top = 0;};
int push(int elem);
int pop(int& elem);
void print();
};

```

To define a derived class, the base class definition must be available, so we include its header file (line 3). Line 5 informs the compiler that the **Stack** class is derived from the **List** class. The keyword **public** states that the public members of **List** should be considered public in **Stack** also (this is what is usually needed). Since the **List** class has a constructor that takes an argument, the **Stack** constructor invokes the **List** constructor directly (line 11). Base class constructors are executed before those of a derived class.

```

// stack.cpp: Implementation of the Stack class
// from Chapter 6 of the User's Guide
#include <iostream.h>
#include "stack.h"

int Stack::push(int elem)
{
    int m = getmax();
    if (top < m)
    {
        put_elem(elem, top++);
        return 0;
    }
    else
        return -1;
}

int Stack::pop(int& elem)
{
    if (top > 0)
    {
        get_elem(elem, --top);
        return 0;
    }
    else
        return -1;
}

void Stack::print()
{
    int elem;
    for (int i = top-1; i >= 0; --i)

```

```

    { // Print in LIFO order
      get_elem(elem,i);
      cout << elem << "\n";
    }
  }
}

```

Note that the public member functions of the **List** class can be used directly, because a **Stack** is a **List**. However, the private members of the **List** portion of a **Stack** object cannot be referenced directly.

```

// ex8.cpp: Using the Stack Class
// from Hands-on C++
#include "stack.h"

main()
{
    Stack s(5);
    int i = 0;

    // Insert the numbers 1 through 5
    while (s.push(i+1) == 0)
        ++i;

    s.print();
}

```

Output

```

5
4
3
2
1

```

Program 9

Sometimes it is convenient to allow a derived class to have direct access to some of the private data members of a base class. Such data members are said to be *protected*.

*Build EX9.CPP, LIST2.OBJ,
STACK2.OBJ, or use EX9.PRJ*

```

// list2.h: A Integer List Class
// from Hands-on C++
const int Max_elem = 10;

class List
{
protected: // The protected keyword gives subclasses
           // direct access to inherited members
    int *list; // An array of integers
    int nmax; // The dimension of the array
    int nelelem; // The number of elements
}

```

```

public:
    List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
    ~List() {delete list;};
    int put_elem(int, int);
    int get_elem(int&, int);
    void setn(int n) {nelem = n;};
    int getn() {return nelem;};
    void incn() {if (nelem < nmax) ++nelem;};
    int getmax() {return nmax;};
    virtual void print(); // line 22
};

```

We can now replace calls to **List's** member functions with direct references to **List's** data in the **Stack** implementation.

```

// stack2.cpp: Implementation of the Stack class
#include <iostream.h>
#include "stack2.h"

int Stack::push(int elem)
{
    if (top < nmax)
    {
        list[top++] = elem;
        return 0;
    }
    else
        return -1;
}

int Stack::pop(int& elem)
{
    if (top > 0)
    {
        elem = list[--top];
        return 0;
    }
    else
        return -1;
}

void Stack::print()
{
    for (int i = top-1; i >= 0; --i)
        cout << list[i] << "\n";
}

```

And then we can try it out:

```

// ex9.cpp: Using the print() virtual function
// from Hands-on C++

```



```

#include <iostream.h>
#include "stack2.h"

main()
{
    Stack s(5);
    List l, *lp;
    int i = 0;

    // Insert the numbers 1 through 5 into the stack
    while (s.push(i+1) == 0)
        ++i;

    // Put a couple of numbers into the list
    l.put_elem(1,0);
    l.put_elem(2,1);
    l.setn(2);

    cout << "Stack:\n";
    lp = &s;           // line 22
    lp->print();       // Invoke the Stack print() method; line 23

    cout << "\nList:\n";
    lp = &l;
    lp->print();       // Invoke the List print() method; line 27
}

```

Output

```

Stack:
5
4
3
2
1

List:
1
2

```

The above example illustrates *polymorphism* (also known as "late binding" or "dynamic binding," which in C++ is accomplished using *virtual functions*). This means that an object's type is not identified until run time. By defining the **print** member function to be **virtual** (see line 22 of "list2.h"), we can invoke the different **print** member functions through a pointer to the base class. In line 22 above, *lp* points to a **Stack** object (remember: a **Stack** is a **List**), so the **Stack** print method is invoked in line 23. Likewise, the **List** print member function is executed in line 27.

Summary

There is much more to C++ than this chapter covers (multiple inheritance, for example). This chapter is intended give you a sense of the “look and feel” of C++, to show how it differs from C, and to demonstrate how to use most of the basic features of C++. For more information on the basic concepts of C++, read or review Chapter 4, “Object-oriented programming with C++;” Chapter 13, “C++ specifics,” give more advanced material on C++.

Debugging in the IDE

Because C++ is complex subject, this tutorial concentrates just on C. We have included some elementary C++ debugging information.

In previous chapters, you learned the major elements of C. Through a variety of examples, you learned how these elements are put together to create working programs. With that knowledge, you've probably already written your own short programs. If not, now is a good time to begin, because there is nothing like writing your own code to test and extend your understanding of the concepts we have been discussing.

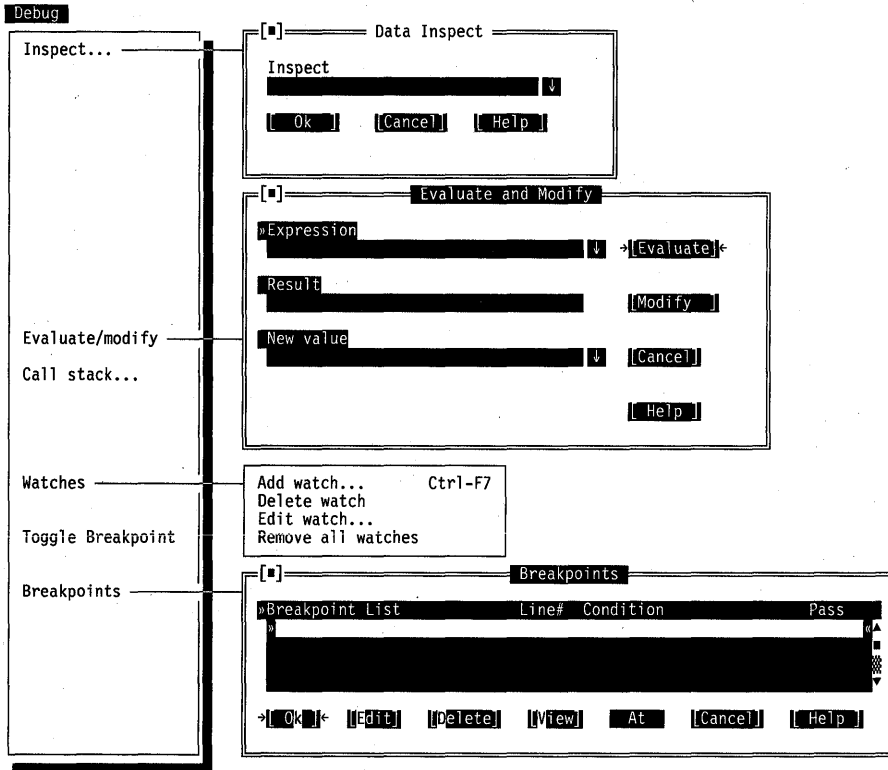
Of course, writing programs means dealing with your mistakes. Most experienced programmers agree that tracking down logical problems, commonly called *bugs*, in programs is a major part of program development. In fact, debugging, or finding and fixing bugs, can often take longer than writing the program itself.

Turbo C++ comes with an integrated, source-level debugger that provides many capabilities of a standalone debugger, while giving you the convenience and speed of remaining within the IDE.

Source level means you can trace through your actual program code—including, if you want, every function that your code calls. By setting breakpoints, you can control how much of your program is run before you examine its condition. You can find out the current value of any variable by selecting it with the cursor or typing its name in the Evaluate field. You can set watches that monitor one or more variables and display their changing values as the program runs. All of this is done without your having to stop thinking in C (or C++), since you use the same variables,

expressions, and operators you have been using to write your program.

Type **TC** to start Turbo C++ (if it isn't already running), and take a few minutes to examine the **Debug** menu. Highlight each item on the menu in turn, pressing **F1** each time to view the associated help text. These menu options are the debugging tools that you'll learn to use while developing this chapter's example program.



Debugging and program development

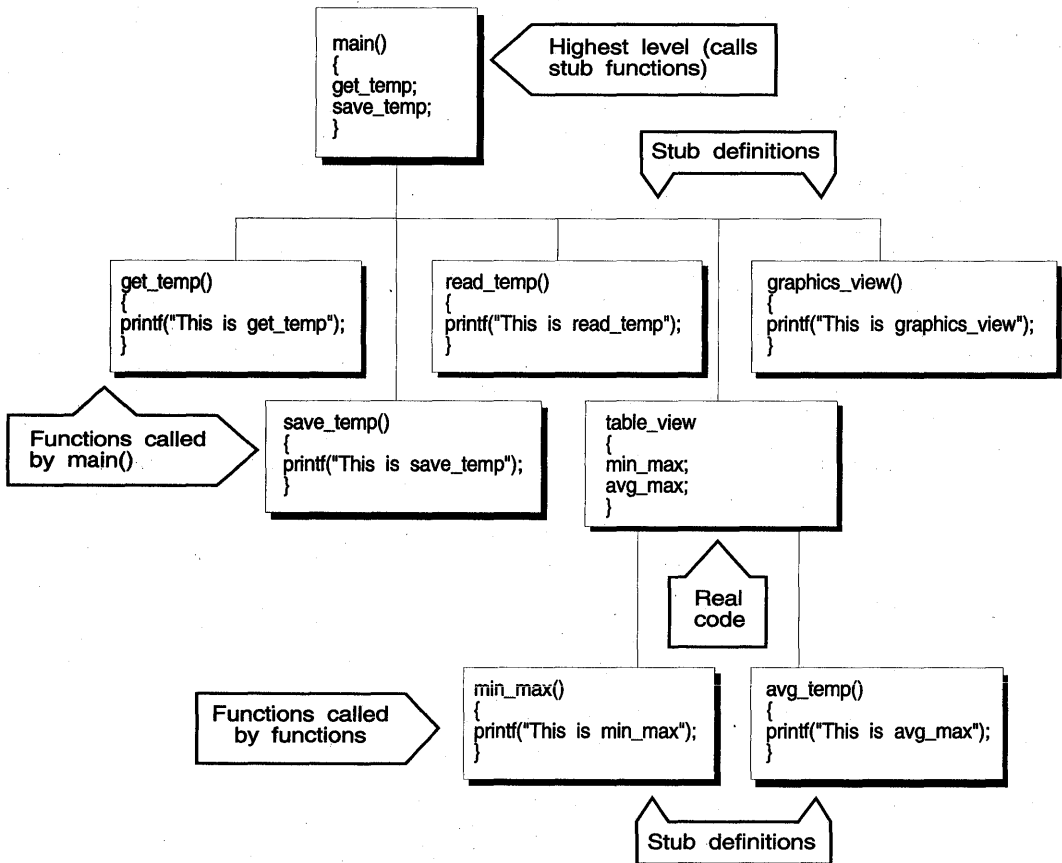
While this chapter, **PLOTEMP.C**, focuses on debugging, it's important not to treat debugging as something separate from designing and writing a program. Turbo C++'s integrated debugger makes the mechanics of finding and fixing bugs as easy as possible, but the way you design your program can help make debugging easier, too.

Consider the old tube televisions. They were hard to repair because all their parts were wired into one big mass of tubes, resistors, capacitors, and so on. This often made it hard to find out which part or parts were not working. When you found the part that needed to be replaced, it was hard to get at that one part in the mass of wires.

After transistors and integrated circuits came into use, people started designing TVs differently. Each circuit came on its own slide-in module, which could slide out for easy access and testing. To replace a defective module, you simply slid in a replacement.

The C equivalent of these plug-in modules is the *function*. You'll design, implement, and test the example program in this chapter, one function at a time. It is often tempting to write a whole program, particularly if it is a small one, and only then start debugging. Like the old TV, though, this makes things unnecessarily difficult. To see why, suppose **main** calls a function **b**, which in turn calls functions **c** and **d**. If you don't test each of these functions as you develop them, it is hard to tell whether a possible problem in function **d** is due to a coding error there, or to one in **b** or **c**. Of course, there is often more than just one bug. Meanwhile, a seemingly unconnected function **a** may have modified global data that **b** needs to pass to **c**. You can see that incremental program development—developing and testing one part of your program at a time—can save you considerable time and frustration.

Figure 6.1: Program development flowchart



Designing the example program: PLOTEMP.C

The example program for this chapter collects temperature readings and displays them using either a table or graph view. It illustrates a diverse assortment of Turbo C++ capabilities: console and file I/O, passing an array to a function, and some graphics charting functions. Later, if you want, you can modify PLOTEMP.C to work with other kinds of data, and to provide different kinds of reports and charts.

One of the best ways to design a program is to prototype it by showing how it interacts with the user—how it requests information, responds to commands, and displays information. Before looking at the program code, let's see what the program looks like from the user's point of view.

When you run PLOTEMP.C, it displays this menu:

```
Temperature Plotting Program Menu
E - Enter temperatures for scratchpad
S - Store scratchpad to disk
R - Read disk file to scratchpad
T - Table view of current data
G - Graph view of current data
X - Exit the program
Press one of the above keys:
```

If you press *E*, you are prompted for a set of temperature readings. The program is set up to handle a set of eight readings, but you can change this simply by modifying the following line, which is located near the top of the program:

```
#define READINGS 8
```

Suppose you entered the following data:

```
Enter temperatures, one at a time.
Enter reading # 1: 52
Enter reading # 2: 55
Enter reading # 3: 62
Enter reading # 4: 65
Enter reading # 5: 73
Enter reading # 6: 76
Enter reading # 7: 68
Enter reading # 8: 61
```

The PLOTEMP menu is redisplayed, unless you select *X* (for Exit).

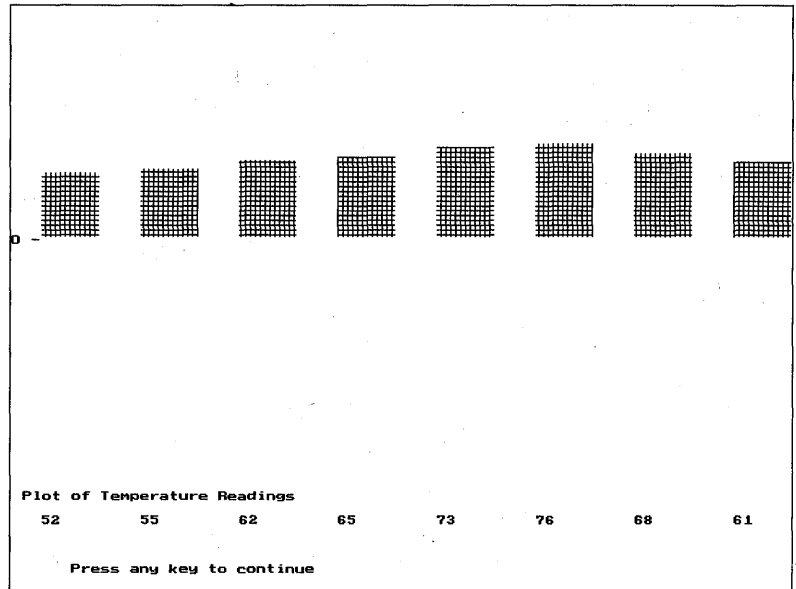
Choosing *S* saves the data set currently in memory to a disk file after you're asked for the file name. Choosing *R* reads a data set from the disk file you specify to the program's "scratchpad" data array.

Once you have data in the scratchpad (either by entering it from the keyboard or reading it from disk), you can display a summary table of the data by choosing *T* (for Table view):

```
Reading      Temperature (F)
1            52
2            55
3            62
4            65
5            73
6            76
7            68
8            61
Minimum temperature: 52
Maximum temperature: 76
Average temperature: 64.000000
```

Alternatively, you can select the graph view shown in the following figure of the current data by choosing *G*:

Figure 6.2
Graph view of temperature
data



There's one problem: The program you'll be putting together won't run entirely as expected. You'll have to use the debugger to find and fix the bugs.

Writing the prototype program

You can load this file right now: File | Open | PLOTEMP1. As you load and run these successive pieces of code, remember that we've deliberately sprinkled them with bugs.

Having decided what the program should do, you can determine what global data and other definitions the program needs, and write the **main** function:

Notice the following important features of this prototype program:

- Global **#defines** and data structures (the array *temps*).
- The function **main** provides the top-level menu.
- Other functions are declared with **void** return and argument type.
- Each function contains only a **printf** statement that identifies it when it is executing.
- The program is already complete enough to execute.

Why are functions given **void** declarations? They let you run the program and check the flow of execution at the top level. If you gave the functions full ANSI prototypes with arguments of various data types, you would have to start implementing the function definitions and writing code to use the values passed to the functions. Otherwise, you would receive compiler warnings about unused parameters. A good rule of program development is “do one step at a time, test one step at a time.” At this point, you want to verify that the top-level structure of the program is sound.

Using the integrated debugger

Compile PLOTEMP1.C by choosing **Compile | Build All**. The compiler halts after displaying the error message:

```
Error C:\TC\EXAMPLES\PLOTEMP1.C 45:  
Statement missing ;
```

Turbo C++ has found a syntax error. You often need to wade through a flock of syntax errors before you can actually run your program, let alone debug it. Fortunately, syntax errors are usually easy to fix. Press the space bar: Here, the error bar highlighted the first line of the **switch** statement that displays the menu. A reference to a missing semicolon usually refers to the preceding statement (in this case, the last in the preceding group of **printf** statements). *Alt-F8* moves you directly to the next error/warning message. As you probably recall, you can press *Enter* or *F6* to switch to the Edit window and make your correction. If you have more than one syntax error to deal with, press *F6* to switch back to the Message window, and then use *Alt-F8* (or the ↓ key) to move to the next error listed.

Highlighting the error/warning message and pressing *F1* displays information about the message.

Now that you’ve fixed the syntax error, compile the program again. This time the compiler and linker run without error messages, which means the program is now free of syntax errors.

Now choose **Run | Run** to run PLOTEMP1.EXE. You’ll see the program’s menu. Try pressing each of the keys listed in the menu—exercise the program. Try both uppercase and lowercase letters. Try letters that aren’t given on the menu at all. What happens in each case? Did everything work as it should?

You probably noticed that with each selection made from the menu, a line describing the function was displayed. (Remember those **printf** statements in the stub functions in the listing?) Usually, the menu was then redisplayed. If you press *X* (or *x*), the program exits, and you are returned to Turbo C++. What happens when you press *G* (or *g*), though? You are also returned to Turbo C++. That's not right — you should get the menu again. There's a bug lurking somewhere.

Tracing the flow of a program

By using options on the **Run** menu and observing the run bar, you can observe the order of execution of your program's statements and control how detailed the tracing is.

Choose **Run | Trace Into** (or press *F7*). The debugger scrolls the beginning of function **main** into the Edit window and highlights it. This highlight is called the *run bar* and marks the *execution position*, indicating the next statement to be executed.

Tracing high-level execution

To trace the high-level flow of your program, choose **Run | Step Over** (or press *F8*), and the next line containing code is executed (comment lines are skipped over). As you continue pressing *F8*, the run bar moves through the series of **printf** statements that display the menu. The screen flickers, because each time the debugger executes a statement that displays information onscreen or executes a function call, it momentarily switches to the User screen. This is the screen display your program would generate if you weren't executing it within Turbo C++'s IDE, but the screen switching happens too fast for you to see the output. To look at the User screen, choose

Window | User Screen, or press *Alt-F5*. Depending on how far you've gotten in executing the **printf** statements, you'll see part or all of PLOTEMP's menu. Press any key to switch back to the debugger.

Continue pressing *F8* until you reach the first line of the **switch** statement. This line contains a call to the **getche** function, which requires the user to input a character. Any time user input is requested by the program, Turbo C++ switches to the User screen. Since you want to observe what happens when the graph view

(G) option is selected, press *F8* again, then press *G*. After you supply the requested input, the display switches back to the debugger, and the run bar moves to the following statement:

```
case 'G': graph_view();
```

So far, so good. The next time you step, however, the following statement is `exit`, and you are back in the editor the next time you step. Press *F8* to verify that the program finishes. By now, you've probably noticed that a **break** statement is needed on the preceding line. Correct the line so that it reads:

```
case 'G': graph_view(); break;
```

Now let's see if the fix worked. You could start the program and step one line at a time all the way from the beginning, but that would be tedious. Instead, choose **Run | Go to Cursor** (or press *F4*; make sure the cursor is on the correct line). Rebuild the program (say yes to the "OK to rebuild" prompt); its lines execute until user input is needed. Press *G* in response to PLOTEMP's menu; execution continues up to the line you just fixed. Continue stepping. Notice that this time the **break** statement is executed, and the execution position then goes back to the top of the **while** loop. PLOTEMP's menu now appears to be operating correctly.

Tracing into called functions

When you use **Run | Step Over**, only the highest level of your program is stepped through. As you saw, the run bar stayed within the **main** function, stepping through the **printf** statements and the various **cases** within the **switch** statement. Often, however, you need to trace into the functions called by your **main** function, and sometimes the functions called by the functions. To trace through function calls, choose **Run | Trace Into**, or press *F7*.

Try this now: Trace through the program, and press *E* at the PLOTEMP menu. This time, the run bar goes to the appropriate **case** in the **switch** statement, and then goes into the definition of the **get_temps** function. After stepping through the function (right now, there's just a **printf** statement there), execution falls to the bottom of the big **while** loop in **main**, then returns to the top where the menu is redisplayed.

Continuing program development

Through the rest of this chapter, you add one function at a time to PLOTEMP and then testing it by running the program again. Instead of making you do all the work of typing in code, we've provided incremental versions of PLOTEMP that you can load to complete each step. If you were debugging one of your own programs, you would follow these steps:

1. If necessary, replace the function prototype with the complete one.
2. Replace the stub function definition with the actual code needed to perform the task.
3. Test the new function by running the program again, making the appropriate choice from its menu.
4. Fix any bugs that crop up, testing until there are no more bugs.
5. Implement the next function in the same way, until the fleshed-out program is complete.

For best and fastest results, write, compile, run, and debug each function separately. Don't start developing the next function until you have a working program that has no apparent errors. This strategy won't eliminate all bugs, because "hidden" bugs sometimes continue to lurk, waiting for some unforeseen combination of circumstances. But this incremental approach minimizes the chance of unexpected crashes.

Start with the **get_temps** function, which gets a set of temperature readings from the keyboard. Since it doesn't take any arguments, and doesn't return anything directly, the following prototype declaration doesn't need to be changed:

```
void get_temps(void)
```

A professional programmer probably would have every function return a value, so errors could be detected. In our example program, return values are omitted to reduce the size and complexity of the example.

Find the existing definition of **get_temps**. Right now, it's a stub definition that just prints a message when it executes. Either replace the stub with the following code, or load PLOTEMP2.C, which already includes the correct code:

```

void get_temps(void)
{
    char inbuf[130];
    int reading;

    printf("\nEnter temperatures, one at a time.\n");
    for (reading = 0; reading < READINGS; reading++)
    {
        printf("\nEnter reading # %d: ", reading + 1);
        gets(inbuf);
        sscanf(inbuf, "%d", temps[reading]);

        /* Show what was read */
        printf("\nRead temps[%d] = %d", reading, temps[reading]);
    }
}

```

Setting breakpoints

Remember that arrays in C begin with element 0: The first reading goes into element 0, the second into element 1, and so on.

If **get_temps** works correctly, the **for** loop prompts for each reading, **gets** grabs a string from *inbuf*, **sscanf** stores it in an element of the *temps* array, and the last **printf** statement displays the value stored in the array. After this function is debugged, you can remove the last **printf** statement.

Run PLOTEMP2 and press *E* at the PLOTEMP menu so that **get_temps** executes. As you enter the data, readings of zero are displayed, as shown in the following example:

```

Enter reading # 1: 40

Read temps[0] = 0
Enter reading # 2: 50

Read temps[1] = 0
Enter reading # 3: 55

Read temps[2] = 0
Enter reading # 4: 57

Read temps[3] = 0
Enter reading # 5: 61

Read temps[4] = 0
Enter reading # 6: 64

Read temps[5] = 0
Enter reading # 7: 65

Read temps[6] = 0
Enter reading # 8: 60

Read temps[7] = 0

```

What's happening here? Regardless of what data you enter, the corresponding element of the *temps* array remains set to 0. Somehow the data being entered isn't finding its way into the array.

A breakpoint is a place in your program where you want execution to run to, then stop at.

Clearly, this code needs closer examination. To run the program until a particular area of interest is reached, set a breakpoint. Move the cursor to the beginning of the loop in the **get_temps** function:

```
for (reading = 0; reading < READINGS; reading++)
```

*If you choose **Toggle Breakpoint** for a line that already has a breakpoint, it removes that breakpoint.*

Now choose **Debug | Toggle Breakpoint** (or press *Ctrl-F8*) to set a breakpoint at this line. The line with **for** is highlighted. Any time the flow of execution reaches a line where a breakpoint is set, the program is interrupted, and you're returned to the debugger. Now you can use other debugger commands to examine and change variables and other data structures. When you have several breakpoints in a program, you can choose **Debug | Breakpoints** and select the View button in the dialog box to see the next breakpoint.

Breakpoints stay set until you do one of the following:

- Leave the IDE.
- Toggle the breakpoint off with *Ctrl-F8*.
- Delete the breakpoint using **Debug | Breakpoints | Delete**.
- Delete the line(s) on which the breakpoints are set.
- Edit a file containing breakpoints and abandon the file without saving it.

Any time you correct a bug or otherwise edit the current file, and resume using debugger commands, Turbo C++ asks, "Source modified, rebuild?" Normally you'd press *Y* at this prompt, so the program can be rebuilt into a version that reflects your changes. If you press *N*, both breakpoints and the run bar appears in the wrong places because the source file no longer matches the executable program.

Instant breaking with *Ctrl-Break*

Pressing *Ctrl-Break* allows you to "break out" of many running programs, including Turbo C++ and the integrated debugger, but the debugger doesn't always stop the program instantly. The debugger waits until the machine code corresponding to one of your lines of source code is being executed. It then stops the program at the machine instruction that corresponds to the

beginning of the next source code line. The run bar appears on the line following the last line executed.

To break instantly, press *Ctrl-Break* twice. When the second key depression is detected, the debugger terminates the program immediately, without flushing any output or calling any exit functions. (This is similar to using the `_exit` function.) A “double break” is usually undesirable, since it changes the contents of data files unpredictably, and the debugger doesn’t know which line to execute next. You’ll probably only want to use a second *Ctrl-Break* when your program is “hung” or stuck in an infinite loop.

Run the program again; the PLOTEMP menu is displayed. Press *E*. The program runs into the `get_temps` function until the breakpoint is reached. Step with *F8* until the run bar has moved through the statements in the body of the `for` loop and returned to the first line of the loop. Now it’s time to look at the key variables to learn more about the bug.

Inspecting your data

Complex programs usually use a variety of data structures—arrays, structures, unions, lists, and so on. To understand what is going on in a particular part of your program, you often need to know the actual contents of these data structures. Turbo C++ provides a new facility called *inspectors*. Inspectors let you raise the hood at any part of your program and examine the inner workings.

Inspector windows

You can inspect any legal C or C++ expression, provided it doesn’t contain symbols that were created with `#define` or function calls.

To open an Inspector window for an item, move the cursor to the item in the Edit window and press *Alt-F4*. (You can also choose **Debug | Inspect** to bring up the Inspector dialog box, and type in the variable or expression for inspection.) Try inspecting the variable *reading* from your current version of PLOTEMP2.C (which should currently be in the Edit window). Choose **Run | Trace Into**, then **Debug | Inspect**.

All Inspector windows begin with the name of the item. The line below the variable name contains the address of the variable, expressed as *segment:offset*. (Variables that have been declared to be of **register** type don’t have an address, since they are being stored in the CPU rather than in RAM. Instead, you’ll see the

word *register*.) The next line describes the item's data type (for example, **unsigned int**).

The actual value of the item is displayed to the right of the data type. Turbo C++ automatically selects the appropriate formats for displaying the type of data involved. For nonprinting characters, a backslash (\) followed by the hexadecimal character value is used in place of the character value. An **int** variable, on the other hand, would have decimal and hexadecimal values but no character representation. The other numeric types are handled similarly.

Inspecting arrays and strings

For an array, a separate line is shown in the window for each element. If there are too many elements to fit in the window, use the arrow keys to scroll through them. For a string, a character representation for the string as a whole is also shown. The next figure shows the array *temps* from PLOTEMP.C after it has been filled with data via the **get_temps** function.

Figure 6.3
Inspecting the *temps* array

Index	Value	Hexadecimal
[0]	55	(0x0037)
[1]	58	(0x003A)
[2]	61	(0x003D)
[3]	64	(0x0040)
[4]	69	(0x0045)
[5]	72	(0x0048)

int [6]

The display for strings is similar to that for arrays (they are, after all, the same data structure). With strings, however, the character representation of the string is shown in addition to the individual elements.

Inspecting structs and unions

For structures and unions, the values of the individual members are shown. To see how this works, load SOLAR.C, introduced in Chapter 3, "An introduction to C++", and inspect the *solar_system* array. Since this is an array whose elements are structures of type **planet**, you can browse through the array elements and view the data stored in the members of each **planet** structure.

Inspecting pointers

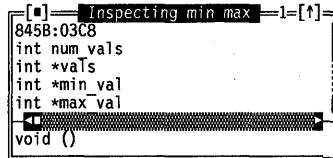
For a pointer, the Inspector window shows the address of the pointer, the address the pointer points to, and the data found at that address, which can be a simple variable, an array, a structure, and so on. Indexes [0], [1], and so on are shown, clearly identifying the position of each piece of data. This program defines a character pointer *ptr*:

```
main()
{
    char * ptr = "This is a string\n";
}
```

Inspecting functions

The Inspector window for a function shows its return type and address, as well as all of the function's parameters. The Inspector window for the function **min_max** from the complete version of PLOTEMP.C is shown in the next figure.

Figure 6.4
Inspecting the min_max
function



You can use function inspectors to review a function's return type and parameters without having to go back to the function declaration.

When should you use inspectors?

Using inspectors may seem like overkill when dealing with simple variables and even arrays, since the **Debug | Evaluate/modify** option (discussed later) also shows the values of variables. Inspectors help you deal with more complex data structures (structures, classes, unions, and arrays of these data types). In general, inspectors are most useful for studying data in depth, while the **Debug | Evaluate/modify** facility is better for taking a quick look at simple data.

Evaluating and changing variables

Load PLOTEMP2.C, compile it, and step to the last **printf** in the function **get_temps**. Select *E* and enter one reading. You should now have executed the contents of the **for** loop once, and received as input, formatted, and stored one reading (via the **gets** and **scanf** functions). Now choose **Debug | Evaluate/modify**, which pops up a window containing three fields:

- the Expression field, which contains the expression you are interested in
- the Result field, which displays the value of the expression in the Evaluate field
- the New Value field, where you can enter a new value for the selected expression

By default, the “word” (C variable, keyword, function call, and so on) at the cursor is displayed in the Evaluate field. You’re interested in two variables: *reading*, which is the loop’s counter variable, and the array *temps*, which is supposed to contain the input data.

Type *reading* and press *Enter*. The debugger now displays 0 in the Result field. If you step through the statements in the loop and evaluate *reading* again, you’ll find its value is now 1.

You can also examine the values of more complex data structures, such as arrays, strings, and structures. Here, you want to know more about what’s happening to the array *temps*. Go ahead and type *temps* in the Evaluate field. The following is displayed:

```
{0, 0, 0, 0, 0, 0, 0, 0, 0}
```

This shows the integer values currently stored in the array *temps*. You can get a single value by using an index: Specifying *temps[0]* would get you the first integer, for example. Or you could open an Inspector window.

Notice that we have been referring to expressions, not just values (such as variables) by themselves. Recall that an expression is a combination of variables, constants, and operators that yields a single value; for example, *vals1[index] + vals2[index] + 1*. You can display the value of any expression, provided that

- It doesn’t involve a function call (so an expression such as $\text{sqrt}(a) + 1$ can’t be used).

- It doesn't use a **#define** value, such as `READINGS` in the current program.

For practice, get the values of the following expressions by typing them into the Evaluate field:

```
reading + 2
temps[reading + 1]
```

Specifying display format

Optionally, you can add a comma and a format specifier to the value you want displayed. For example, type `reading,h` to see the current value of `reading` in hexadecimal (you could also type `reading,x`). By default, integers are displayed as decimal, and character arrays are displayed as strings.

The specifier **m** is useful when dealing with arrays: It displays a memory dump starting at the specified address. For example, `temps,m` gets you a memory dump starting at the location specified. Since `temps` is an array, its name points to the starting address of the following stored data:

```
00 00 00 00 00 00 00
```

All the elements of the array `temps` are currently set to 0. The number of elements displayed depends on the size of the array. You can combine **m** with other specifiers:

```
temps,mh
```

displays a memory dump in hexadecimal format.

Another useful specification is **p**, which displays the selected variable as a pointer, giving information about the area of memory being pointed to (for example, the Interrupt Vector Table, the BIOS data area, or the user program's program segment prefix [PSP]). If the memory pointed to is within the program's own allocated memory, the name of the variable (if any) at the address of the segment offset is also displayed.

Specifying the number of values

When dealing with an array, you can specify how many values you want to display: `temps,5` specifies the first five elements of the array `temps`. Format specifiers can be combined with numerals. For example, `temps[2],3h` specifies that the three elements

following the third element of the *temps* array should be displayed in hexadecimal format.

Other format specifiers and more details of the debugger commands are given in the online help.

Copying from the cursor position

As noted earlier, the Evaluate field contains whatever word was at the cursor position when you chose **Debug | Evaluate/modify**. Take advantage of this to save typing. For example, if you move the cursor to the beginning of the expression

```
temps[reading]
```

temps appears in the Evaluate field. As you press →, the characters following the word *temps* appear, letting you copy the complete expression `temps[reading]` into the Evaluate field. Press *Enter* as usual to display the value.

Specifying variables in other functions

Right now, you're looking at the variables *reading* and *temps* in the function **get_temps**. You can also ask the debugger for the values of static variables in other functions because static variables retain their values even when the function that uses them isn't being executed. You can also look at variables in the functions that called the one you're executing. You can't look at ordinary automatic variables declared in other functions, because they no longer have a value when their function exits. The context in which expressions are evaluated is given by the current cursor position in the Edit window.

To specify a variable outside of the current function, you can move the cursor into the body of the function, or you can give the name of the function, a period, and the name of the variable.

If the variable you want to inspect is in another program module, you must specify the module name first; for example,

```
module2.getvals.count
```

Changing values

Now you know how to look at different kinds of variables and expressions, and how to see their values in different formats. Practice stepping through the **for** loop in **get_temps** and examine

the values of *reading* and *temps[reading]*. The latter insists on always being 0.

Try evaluating the expression *temps[reading]* as you step a few times through the loop. The debugger displays its value as 0, regardless of the value of *reading*. But what should this expression represent? Since it is specifying the storage destination for the **sscanf** function, it needs to be an address. This means that all of the values entered are being stored at address 0! You can confirm this by using the pointer format *temps[reading],p* and finding that that value is still 0.

Use the address operator **&** to make this expression refer to the address of the *temps* array. Evaluate *&temps[reading],p*. The result looks something like this: DS:1278 *temps+1*. The actual values displayed depend on your system configuration and the current value of *reading*, but you can see that *&temps[reading]* points to a bona fide address in the data segment, with an offset from the address pointed to by the variable *temps*.

Change the expression *temps[reading]* in the **sscanf** statement to *&temps[reading]*. If you now continue stepping through the program, you are asked whether to rebuild the program; press Y for Yes. Now, if you step through the **for** loop again and evaluate *temps[reading]*, you'll find that the values you are entering are stored correctly in the array.

This is a good time to practice changing values with the debugger: Evaluate the current value of *temps[reading]*.

Use the *Tab* key to move among the Evaluate, Result, and New Value fields. Once the input cursor is in the New Value field, type a value, such as 66, and press *Enter*. If you type *temps[reading]* in the Evaluate field, its new value, 66, is shown in the Result field. You can change the value of any expression that represents a single data element, such as a simple variable, a pointer, or an array element.

Changing values interactively with the debugger is useful for temporarily fixing a bug and continuing program execution a little further, looking for the next bug. Here, for example, you could put new values in *temps[0]* through *temps[7]*, set *reading* to 8 to break out of the **for** loop, and return to the program's main menu. You can also force a function to return a specific value, or to pass that value to another function. This lets you test unusual conditions that might lead to bugs, without having to put temporary assignment statements in your code.

The `get_temps` function should now be working correctly. Next, implement the `table_view` function, so you can view the entered data. Either replace the stub code for `table_view` with the following code, or load `PLOTEMP3.C`, which contains the correct code:

Load `PLOTEMP3.C`.

```
void table_view (void)
{
    int reading;

    clrscr();          /*clear the screen */
    printf("Reading\t\tTemperature (F)\n");

    for (reading = 0; reading <= READINGS; reading++)
        printf("%d\t\t\t%d\n", reading + 1, temps[reading]);

    min_max();
    printf("Minimum temperature: \n");
    printf("Maximum temperature: \n");
    avg_temp();
    printf("Average temperature: \n");
}
```

This function prints the table headings and then uses a **for** loop to obtain and print the values stored in the `temps` array. Eventually, it prints the minimum, maximum, and average temperatures. These functions haven't been implemented yet, so they'll just print out a message saying that they are being executed.

Notice that including the called functions before they are implemented lets you test the program flow, and reminds you of the program structure. This kind of design is often called *top-down* because the program is designed at the top level first (the **main** function). You are in the process of designing the functions called directly by **main**, such as `table_view` here. When the top level of `table_view` is working, you'll then implement the functions it calls—`min_max` and `avg_temp`.

Now build and run `PLOTEMP3.C`, choose *E*, and enter test data (we entered 10, 20, 30, 40, 50, 60, 70, and 80). When you are back at the menu, choose *T* (Table view), and you'll see something like the following information on the screen:

Reading	Temperature (F)
1	10
2	20
3	30
4	40
5	50
6	60
7	70
8	80
9	0

```

Executing min_max().
Minimum temperature:
Maximum temperature:

```

```

Executing avg_temp().
Average temperature:

```

Monitoring your program by setting watches

Well, you entered eight readings and got back nine! The last one had a value of 0. You probably suspect the infamous “one off” bug—getting one less or one more iteration of a loop than you had expected.

First localize the problem by setting a breakpoint at the first line of the **for** loop in **table_view** by moving the cursor to that line and choosing **Debug | Toggle Breakpoint** (or press **Ctrl-F8**).

Now run the program again, enter the test data, and choose Table view. The program stops at the breakpoint in **table_view**; now you can see what’s going on in the **for** loop.

So far, you have obtained the values of variables by stepping through the program and using **Debug | Evaluate/modify** to inspect their values. This is fine when you just need to inspect the values once, but when you’re dealing with loops or repeated function calls, you also want to see how the values change. It would be very tedious to evaluate the variables by hand repeatedly. The debugger lets you monitor these changing values automatically by setting watches. A *watch* is an expression whose value is updated each time it is encountered in the running program.

Adding a watch

You are interested in two variables in this case: *reading*, which is incremented repeatedly by the **for** loop, and *temps[reading]*, which holds the value being printed each time through the loop. Since the cursor is already nearby, the easiest way to set these watches is to move the cursor to the name of the variable you want to

watch, and then choose **Debug | Watches | Add Watch** (or press *Ctrl-F7*). Move the cursor to *reading* and try this; you'll see the pop-up window. As with **Debug | Evaluate/modify**, the default name shown is the one at the cursor; simply press *Enter*. Use the same procedure to set a watch for *temps[reading]*. The pop-up window shows *temps*, but as with **Debug | Evaluate/modify**, you can use *→* to copy the rest of the expression into the window, and then press *Enter* to set the watch.

Watching your watches

Now that you have set two watches, the variable name and value for each watch is shown in the Watch window:

```
reading: 177
temps[reading]: 92
```

Since the loop hasn't been run yet, the values shown (which may be different on your system) are meaningless, representing whatever happens to be at the respective memory locations.

Now start stepping through the loop (with *F8*). As you step, notice how the values change. After the first time through the loop, the values are

```
reading: 0
temps[reading]: 10
```

assuming you entered your test data starting with 10 as described earlier. The next time through the loop, the watches display

```
reading: 1
temps[reading]: 20
```

The last time the loop is executed, the values are

```
reading: 8
temps[reading]: 0
```

This suggests that the loop exits only after *reading* reaches 8. When should it exit? Since there are eight readings entered, and *reading* starts at 0, the last value it should have during the loop is 7, not 8. Now take a look at the loop exit condition:

```
reading <= READINGS
```

Checking your **#defines** at the beginning of the program, you see that **READINGS** is 8. Do you see the problem? To exit when *reading* is 7 (after processing eight readings), the condition should

read reading < READINGS. Change <= to <, and rerun the program to see if it works correctly.

Controlling the debugger windows

This also works within the Debug | Evaluate/modify window.

If you have set more than a few watches, there won't be room to see them all at once. You can scroll the Watch window using the *PgUp* and *PgDn* keys, or move one line at a time with the *↑* and *↓* keys.

If a particular watch expression is too long to fit in the window, you can see its beginning and end by scrolling it with the *Home*, *End*, *←*, and *→* keys.

Another way to see more is to zoom a window.

Remember that you can look at an entire screen of the program output at any time by pressing *Alt-F5*. Press any key to return to the environment.

Now is a good time to practice using these features.

Editing and deleting watches

Debug | Watches | Edit Watch changes only the expression itself, not its value. To change the value, use Debug | Evaluate/modify.

It's easy to edit, add, or delete watches. When the Watch window is selected, the highlighted expression is active. To highlight a different expression, use the *Home*, *End*, *↑*, or *↓* keys.

To edit (change) the currently highlighted watch, you can choose **Debug | Watches | Edit Watch**, or as the bottom line of the screen says, you can simply press *Enter*. The debugger opens a pop-up window with the selected expression, and you can edit it. Practice by changing the watch for `temps[reading]` to `temps[reading+1]`.

You already know how to add watches, but once the Watch window is active, there's an easier way: Press *Ins*. A pop-up window appears. You can type in the watch expression, add to it with the *→* key, or accept the default that was copied from the cursor position.

To delete the current watch, choose **Debug | Watches | Delete Watch**, or simply press *Del*. Practice by deleting the watch for *reading*. You can delete all of the watches by choosing **Debug | Watches | Remove All Watches**.

Finding a function definition

Now that PLOTEMP.C is starting to flesh out, it's harder to find the function you want to examine. The debugger provides a way to scroll the Edit window to a specified function definition.

Choose **Search | Locate Function**, and Turbo C++ opens a dialog box. Practice by typing `get_temps` (don't type the parentheses after the function name, or the debugger won't find your function). The definition of **get_temps** is now displayed in the Edit window. This is useful for reviewing the definition of a function, as well as for finding locations to set breakpoints and watches.

Note that **Search | Locate Function** works only with functions that have source code in a file that has been compiled with debug information. Library functions such as **printf** can't be found with **Search | Locate Function**, since their source code isn't available in the IDE.

Finding out who called whom

In a complex program, there may be several levels of function calls, and you might not remember the calling order of functions before the program reached the breakpoint. The debugger can help you. Set a breakpoint that halts the program at the place where you want to see the call sequence. For practice, set a breakpoint at the **printf** in **min_max**.

Run the program and choose **Table View** from the menu displayed by the program. The program stops at the breakpoint. Now choose **Debug | Call Stack**. A pop-up window lists all the functions that are waiting to finish execution at this point. The call stack has the most recently called function at the top, which in this case is **min_max**. It was called by **table_view**, which in turn was called by **main**.

*You can always scroll back to the current execution position by choosing the first function in the Call Stack window—in this case, **min_max**.*

You can use the **↑** and **↓** keys to highlight a particular function in the Call Stack window. If you press **Enter**, the Edit window scrolls to show the last line executed in that function. Right now,

- the last line executed in **min_max** is the first line of its definition, since that's where you placed the breakpoint.
- the last line executed in **table_view** was the line containing the call to **min_max**.

- the last line executed in **main** is the line that executed **table_view**; namely, `case 'T': table_view; break;`.

In other words, **min_max** is currently being executed, and **table_view** and **main** are pending completion.

Multiple source files

As you work with longer programs, you'll find the features we've been discussing to be especially useful. Many substantial programming projects consist of several source files. The debugger automatically loads the file needed to fulfill your request into the Edit window. For example, if you use **Search | Locate Function** to find a function that is declared in a source file other than the one in the Edit window, the debugger loads the appropriate source file into the editor. If you've made any changes to the current file, you are first asked if you want to save the changed file to disk. The same thing happens when you use **Debug | Call Stack** to examine the last executed line of a function whose definition is in a different source file.

Although the debugger makes it easy to work with multiple source files, it is good practice to debug only one or two source files at a time. Always test a given "bug fix" before moving on because there is always a chance that your fix did not work, or possibly even introduced new bugs.

Preventive medicine

You'll soon resume the development and testing of `PLOTEMP.C`. To aid in this and future debugging efforts, take a look at some ways to minimize bugs, and look at some common "buggy" situations.

Design defensively

Just as you can avoid accidents by driving your car defensively, you can avoid bugs by designing your program defensively. As you've seen, the design of `PLOTEMP.C` represents an approach to defensive design through top-down programming.

Try to build up your program from functions whose purposes are simple and well-defined. This makes it easier to set up test cases and analyze their results. It also makes your program easier to

read and modify. For example, if PLOTEMP.C combined both table and graph views in the same function, the code could easily become unwieldy.

Try to minimize the number of data elements each function requires and the number of elements it changes to simplify testing, analysis of results, and program readability and modification. This too makes it easier to set up test cases and analyze their results, and to read and modify your program. It also tends to limit the amount of havoc a misbehaving function can cause, letting you run the function several times in a single debugging session. A program designed this way is said to be *loosely coupled*.

Write clearly

Put indentation, liberal comments, and descriptive variable names in your code to clarify it.

Keep code simple. Express complicated operations in many simple statements rather than a few complex ones that show off your knowledge of C's more obscure features.

Don't try to squeeze the last bit of efficiency out of your program when you write it. The most efficient code is often hard to read and debug. Debug and test the program until it's working before trying to improve program performance.

Be alert for opportunities to write multipurpose and reusable functions. Writing and debugging one generalized function is usually easier than writing multiple specialized ones.

Systematic software testing

Before a jet liner takes off, the crew goes through a systematic checklist to ensure that everything is working properly. Following a specific routine reduces reliance on fallible human memory. In the same way, you should work toward a standard approach to software testing: A checklist of steps, developed from experience, helps you create a reliable program.

There is no single "right" way to test a program; your checklist depends on the types of programs you write, your strengths and weaknesses as a programmer, and personal style. The following

checklist is a model that you can build upon until you have a list that works well for your particular needs.

- *Feed the program some input that is simple but not trivial.* Try the unusual—for example, have you tried entering negative temperatures into PLOTEMP? Trace into the code using **Debug | Evaluate/modify** and watch expressions liberally to check the values of data items. Correct the bugs you find, one or a few at a time.
- *Feed the program other sets of data that let you exercise the parts you couldn't test in the preceding step.* If possible, have someone who is unfamiliar with your program interact with it at the keyboard. Common experience shows that programmers have difficulty exercising their own programs properly because they know which values are appropriate and which aren't. If your program is designed to be used by accountants, try to find an accountant.
- *Test every statement in your program.* You might find bugs in surprising places.
- *Put aside the debugger and test the entire program for correct behavior.* If users expect a robust program, test how the program responds to every type of entry error you can imagine.

Test modifications thoroughly

When you modify a program, retest the affected parts thoroughly. You may have to retest parts that haven't changed but are affected by the changes.

If the program is complex, keep a record of the tests you have performed. When you modify the program, this record helps you repeat all the tests whose results could be affected by the change. If the tests involve input files, save the files.

Areas to watch carefully

As you continue to learn C and to develop programs, keep a list of common bugs and check your program for these errors. Here are some common C programming pitfalls:

- making out-of-bounds errors
- confusing addresses versus values at those addresses
- placing the increment and decrement operators incorrectly
- not testing statements thoroughly

- using Pascal syntax instead of C

The following section discusses each of these pitfalls.

Give special attention to boundary conditions—conditions that make a program escape from a loop, fill an array, and so on. Bugs are often manifested as failures to handle boundary conditions. You've already seen how the condition `reading <= READINGS` caused one too many values to be displayed. Other problems could be caused by starting a loop-counter at 1 instead of 0.

Always be careful about whether you are specifying an address or a value at that address. For example, don't confuse the value `temps[reading]` with the address `&temps[reading]`.

Be careful with the increment and decrement operators `++` and `--`. Is the value being incremented before or after it is used?

Be alert for individual statements or expressions that must be tested more than one way, like these:

```
switch ( strcmp(a,b) ) ...
```

strcmp can return three values: 0 (*a* equals *b*), -1 (*a* is less than *b*), or +1 (*a* is greater than *b*). This suggests that you should test the statement with three sets of input values to verify that **strcmp** works correctly in each case.

The following statement contains an "implicit if" that can produce two different results:

```
x = (x>0) ? func(x) : 0 ;
```

Finishing PLOTEMP.C

You've installed and tested the prototype PLOTEMP.C and have implemented, tested, and debugged the **get_temps** and **table_view** functions. You've learned how to use the debugger features. The completion of PLOTEMP.C involves the following exercises:

- Either replace the stub code for a particular function with the code given previously in this chapter, or load the next version of PLOTEMPX.C, which already has the correct code.
- Change the function prototype (if necessary).

- Test the function implementation using appropriate debugger facilities.
- Find and fix the bugs.
- Move on to the next function.

Finishing table_view

Load PLOTEMP4.C. Remember that we have included deliberate bugs in the PLOTEMP programs so you can practice your debugging skills.

To finish this function, implement the following functions:

```
void min_max (int num_vals, int vals[], int *min_val, int *max_val)
{
    int reading;

    *min_val = *max_val = vals[0];

    for (reading = 1; reading < num_vals; reading++)
    {
        if (vals[reading] < *min_val)
            *min_val = &vals[reading];
        else if (vals[reading] > *max_val)
            *max_val = &vals[reading];
    }
}

float avg_temp(int num_vals, int vals[])
{
    int reading, total = 1;

    for (reading = 0; reading < num_vals; reading++)
        total += vals[reading];

    return (float) total/reading; /* reading equals total vals */
}
```

Since these functions have parameters and return values, the following corrected prototypes are required:

```
void min_max (int num_vals, int vals[], int *min_val, int *max_val)
float avg_temp(int num_vals, int vals[])
```

Finally, change **table_view** so that the return values from the functions are used properly. The revised **table_view** should read as follows:

```
void table_view (void)
{
    int reading, min, max;

    clrscr(); /* clear the screen */
    printf("Reading\t\tTemperature(F)\n");
```

This change is included in PLOTEMP4.C.

This change is included in PLOTEMP4.C


```

    for (reading = 0; reading < READINGS; reading++)
        printf("%d\t\t\t%d\n", reading + 1, temps[reading]);

    min_max(READINGS, temps, &min, &max);
    printf("Minimum temperature: %d\n", min);
    printf("Maximum temperature: %d\n", max);
    printf("Average temperature: %f\n", avg_temp(READINGS, temps));
}

```

Try some debugging on your own by checking the following normal operations:



- Are the loops working properly?
- Are the arithmetic operations appropriate?
- What do the comparisons compare?

Implementing graph_view

Recall that the **graph_view** function creates the chart shown in Figure 6.2. To implement this function, replace its definition with the following code. Add `#include <graphics.h>` at the beginning of your code, and change the third parameter of **initgraph** to the path of the BGI files.

Load PLOTEMP5.C.

```

void graph_view(void)
{
    int graphdriver = DETECT, graphmode;
    int reading, value;
    int maxx, maxy, left, top, right, bottom, width;
    int base;
    int vscale = 1.5; /* zero x-axis for graph */
    int space = 10; /* value to scale vertical bar size */
    /* spacing between bars */

    char fprint[20]; /* formatted text for sprintf */

    initgraph(&graphdriver, &graphmode, "");
    if (graphresult() < 0) /* make sure initialized OK */
        return;

    maxx = getmaxx(); /* farthest right you can go */
    width = maxx / (READINGS + 1); /* scale and allow for spacing */
    maxy = getmaxy() - 100; /* leave room for text */
    left = 25;
    right = width;
    base = maxy / 2; /* allow for neg values below */

    for (reading = 0; reading <= READINGS; reading++)
    {
        value = (temps[READING]) * vscale;
        if (value > 0)

```

```

    {
        top = base - value;    /* toward top of screen */
        bottom = base;
        setfillstyle(HATCH_FILL, 1);
    }
    else
    {
        top = base;
        bottom = base - value; /* toward bottom of screen */
        setfillstyle(WIDE_DOT_FILL, 2);
    }
    bar(left, top, right, bottom);
    left += (width + space); /* space over for next bar */
    right += (width + space); /* right edge of next bar */
}

outtextxy(0, base, "0 -");
outtextxy(10, maxy + 20, "Plot of Temperature Readings");
for (reading = 0; reading < READINGS; reading++)
{
    sprintf(fprint, "%d", temps[reading]);
    outtextxy((reading *(width + space)) + 25, maxy + 40, fprint);
}

outtextxy(50, maxy+80, "Press any key to continue");
getch();                /* wait for a key press */
closegraph();
}

```

save_temps and read_temps

The function **save_temps** saves the current “scratchpad” (the contents of the array *temps*) to a disk file. By now, you should be familiar with the logic involved in accessing elements of this array.

Replace the stub definition for the **save_temps** function with the following code:

Load *PLOTTEMP5.C*.

```

void save_temps(void)
{
    FILE * outfile;
    char file_name[40];

    printf("\nSave to what filename? ");
    while (kbhit()); /* "eat" any char already in keyboard buffer */
    gets (file_name);
    if ((outfile = fopen(file_name,"wb")) == NULL)
        perror("\nOpen failed! ");
}

```

```

        return;
    fwrite(temps, sizeof(int), READINGS, outfile);
    fclose (outfile);
}

```

The function **read_temps** is the counterpart to **save_temps**; it reads values from a disk file into the *temps* array. Implement **read_temps** by replacing its stub definition with the following code:

*This change is also included
in PLOTEMP5.C.*

```

void read_temps(void)
{
    FILE * infile;
    char file_name[40] = "test";

    printf("\nRead from which file? ");
    gets(file_name);

    while (kbhit()); /* "eat" any char already in keyboard buffer */
    if((infile == fopen(file_name,"rb")) == NULL)
        perror("\nOpen failed! ");

    fread(temps, sizeof(int), READINGS, infile);
    fclose (infile);
}

```

After you're finished with **read_temps**, you should have a complete, working version of PLOTEMP.C. PLOTEMP6.C is a bug-free version of this program.

Answers to debugging exercises

Here are the bugs in the remaining functions of PLOTEMP5.C.

min_max and
avg_temps

In **min_max**, the **if** statements assign the value *&vals[reading]*, which is an address to the *min* or *max*, rather than the correct value *vals[reading]*. Also, in **avg_temp**, the variable *total* should be 0. Having it start as 1 adds 1 to the total of the readings, thereby giving an incorrect average.

When **table_view** calls **min_max**, pointers to *min_val* and *max_val*, rather than local values, are passed.

graph_view

There are two obscure bugs in the **for** loop of this function:

```
for (reading = 0; reading <= READINGS; reading++)
{
    value = temps[READINGS] * vscale;
```

The value being read is *temps*[READINGS]. The constant READINGS must be changed to the variable *reading*. This error causes the program to graph the nonexistent element *temps*[8]. Also, the condition `<= READINGS` should be `< READINGS` to read the correct number of values. The first bug masks the second. Often you can't detect a given bug until you've fixed another bug, since the first bug prevents proper execution of the code containing the second bug.

save_temps

Here, the problem is not in the first line of the **for** loop, but rather in the body:

```
if ((outfile = fopen(file_name, "wb")) == NULL)
    perror("\nOpen failed! ");
return;
```

Braces are needed to place both the **perror** statement and the **return** statement under the jurisdiction of the **if**; otherwise, the function prints "Open failed" and it returns, even if the file was opened correctly. The compiler warns, "Unreachable code in function *save_temps*", because the line following the **return** statement can't ever be executed.

read_temps

When you compiled **read_temps**, you saw the compiler warnings "Possible use of *infile* before definition". Consider the following code:

```
if ((infile == fopen(file_name, "rb")) == NULL)
```

When you open a file, the file handle *infile* should be getting a value from **fopen**; the value is tested to see if it is **NULL**, which indicates **fopen** failed. Why isn't *infile* getting a value right away?

The reason is that `==`, rather than `=`, follows *infile*. Since `==` indicates a comparison rather than an assignment, *infile* isn't getting a value.

Advanced options

The Run | Program Reset command stops the current debugging session, releases memory your program has allocated, and closes any open files your program was using. This command cancels a debugging session, allows you to transfer programs when there's not enough memory, and invokes a DOS shell.

The Run | Arguments command lets you give running programs command-line arguments as if you'd typed them on the DOS command line. DOS redirection commands are ignored. If you're already debugging a program and want to change the arguments, select Program Reset and Run | Run to start the program with the new arguments.

Managing multi-file projects

Since most programs consist of more than one file, having a way to automatically identify those that need to be recompiled and linked would be ideal. Turbo C++'s built-in Project Manager does just that and more.

The Project Manager allows you to specify the files belonging to the project. Whenever you rebuild your project, the Project Manager automatically updates the information kept in the project file. This project file includes

- all the files in the project
- where to find them on the disk
- the header files for each source module
- which compilers and command-line options need to be used when creating each part of the program
- where to put the resulting program
- code size, data size, and number of lines from the last compile

Using the Project Manager is easy. To build a project,

- pick a name for the project file (from Project | Open Project)
- add source files using the Project | Add Item dialog box
- tell Turbo C++ to Compile | Make

Then, with the project-management commands available on the Project menu, you can

- add or delete files from your project

- set options for a file in the project
- view included files for a specific file in the project

All the files in this chapter are in the Examples directory.

Let's look at an example of how the Project Manager works.

Sampling the project manager

Suppose you have a program that consists of a main source file, MYMAIN.CPP, a support file, MYFUNCS.CPP, which contains functions and data referenced from the main file, and myfuncs.h. MYMAIN.CPP looks like this:

```
#include <iostream.h>
#include "myfuncs.h"

main(int argc, char *argv[])
{
    char *s;

    if(argc > 1)
        s=argv[1];
    else
        s="the universe";
    cout << GetString() << s << "\n";
}
```

MYFUNCS.CPP looks like this:

```
char ss[] = "The restaurant at the end of ";
char *GetString(void)
{
    return ss;
}
```

And myfuncs.h looks like this:

```
extern char *GetString(void);
```

These files make up the program that we'll now describe to the Project Manager.

These names can be the same (except for the extensions), but they don't have to be. The name of your executable file (and any map file produced by the linker) is based on the project file name.

The first step is to tell Turbo C++ the name of the project file that you're going to use: Call it MYPROG.PRJ. Notice that the name of the project file is not the same as the name of the main file (MYMAIN.CPP). And in this case, the executable file is MYPROG.EXE (and if you choose to generate it, the map file is MYPROG.MAP).

Go to the Project menu and choose Open Project. This brings up the Open Project File dialog box, which contains a list of all the files in the current directory with the extension .PRJ. Since you're starting a new file, type in the name MYPROG in the Open Project File input box.

Notice that once a project is opened, the Add Item, Delete Item, Local Options, and Include Files options are enabled on the Project menu.

If the project file you load is in another directory, the current directory is set to where the project file is loaded.

You can keep your project file in any directory; to put it somewhere other than the current directory, just specify the path as part of the file name. (You must also specify the path for source files if they're in different directories.) Note that all files and corresponding paths are relative to the directory where the project file is loaded from. After you enter the project file name, you'll see a Project window.

The Project window contains the current project file name (MYPROG). Once you indicate which files make up your project, you'll see the name of each file and its path. When the project file is compiled, the Project window also shows the number of lines in the file and the amount of code and data in bytes generated by the compiler.

The status line at the bottom of the screen shows which actions can be performed at this point: *F1* gives you help, *Ins* adds files to the Project, *Del* deletes a file from the Project, *Ctrl+O* lets you select options for a file, *Spacebar* lets you view information about the include files required by a file in the Project, *Enter* opens an editor window for the currently selected file, and *F10* takes you to the main menu. You can also click on any of these items with the mouse to take the appropriate action. Press *Ins* now to add a file to the project list.

*You can change the file-name specification to whatever you want with the Name input box; *.CPP is the default.*

The Add to Project List dialog box appears; this dialog box lets you select and add source files to your project. The Files list box shows all files with the .CPP extension in the current directory. (MYMAIN.CPP and MYFUNCS.CPP both appear in this list.) Three action buttons are available: Add, Done, and Help.

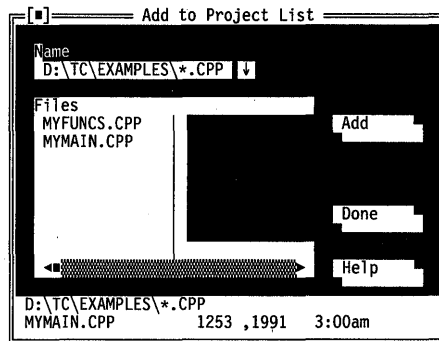
If you copy the wrong file to the Project window, press Esc to return to the Project window, then Del to remove the currently selected file.

Since the Add button is the default, you can place a file in the Project window by typing its name in the Name input box and pressing *Enter* or by choosing it in the Files list box and clicking OK. You can also search for a file in the Files list box by typing the first few letters of the one you want. In this case, typing *my* should take you right to MYFUNCS.CPP. Press *Enter*. You'll see that

MYFUNCS gets added to the Project window and then you're returned to the Add Item dialog box to add another file. Go ahead and add MYMAIN.CPP. Turbo C++ compiles files in the exact order they appear in the project.

Note that the Add button commits your change; pressing Esc when you're in the dialog box just puts the dialog box away.

Close the dialog box and return to the Project window. Notice that the Lines, Code, and Data fields in the Project window show n/a. This means the information is not available until the modules are actually compiled.



After all compiler options and directories have been set, Turbo C++ knows how to build the program called MYPROG.EXE using the source code in MYMAIN.CPP, MYFUNCS.CPP, and myfuncs.h. Now you'll actually build the project.

You can also view your output by choosing Window | Output.

Choose Compile | Make to make your project and choose Run | Run to run it. To view your output, choose Window | User Screen, then press any key to return to the IDE.

When you leave the IDE, the project file you've been working on is automatically saved on disk; you can disable this by unchecking Project in the Preferences dialog box (Options | Environment).

For more information on .PRJ and .DSK files, refer to the section, "Configuration and project files," in Chapter 2.

The saved project consists of two files: the project file (.PRJ) and the desktop file (.DSK). The project file contains the information required to build the project's related executable. The build information consists of compiler options, linker options, make options, INCLUDE/LIB/OUTPUT paths, and transfer items. The desktop file contains the state of all windows at the last time you were using the project.

You can specify a project to load on the DOS command line like this: TC myprog.prj.

The next time you use Turbo C++, you can jump right into your project by reloading the project file. Turbo C++ automatically loads a project file if it is the only .PRJ file in the current directory;

otherwise the default project and desktop (TCDEF.*) are loaded. Since your program files and their corresponding paths are relative to the project file's directory, you can work on any project by moving to the project file's directory and bringing up Turbo C++. The correct file is loaded for you automatically. If no project file is found in the current directory, the default project file is loaded.

Error tracking

Syntax errors that generate compiler warning and error messages in multifile programs can be selected and viewed from the Message window.

To see this, let's introduce some syntax errors into the two files, MYMAIN.CPP and MYFUNCS.CPP. From MYMAIN.CPP, remove the first angle bracket in the first line and remove the *c* in **char** from the fifth line. These changes generate five errors and two warnings in MYMAIN.

In MYFUNCS.CPP, remove the first *r* from `return` in the fifth line. This change produces two errors and one warning.

Changing these files makes them out of date with their object files, so make recompiles them.

Since you want to see the effect of tracking in multiple files, you need to modify the criterion Turbo C++ uses to decide when to stop the make process. This is done by setting a radio button in the Make dialog box (Options | Make).

Stopping a make

You can choose the type of message you want the make to stop on by setting one of the Break Make On options in the Make dialog box (Options | Make). The default is Errors, which is normally the setting you'd want to use. However, you can have a make stop after compiling a file with warnings, with errors, or with fatal errors, or have it stop after all out-of-date source modules have been compiled.

The usefulness of each of these modes is really determined by the way you like to fix errors and warnings. If you like to fix errors and warnings as soon as you see them, you should set Break Make On to Warnings or maybe to Errors. If you prefer to get an entire list of errors in all the source files before fixing them up, you should set the radio button to Fatal Errors or to Link. To

demonstrate errors in multiple files, choose Fatal Errors in the Make dialog box.

Syntax errors in multiple source files

Since you've already introduced syntax errors into MYMAIN.CPP and MYFUNCS.CPP, go ahead and choose Compile | Make to "make the project." The Compiling window shows the files being compiled and the number of errors and warnings in each file and the total for the make. Press any key when the Errors: Press any key message flashes.

Your cursor is now positioned on the first error or warning in the Message window. If the file that the message refers to is in the editor, the highlight bar in the edit window shows you where the compiler detected a problem. You can scroll up and down in the Message window to view the different messages.

Note that there is a "Compiling" message for each source file that was compiled. These messages serve as file boundaries, separating the various messages generated by each module and its include files. When you scroll to a message generated in a different source file, the edit window only tracks in files that are currently loaded.

Thus, moving to a message that refers to an unloaded file causes the edit window's highlight bar to turn off. Press *Spacebar* to load that file and continue tracking; the highlight bar reappears. If you choose one of these messages (that is, press *Enter* when positioned on it), Turbo C++ loads the file it references into an edit window and places the cursor on the error. If you then return to the Message window, tracking resumes in that file.

The Source Tracking options in the Preferences dialog box (Options | Environment) help you determine which window a file is loaded into. You can use these settings when you're message tracking and debug stepping.

Note that Previous message and Next message are affected by the Source Tracking setting. These commands always find the next or previous error and load the file using the method specified by the Source Tracking setting.

Saving or deleting messages

Normally, whenever you start to make a project, the Message window is cleared out to make room for new messages. Sometimes, however, it is desirable to keep messages around between makes.

Consider the following example: You have a project that has many source files and your program is set to stop on Errors. In this case, after compiling many files with warnings, one error in one file stops the make. You fix that error and want to find out if the compiler accepts the fix. But if you do a make or compile again, you lose your earlier warning messages. To avoid this, check Save Old Messages in the Preferences dialog box (Options | Environment). This way the only messages removed are the ones that result from the files you *recompile*. Thus, the old messages for a given file are replaced with any new messages that the compiler generates.

You can always get rid of all your messages by choosing Compile | Remove Messages, which deletes all the current messages. Unchecking Save Old Messages and running another make also gets rid of any old messages.

Autodependency checking

When you made your previous project, you dealt with the most basic situation: a list of C++ source file names. The Project Manager provides you with a lot of power to go beyond this simple situation.

The Project Manager collects autodependency information at compile time and caches these so that only files compiled outside the IDE need to be processed. The Project Manager can automatically check dependencies between source files in the project list (including files they themselves include) and their corresponding object files. This is useful when a particular C++ source file depends on other files. It is common for a C++ source to include several header files (.h files) that define the interface to external routines. If the interface to those routines changes, you'll want the file that uses those routines to be recompiled.

If you've checked the Auto-Dependencies option (Options | Make), Make obtains time-date stamps for all .CPP files and the files included by these. Then Make compares the date/time information

of all these files with their date/time at last compile. If any date/time is different, the source file is recompiled.

Important: Changing compiler options won't cause the Make facility to recompile modules. When you change a menu option, such as Memory Model, select Compile / Build All to ensure all modules are recompiled with the new options.

If the Auto-Dependencies option is unchecked, the .CPP files are checked against .OBJ files. If earlier .CPP files exist, the source file is recompiled.

When a file is compiled, the IDE's compiler and the command-line compiler put dependency information into the .OBJ files. The Project Manager uses this to verify that every file that was used to build the .OBJ file is checked for time and date against the time and date information in the .OBJ file. The .CPP source file is recompiled if the dates are different.

That's all there is to dependencies. You get the power of more traditional makes while avoiding long dependency lists.

Using different file translators

So far you've built projects that use Turbo C++ as the only language translator. Many projects consist of both C++ code and assembler code, and possibly code written in other languages. It would be nice to have some way to tell Turbo C++ how to build such modules using the same dependency checks that we've just described. With the Project Manager, you don't need to worry about forgetting to rebuild those files when you change some of the source code, or about whether you've put them in the right directory, and so on.

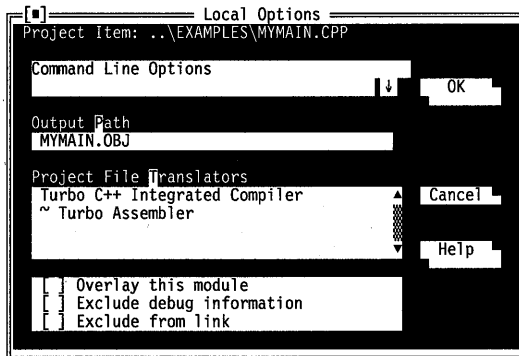
For every source file that you have included in the list in the Project window, you can specify

- which program to use as its target file, for example Turbo C++
- which command-line options to give that program
- whether the module is an overlay
- what to call the resulting module and where it is placed (this information is used by the project manager to locate files needed for linking)
- whether the module contains debug information
- whether the module gets included in the link

By default, the IDE's compiler is chosen as the translator for each module, using no command-line override options, using the Output directory for output, assuming that the module is not an

overlay, and assuming that debug information is not to be excluded. If you use the IDE's compiler, you can use any command-line option except **c**, **Efilename**, **efilename**, **lpath**, **Lpath**, **lx**, **M**, **Q**, and **y**.

Let's look at a simple example. Go to the Project window and move to the file MYFUNCS.CPP. Now press **Ctrl+O** to bring up the Local Options dialog box for this file:



Except for Turbo C++, each of the names in the Project File Translators list box is a reference to a program defined in the Transfer dialog box (Options | Transfer).

Press **Esc**, then **F10** to return to the main menu, then choose Options | Transfer. The Transfer dialog box that appears contains a list of all the transfer programs currently defined. Use the arrow keys to select ~GREP and press **Enter**. (Since the Edit button is the default, pressing **Enter** brings up the Modify/New Transfer Item dialog box.) Here you see that ~GREP is defined as the program `grep`, found on the current path. Notice that the Translator check box is not marked with an **X**; if it were this translator item would be displayed in the Local Options dialog box. Press **Esc** to return to the Transfer dialog box.

Suppose you want to modify a module using a standalone tool instead of the IDE's compiler. To do so, you would perform the following steps:

1. First, you need to define the tool as one of the Project File Translators in the Transfer dialog box. Cursor past the last entry in the Program Titles list, then press **Enter** to bring up the Modify/New Transfer Item dialog box. In the Program Title input box, type your ~Tool Name; in the Program Path input

box, type `toolname`; and in the command line, type `$(EDNAME)` (transfer macro, name of file in active editor).

2. Select the Translator check box. Then press *Enter* (New is the default action button). Back at the Transfer dialog box, you see that *~Tool Name* is now in the Program Titles list box (the last part may not show). Accept the dialog box.
3. Back in the Project window, press *Ctrl+O* to go to the Local Options dialog box again. Notice that *~Tool Name* is now a choice on the Project File Translators list for MYFUNCS.CPP (as well as for all of your other files).

Select your *~Tool Name* from the Project File Translators list box. Use the Command-line Options input box to add any command-line options you want to give the tool.

Your module now uses your tool, while all of your other source modules compile with TC.EXE. The Project Manager applies the same criteria to your module when deciding whether to recompile the module during a make as it does to all the modules that are compiled with TC.EXE.

Overriding libraries

In some cases, it's necessary to override the standard startup files or libraries. You override the startup file by placing a file called `C0x.OBJ` as the *first* name in your project file, where *x* stands for any DOS name (for example, `C0MINE.OBJ`). It's critical that the name start with `C0` and that it is the first file in your project.

To override the standard library, choose Options | Linker and, in the Libraries dialog box, select None for the Standard Run-time Library. Then add the library you want your project to use to the project file just as you would any other item.

More Project Manager features

Let's take a look at some of the other features the Project Manager has to offer. When you're working on a project that involves many source files, you want to be able to easily view portions of those files, and be able to record notes about what you're doing as

you're working. You'll also want to be able to quickly access files that are included by others.

For example, expand MYMAIN.CPP to include a call to a function named **GetMyTime**:

```
#include <iostream.h>
#include "myfuncs.h"
#include "mytime.h"

main(int argc, char *argv[])
{
    char *s;
    if(argc > 1)
        s=argv[1];
    else
        s="the universe";
    cout << GetString() << s << "\n";
}
```

This code adds one new include file to MYMAIN: mytime.h. Together myfuncs.h and mytime.h contain the prototypes that define the **GetString** and **GetMyTime** functions, which are called from MYMAIN. The mytime.h file contains

```
#define HOUR 1
#define MINUTE 2
#define SECOND 3
extern int GetMyTime(int);
```

Go ahead and put the actual code for **GetMyTime** into a new source file called MYTIME.CPP:

```
#include <time.h>
#include "mytime.h"

int GetMyTime(int which)
{
    struct tm *timeptr;
    time_t secsnow;

    time(&secsnow);
    timeptr = localtime(&secsnow);
    switch (which) {
        case HOUR:
            return (timeptr -> tm_hour);
        case MINUTE:
            return (timeptr -> tm_min);
    }
```



```

    case SECOND:
        return (timeptr -> tm_sec);
    }
}

```

MYTIME includes the standard header file `time.h`, which contains the prototype of the **time** and **localtime** functions, and the definition of `tm` and `time_t`, among other things. It also includes `mytime.h` in order to define HOUR, MINUTE, and SECOND.

Create these new files, then use Project | Open Project to open MYPROG.PRJ. The files MYMAIN.CPP and MYFUNCS.CPP are still in the Project window. Now to build your expanded project, add the file name MYTIME.CPP to the Project window. Press *Ins* (or choose Project | Add Item) to bring up the Add Item dialog box. Use the dialog box to specify the name of the file you are adding and choose Done.

Now choose Compile | Make to make the project. MYMAIN.CPP will be recompiled because you've made changes to it since you last compiled it. MYFUNCS.CPP won't be recompiled, because you haven't made any changes to it since the make in the earlier example. MYTIME.CPP will be compiled for the first time.

In the MYPROG project window, move to MYMAIN.CPP and press *Spacebar* (or Project | Include Files) to display the Include Files dialog box. This dialog box contains the name of the selected file, several buttons, and a list of include files and locations (paths). The first file in the Include Files list box is highlighted; the list box lists all the files that were included by the file MYMAIN.CPP. If any of the include files is located outside of the current directory, the path to the file is shown in the Location field of the list box.

As each source file is compiled, the information about which include files are included by which source files is stored in the source file's .OBJ file. If you access the Include Files dialog box before you perform a make, it might contain no files or it might have files left over from a previous compile (which may be out of date). To load one of the include files into an edit window, highlight the file you want and press *Enter* or click the View button.

Notes for your project

Now that you've had a chance to see the code in `MYMAIN.CPP` and `mytime.h`, you might decide to make some changes at a later time. Choose `Window | Project Notes` to bring up a new edit window that is kept as part of your project file. Type in any comments you want to remember about your project.

Each project maintains its own notes file, so that you can keep notes that go with the project you're currently working on; they're available at the touch of a button when you select the project file.

The command-line compiler

The command-line compiler lets you invoke all the functions of the IDE compiler from the DOS command line.

As an alternative to using the IDE, you can compile and run your programs with the command-line compiler (TCC.EXE). Almost anything you can do within the IDE can also be done using the command-line compiler. You can set warnings on or off, invoke the linker to generate executable files, and so on. In fact, if you *only* want to compile your C or C++ source file(s), you must use the `-c` option at the command line.

This chapter is organized into two parts. The first describes how to use the command-line compiler and provides a summary table of all the options. The second part, starting on page 272, presents the options organized functionally with groups of related options.

The summary table, Table 8.1 (starting on page 267), summarizes the command-line compiler options and provides a page-number cross-reference to where you can find more detailed information about each option.

Using the command-line compiler

The command-line compiler uses DPMI (DOS Protected Mode Interface) to run in protected mode on 286, 386, or i486 machines with at least 640K conventional RAM and at least 1MB extended memory.

Note that, although Turbo C++ runs in protected mode, it still generates applications that run in real mode. The advantage to

using Turbo C++ in protected mode is that the compiler has *much* more room to run than if you were running it in real mode, so it can compile larger projects faster and without extensive disk-swapping.

DPMIINST

For more information about running DPMIINST, see Chapter 1, Installing Turbo C++.

The protected mode interface is completely transparent to the user. Turbo C++ uses an internal database of various machine characteristics to determine how to enable protected mode on your machine, and configures itself accordingly. The only time you might need to be aware of it is when running the compiler for the first time. If your machine is not recognized by Turbo C++, run the DPMIINST program by typing the following command at the DOS prompt and following the program's instructions:

```
DPMIINST
```

DPMIINST runs your machine through a series of tests to determine the best way of enabling protected mode.

Running TCC

To invoke Turbo C++ from the command line, type `TCC` at the DOS prompt and follow it with a set of command-line arguments. Command-line arguments include compiler and linker options and file names. The generic command-line format is

```
TCC [option [option...]] filename [filename...]
```

You can also use a configuration file. See page 271 for details.

Each command-line option may be preceded by either a hyphen (-) or slash (/), whichever you prefer. Each option must be separated from the TCC command, other options, and following file names by at least one space.

Using the options

Compiler options are further divided into ten groups.

The options are divided into three general types:

- compiler options, described starting on page 272
- linker options, described starting on page 290
- environment options, described starting on page 291

To see an onscreen list of the options, type `TCC` (without any options or file names) at the DOS prompt. Then press *Enter*.

Use this feature to override settings in configuration files.

In order to select command-line options, enter a hyphen (-) or slash (/) immediately followed by the option letter (for example, -I or /I). To turn an option off, add a second hyphen after the option letter. This is true for all toggle options (those that turn an option on or off): A trailing hyphen (-) turns the option off, and a trailing plus sign (+) or nothing turns it on. So, for example, -C and -C+ both turn nested comments on, while -C- turns nested comments off.

- Option precedence rules
- The option precedence rules are simple; command-line options are evaluated from left to right, and the following rules apply:
- For any option that is *not* an -I or -L option, a duplication on the right overrides the same option on the left. (Thus an *off* option on the right cancels an *on* option to the left.)
 - The -I and -L options on the left, however, take precedence over those on the right.

Table 8.1: Command-line options summary

Option	Page	Function
@filename	271	Read compiler options from the response file <i>filename</i>
+filename	271	Use the alternate configuration file <i>filename</i>
-1	275	Generate 80186 instructions
-1-	275	Generate 8088/8086 instructions (default)
-2	275	Generate 80286 protected-mode compatible instructions
-A	281	Use only ANSI keywords
-A-, -AT	281	Use Turbo C++ keywords (default)
-AK	281	Use only Kernighan and Ritchie keywords
-AU	281	Use only UNIX keywords
-a	275	Align word
-a-	275	Align byte (default)
-B	285	Compile and call the assembler to process inline assembly code
-b	275	Make enums always word-sized (default)
-b-	275	Make enums byte-sized when possible
-C	281	Nested comments on
-C-	281	Nested comments off (default)
-c	285	Compile to .OBJ but do not link
-Dname	274	Define <i>name</i> to the null string
-Dname=string	274	Define <i>name</i> to <i>string</i>
-d	275	Merge duplicate strings on
-d-	275	Merge duplicate strings off (default)
-Efilename	286	Use <i>filename</i> as the assembler to use
-efilename	290	Link to produce <i>filename</i> .EXE
-Fc	275	Generate COMDEFs
-Ff	275	Create far variables automatically
-Ff=size	275	Create far variables automatically; sets the threshold to <i>size</i>
-ff	275	Optimize floating point operations

Table 8.1: Command-line options summary (continued)

-ff-	275	Turn off -ff
-Fm	275	Enables the -Fc , -Ff , and -Fs options
-Fs	275	Assume DS = SS in all memory models
-f	276	Emulate floating point (default)
-f-	276	Don't do floating point
-ff	276	Fast floating point (default)
-ff-	276	Strict ANSI floating point
-f87	277	Use 8087 hardware instructions
-f287	277	Use 80287 hardware instructions
-G	279	Select code for speed
-G-	279	Select code for size (default)
-gn	282	Warnings: stop after <i>n</i> messages
-H	286	Causes the compiler to generate and use precompiled headers
-H-	286	Turns off generation and use of precompiled headers (default)
-Hu	286	Tells the compiler to use but not generate precompiled headers
-H=filename	286	Sets the name of the file for precompiled headers
-h	277	Use fast huge pointer arithmetic
-Ipath	291	Directories for include files
-in	281	Make significant identifier length to be <i>n</i>
-Jg	290	Generate definitions for all template instances and merge duplicates (default)
-Jgd	290	Generate public definitions for all template instances; duplicates results in redefinition errors
-Jgx	290	Generate external references for all template instances
-jn	282	Errors: stop after <i>n</i> messages
-K	277	Default character type unsigned
-K-	277	Default character type signed (default)
-k	277	Standard stack frame on (default)
-Lpath	291	Directories for libraries
-lx	291	Pass option <i>x</i> to the linker (can use more than one <i>x</i>)
-l-x	291	Suppress option <i>x</i> for the linker
-M	291	Instruct the linker to create a map file
-mc	273	Compile using compact memory model
-mh	273	Compile using huge memory model
-ml	273	Compile using large memory model
-mm	273	Compile using medium memory model
-mm!	273	Compile using medium model; assume DS != SS
-ms	273	Compile using small memory model (default)
-ms!	273	Compile using small model; assume DS != SS
-mt	273	Compile using tiny memory model
-mt!	273	Compile using tiny model; assume DS != SS
-N	278	Check for stack overflow
-npath	291	Set the output directory
-O-	279	Optimize jumps
-O-	279	No optimization (default)
-ofilename	286	Compile source file to <i>filename.obj</i>
-P	286	Perform a C++ compile regardless of source file extension
-Pext	286	Perform a C++ compile and set the default extension to <i>ext</i>
-P-	286	Perform a C++ or C compile depending on source file extension (default)
-P-ext	286	Perform a C++ or C compile depending on extension; set default extension to <i>ext</i>

Table 8.1: Command-line options summary (continued)

-p	278	Use Pascal calling convention
-p-	278	Use C calling convention (default)
-Qe	287	Instructs the compiler to use all available EMS memory (default)
-Qe=yyyy	287	Instructs the compiler to use yyyy 16K pages of EMS memory
-Qe-	287	Instructs the compiler to not use any EMS memory
-Qx=nnnn	287	Instructs the compiler to use nnnn Kbytes of extended memory
-r	280	Use register variables on (default)
-r-	280	Suppresses the use of register variables.
-rd	280	Only allow declared register variables to be kept in registers
-S	286	Produce .ASM output file
-Tstring	287	Pass string as an option to TASM or assembler specified with -E
-T-	287	Remove all previous assembler options
-tDe	291	Make the target a DOS .EXE file
-tDc	291	Make the target a DOS .COM file
-Uname	274	Undefine any previous definitions of name
-u	278	Generate underscores (default)
-u-	278	Disables underscores
-v, -v-	278	Source debugging on
-vi, -vi-	279	Controls expansion of inline functions
-V	287	Smart C++ virtual tables
-Va	292	Pass class arguments by reference to a temporary variable
-Vb	292	Make virtual base class pointer same size as 'this' pointer of the class
-Vc	292	Do not add the hidden members and code to classes with pointers to virtual base class members
-Vf	288	Far C++ virtual tables
-Vmv	289	Member pointers have no restrictions (most general representation)
-Vmm	289	Member pointers support multiple inheritance
-Vms	289	Member pointers support single inheritance
-Vmd	289	Use the smallest representation for member pointers
-Vmp	289	Honor the declared precision for all member pointer types
-Vo	293	Enable all of the 'backward compatibility' -V switches (-Va, -Vb, -Vc, -Vp, -Vt, -Vv)
-Vp	292	Pass the 'this' parameter to 'pascal' member functions as the first parameter on the stack
-Vs	288	Local C++ virtual tables
-Vt	293	Place the virtual table pointer after non-static data members
-Vv	293	Do not change the layout of classes to relax restrictions on member pointers
-V0, -V1	288	External and Public C++ virtual tables
-w	282	Display warnings on
-wxxx	282	Enable xxx warning message
-w-xxx	282	Disable xxx warning message
-X	279	Disable compiler autodependency output
-Y	279	Enable overlay code generation
-Yo	279	Overlay the compiled files
-y	279	Line numbers on
-zAname	284	Code class
-zBname	284	BSS class
-zCname	284	Code segment
-zDname	284	BSS segment
-zEname	284	Far segment
-zFname	284	Far class

Table 8.1: Command-line options summary (continued)

<code>-zGname</code>	285	BSS group
<code>-zHname</code>	285	Far group
<code>-zPname</code>	285	Code group
<code>-zRname</code>	285	Data segment
<code>-zSname</code>	285	Data group
<code>-zTname</code>	285	Data class
<code>-zVname</code>	285	Far virtual table segment
<code>-zWname</code>	285	Far virtual table class
<code>-zX*</code>	285	Use default name for X. (default)
<code>-Z</code>	280	Suppress redundant loads. (default)

Syntax and file

names

Turbo C++ compiles files according to the following set of rules:

C++ files have the extension .CPP; see page 286 for information on changing the default extension.

<code>filename.asm</code>	Invoke TASM to assemble to .OBJ
<code>filename.obj</code>	Include as object at link time
<code>filename.lib</code>	Include as library at link time
<code>filename</code>	Compile FILENAME.CPP
<code>filename.cpp</code>	Compile FILENAME.CPP
<code>filename.c</code>	Compile FILENAME.C
<code>filename.xyz</code>	Compile FILENAME.XYZ

For example, given the following command line

```
TCC -a -f -C -emyexe oldfile1 oldfile2 nextfile
```

Turbo C++ compiles `OLDFILE1.CPP`, `OLDFILE2.CPP`, and `NEXTFILE.CPP` to an `.OBJ`, linking them to produce an executable program file named `MYEXE.EXE` with word alignment (`-a`), floating-point emulation (`-f`), and nested comments (`-C`).

Turbo C++ invokes TASM if you give it an `.ASM` file on the command line or if a `.C` or `.CPP` file contains inline assembly. Here are the options that the command-line compiler gives to TASM:

```
/D_ _MODEL_ _ /D_ _LANG_ _ /ml /FLOATOPT
```

where `MODEL` is one of: `TINY`, `SMALL`, `MEDIUM`, `COMPACT`, `LARGE`, or `HUGE`. The `/ml` option tells TASM to assemble with case sensitivity on. `LANG` is `CDECL` or `PASCAL`; `FLOATOPT` is `r` when you've specified `-f87` or `-f287`; `e` otherwise.

Response files

Response files allow you to have longer command strings than DOS normally allows.

If you need to specify many options or files on the command line, you can place them in an ASCII text file, called a response file (you can name it anything you like). You tell the command-line compiler to read its command line from the file by including the file name prefixed with @. You can specify any number of response files, and you can mix them freely with other options and file names.

For example, suppose the file MOON.RSP contains STARS.C and RAIN.C. The following command tells Turbo C++ to compile the files SUN.C, STARS.C, RAIN.C, and ANYONE.C.

```
TCC SUN.C @MOON.RSP ANYONE.C
```

This command expands to the following command:

```
TCC SUN.C STARS.C RAIN.C ANYONE.C
```

See page 267 for what those rules are.

Any options included in a response file are evaluated just as though they had been typed in on the command line.

Configuration files

TURBOC.CFG is not the same as TCCONFIG.TC, which is the default IDE version of a configuration file.

If you find you use a certain set of options over and over again, you can list them in a configuration file, called TURBOC.CFG by default. If you have a TURBOC.CFG configuration file, you don't need to worry about using it. When you run TCC, it automatically looks for TURBOC.CFG in the current directory. If it doesn't find it there, Turbo C++ then looks in the startup directory (where TCC.EXE resides).

You can create more than one configuration file; each must have a unique name. To specify the alternate configuration file name, include its file name, prefixed with +, anywhere on the TCC command line. For example, to read the option settings from the file D:\ALT.CFG, you could use the following command line:

```
TCC +D:\ALT.CFG .....
```

Your configuration file can be used in addition to or instead of options entered on the command line. If you don't want to use certain options that are listed in your configuration file, you can override them with options on the command line.

You can create the TURBOC.CFG file (or any alternate configuration file) using any standard ASCII editor or word processor, such as Turbo C++'s integrated editor. You can list options (separated by spaces) on the same line or list them on separate lines.

Option precedence rules

In general, you should remember that command-line options override configuration file options. If, for example, your configuration file contains several options, including the **-a** option (which you want to turn *off*), you can still use the configuration file but override the **-a** option by listing **-a-** in the command line. However, the rules are a little more detailed than that. The option precedence rules detailed on page 267 apply, with these additional rules:

1. When the options from the configuration file are combined with the command-line options, any **-I** and **-L** options in the configuration file are appended to the right of the command-line options. This means that the include and library directories specified in the command line are the first ones that Turbo C++ searches (thereby giving the command-line **-I** and **-L** directories priority over those in the configuration file).
2. The remaining configuration file options are inserted immediately after the TCC command (to the left of any command-line options). This gives the command-line options priority over the configuration file options.

Compiler options

Turbo C++'s command-line compiler options fall into ten groups; the page references to the left of each group tell where you can find a discussion of each kind of option:

- | | |
|---------------|---|
| See page 273. | 1. Memory model options let you tell Turbo C++ which memory model to use when compiling your program. |
| See page 274. | 2. Macro definitions let you define and undefine macros on the command line. |
| See page 275. | 3. Code-generation options govern characteristics of the generated code, such as the floating-point option, calling convention, character type, or CPU instructions. |
| See page 281. | 4. Source code options cause the compiler to recognize (or ignore) certain features of the source code; implementation- |

- specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths.
- See page 282. 5. **Error-reporting options** let you tailor which warning messages the compiler reports, and the maximum number of warnings and errors that can occur before the compilation stops.
- See page 284. 6. **Segment-naming control options** allow you to rename segments and to reassign their groups and classes.
- See page 285. 7. **Compilation control options** let you direct the compiler to
- compile to assembly code (rather than to an object module)
 - compile a source file that contains inline assembly
 - compile without linking
 - use precompiled headers or not
- See page 287. 8. **EMS options** let you control how much expanded or extended memory Turbo C++ uses.
- See page 287. 9. **C++ virtual table options** let you control how virtual tables are handled.
- See page 289. 10. **C++ member pointer options** let you control how member pointers are used.
- See page 290. 11. **Template generation options** let you control how the compiler generates definitions or external declarations for template instances.
- See page 292. 12. **Backward compatibility options** let you tell the compiler to use particular code generation strategies to insure backward compatibility with earlier versions of Turbo C++.

Memory model

Memory model options let you tell Turbo C++ which memory model to use when compiling your program. The memory models are tiny, small, medium, compact, large, and huge.

See Chapter 18 for in-depth information on the memory models (what they are, how to use them).

- mc** Compile using compact memory model
- mh** Compile using huge memory model
- ml** Compile using large memory model
- mm** Compile using medium memory model
- mm!** Compile using medium model; DS != SS
- ms** Compile using small memory model (the default)
- ms!** Compile using small model; DS != SS
- mt** Compile using tiny memory model
- mt!** Compile using tiny model; DS != SS

NOTE: You can't use the **-N** option when using one of the *DS != SS* models.

The net effect of the **-mt!**, **-ms!**, and **-mm!** options is actually very small. If you take the address of a stack variable (auto or parameter), the default (when *DS == SS*) is to make the resulting pointer a near (*DS* relative) pointer. In this way one can simply assign the address to a default sized pointer in those models without problems. When *DS != SS*, the pointer type created when you take the address of a stack variable is an **_ss** pointer. This means that the pointer can be freely assigned or passed to a far pointer or to a **_ss** pointer. But for the memory models affected, assigning the address to a near or default-sized pointer produces a "Suspicious pointer conversion" warning. Such warnings are usually errors, and the warning defaults to on. You should regard this kind of warning as a likely error.

Macro definitions

Macro definitions let you define and undefine macros (also called *manifest* or *symbolic* constants) on the command line. The default definition is the null string. Macros defined on the command line override those in your source file.

- Dname** Defines the named identifier *name* to the null string.
- Dname=string** Defines the named identifier *name* to the string *string* after the equal sign. *string* cannot contain any spaces or tabs.
- Uname** Undefines any previous definitions of the named identifier *name*.

Turbo C++ lets you make multiple **#define** entries on the command line in any of the following ways:

- You can include multiple entries after a single **-D** option, separating entries with a semicolon (this is known as "ganging" options):
TCC **-Dxxx;yyy=1;zzz=NO MYFILE.C**
- You can place more than one **-D** option on the command line:
TCC **-Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C**
- You can mix ganged and multiple **-D** listings:
TCC **-Dxxx -Dyyy=1;zzz=NO MYFILE.C**

Code-generation options

Code-generation options govern characteristics of the generated code, such as the floating-point option, calling convention, character type, or CPU instructions.

- 1 This option causes Turbo C++ to generate extended 80186 instructions. It also generates 80286 programs running in real mode, such as programs for the IBM PC/AT under DOS.
- 1- Tells the compiler to generate 8088/8086 instructions (the default).
- 2 This option causes Turbo C++ to generate instructions compatible with 80286 protected-mode.
- a This option forces integer size and larger items to be aligned on a machine-word boundary. Extra bytes are inserted in a structure to ensure member alignment. Automatic and global variables are aligned properly. **char** and **unsigned char** variables and fields can be placed at any address; all others are placed at an even-numbered address. This option is off by default (**-a-**), allowing bitwise alignment.
- b This option (which is on by default) tells the compiler to always allocate a whole word for enumeration types.
- b- This option tells the compiler to allocate a signed or unsigned byte if the minimum and maximum values of the enumeration are both within the range of 0 to 255 or -128 to 127, respectively.
- d This option tells the compiler to merge literal strings when one string matches another, thereby producing smaller programs. This option is off by default (**-d-**).
- Fc This generates communal variables (COMDEFs) for global "C" variables that are not initialized and not declared as **static** or **extern**. The advantage of using this option is that header files that are included in several source files can contain declarations of global variables. So long as a given variable doesn't need to be initialized to a nonzero value, you don't need to include a definition for it in any of the source files. You can use

this option when porting code that takes advantage of a similar feature with another implementation.

-Ff When you use this option, global variables greater than or equal to the threshold size are automatically made far by the compiler. The threshold size defaults to 32,767; you can change it with the **-Ff=size** option. This option is useful for code that doesn't use the huge memory model but declares enough large global variables that their total size exceeds (or is close to) 64K. For tiny, small, and medium models this option has no effect.

If you use this option in conjunction with **-Fc**, the generated COMDEFs are **far** in the compact, large, and huge models.

-ff With this option, the compiler optimizes floating-point operations without regard for explicit or explicit type conversions. Computations might be completed faster with this option than they are under ANSI mode. See Chapter 19, Mathematical operations for more information.

-Ff=size Use this option to change the threshold size used by the **-Ff** option.

-Fm This option enables all the other **-F** options (**-Fc**, **-Ff** and **-Fs**). You can use it as a handy shortcut when porting code from other compilers.

-Fs This option tells the compiler to assume that DS is equal to SS in all memory models; you can use it when porting code originally written for an implementation that makes the stack part of the data segment. When you specify this option, the compiler links in an alternate startup module (C0Fx.OBJ) that places the stack in the data segment.

-f This option tells the compiler to emulate 80x87 calls at run time if the run-time system does not have an 80x87; if it does have one, the compiler calls the 80x87 chip for floating-point calculations (the default).

-f- This option specifies that the program contains no floating-point calculations, so no floating-point libraries are linked at the link step.

- ff** With this option, the compiler optimizes floating-point operations without regard to explicit or implicit type conversions. Answers can be faster than under ANSI operating mode. See Chapter 19, "Mathematical operations" for details.
- ff-** This option turns off the fast floating-point option. The compiler follows strict ANSI rules regarding floating-point conversions.
- f87** This option tells the compiler to generate floating-point operations using inline 80x87 instructions rather than using calls to 80x87 emulation library routines. It specifies that a math coprocessor is available at run time; therefore, programs compiled with this option don't run on a machine that does not have a math coprocessor.
- f287** This option is similar to **-f87**, but uses instructions that are only available with an 80287 (or higher) chip.
- h** This option offers an alternative way of calculating huge pointer expressions; a way which is much faster but must be used with caution. When you use this option, huge pointers are normalized only when a segment wraparound occurs in the offset part. This causes problems for huge arrays if any array elements cross a segment boundary. This option is off by default.

Normally, Turbo C++ normalizes a huge pointer whenever adding to or subtracting from it. This ensures that, for example, if you have a huge array of **structs** that's larger than 64K, indexing into the array and selecting a **struct** field always works with **structs** of any size. Turbo C++ accomplishes this by always normalizing the results of huge pointer operations, so that the offset part contains a number that's no higher than 15. That way, a segment wraparound never occurs with huge pointers. The disadvantage of this approach is that it tends to be quite expensive in terms of execution speed.
- K** This option tells the compiler to treat all **char** declarations as if they were **unsigned char** type. This allows for compatibility with other compilers that treat **char** declarations as **unsigned**. By default, **char** declarations are **signed** (**-K-**).

- k** This option generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines. This option is on by default.
- N** This option generates stack overflow logic at the entry of each function, which causes a stack overflow message to appear when a stack overflow is detected. This is costly in terms of both program size and speed but is provided as an option because stack overflows can be very difficult to detect. If an overflow is detected, the message "Stack overflow!" is printed and the program exits with an exit code of 1.
- p** This option forces the compiler to generate all subroutine calls and all functions using the Pascal parameter-passing sequence. The resulting function calls are smaller and faster. Functions must pass the correct number and type of arguments, unlike normal C use, which permits a variable number of function arguments. You can use the **cdecl** statement to override this option and specifically declare functions to be C-type. This option is off by default (**-p-**).
- u** With **-u** selected, when you declare an identifier, Turbo C++ automatically puts an underscore (**_**) in front of the identifier before saving the identifier in the object module.

Turbo C++ treats Pascal-type identifiers (those modified by the **pascal** keyword) differently—they are uppercase and are *not* prefixed with an underscore.

Underscores for C and C++ identifiers are optional, but on by default. You can turn them off with **-u-**. However, if you are using the standard Turbo C++ libraries, you'll have problems unless you rebuild the libraries. (To do this, you need the Turbo C++ run-time library source code; contact Borland for more information.)

*Unless you are an expert, don't use **-u-**. See **BASM.DOC** for details about underscores.*

The **-v** and **-vi** options

- v** This option tells the compiler to include debugging information in the .OBJ file so that the file(s) being compiled can be debugged with Turbo C++'s integrated debugger. The compiler also passes this option on to the linker so it can include the debugging information in the .EXE file.

To facilitate debugging, this option also causes C++ inline functions to be treated as normal functions. If you want to avoid that, use **-vi**.

- vi** With this option enabled, C++ inline functions are expanded inline.
- v-** This option turns debugging off and inline expansion on.
- vi-** This option turns inline expansion off.

So, for example, if you want to turn both debugging and inline expansion on, you must use **-v -vi**.

- X** This option disables generation of autodependency information in the output file. Modules compiled with this option enabled can't use the autodependency feature of MAKE or of the IDE. Normally this option is only used for files that are to be put into .LIB files (to save disk space).
- Y** This option generates overlay-compatible code. Every file in an overlaid program must be compiled with this option; see Chapter 18, "Memory management" for details on overlays.
- Yo** This option overlays the compiled file(s); see Chapter 18 for details.
- y** This option includes line numbers in the object file for use by a symbolic debugger, such as Turbo Debugger. This increases the size of the object file but doesn't affect size or speed of the executable program. This option is useful only in concert with a symbolic debugger that can use the information. In general, **-v** is more useful than **-y** with Turbo Debugger.

Optimization options

Optimization options let you specify how the object code is to be optimized; for size or speed, with or without the use of register variables, and with or without assumptions about aliases.

- G** This option causes the compiler to bias its optimization in favor of speed over size.

- G-** This option, the default, causes the compiler to bias its optimization in favor of size over speed (where smaller is better).
- O** This option eliminates redundant jumps (such as jumps to jumps) and multiple copies of identical code that jump to the same location. It also suppresses redundant register loads.
- r** This option enables the use of register variables (the default).
- r-** This option suppresses the use of register variables. When you are using this option, the compiler won't use register variables, and it won't preserve and respect register variables (SI,DI) from any caller. For that reason, you should not have code that uses register variables call code which has been compiled with **-r-**.

*Unless you are an expert,
don't use -r-.*

On the other hand, if you are interfacing with existing assembly-language code that does not preserve SI,DI, the **-r-** option allows you to call that code from Turbo C++.

- rd** This option only allows declared register variables to be kept in registers.
- Z** This option allows the compiler to assume that variables are not accessed both directly and with a pointer in the same function. It is effective only when used with **-O**. The compiler keeps a table that reflects the current contents of registers. If a variable had to be loaded from memory to a register, the compiler remembers that the register now contains a copy of the variable. If the variable is used again, the compiler uses the copy in the register rather than the value in memory. The **-Z** option determines how the compiler handles indirect assignments. Normally it assumes that such assignments might change a variable; consequently, it ignores copies of register variables, erasing the table. The **-Z** option tells the compiler that indirect assignments won't change variable, and it's safe to keep the copies. If you access a variable directly and through a pointer within a function, erroneous code can be generated if this option is turned on, but better performance and correct code might also result.

Source code options

Source code options cause the compiler to recognize (or ignore) certain features of the source code; implementation-specific (non-ANSI, non-Kernighan and Ritchie, and non-UNIX) keywords, nested comments, and identifier lengths. These options are most significant if you plan to port your code to other systems.

-A This option compiles ANSI-compatible code: Any of the Turbo C++ extension keywords are ignored and can be used as normal identifiers. These keywords include

See Chapter 11, "Lexical elements" for a complete list of the Turbo C++ keywords.

asm	_ds	interrupt	_ss
cdecl	_es	_loadds	_saveregs
_cs	far	near	_seg
	huge	pascal	

and the register pseudovariables, such as `_AX`, `_BX`, `_SI`, and so on.

-A- This option tells the compiler to use Turbo C++ keywords. **-AT** is an alternate version of this option.

-AK This option tells the compiler to use only Kernighan and Ritchie keywords.

-AU This option tells the compiler to use only UNIX keywords.

-C This option allows you to nest comments. Comments may not normally be nested.

-in This option causes the compiler to recognize only the first *n* characters of identifiers. All identifiers, whether variables, preprocessor macro names, or structure member names, are treated as distinct only if their first *n* characters are distinct.

By default, Turbo C++ uses 32 characters per identifier. Other systems, including some UNIX compilers, ignore characters beyond the first eight. If you are porting to these other environments, you may wish to compile your code with a smaller number of significant characters. Compiling in this manner helps you see if there are any name conflicts in long identifiers when they are truncated to a shorter significant length.

Error-reporting options

Error-reporting options let you tailor which warning messages the compiler reports, and the maximum number of warnings and errors that can occur before the compilation stops.

- gn** This option tells Turbo C++ to stop compiling after *n* warning messages.
- jn** This option tells the compiler to stop compiling after *n* error messages.
- w** This option causes the compiler to display warning messages. You can turn this off with **-w-**. You can enable or disable specific warning messages with **-wxxx**, described in the following paragraphs.
- wxxx** This option enables the specific warning message indicated by *xxx*. The option **-w-xxx** suppresses the warning message indicated by *xxx*. The possible options for **-wxxx** are listed here and divided into four categories: ANSI violations, frequent errors (including more frequent errors), portability warnings, and C++ warnings. You can also use the pragma **warn** in your source code to control these options. See Chapter 14, "The preprocessor."

For more information on these warnings, see Appendix C, "Error messages"

ANSI violations

The asterisk (*) indicates that the option is on by default. All others are off by default.

- wbbf** Bit fields must be **signed** or **unsigned int**.
- wbig*** Hexadecimal value contains more than three digits.
- wdpu*** Declare *type* prior to use in prototype.
- wdup*** Redefinition of *macro* is not identical.
- weas** Assigning *type* to *enumeration*.
- wext*** *Identifier* is declared as both external and static.
- wpin** Initialization is only partially bracketed.
- wret*** Both return and return with a value used.
- wstu*** Undefined structure *structure*.
- wsus*** Suspicious pointer conversion.
- wvoi*** Void functions may not return a value.
- wzdi*** Division by zero.

Frequent errors

-wamb	Ambiguous operators need parentheses.
-wamp	Superfluous & with function or array.
-wasm	Unknown assembler instruction.
-waus*	<i>Identifier</i> is assigned a value that is never used.
-wccc*	Condition is always true/false.
-wdef	Possible use of <i>identifier</i> before definition.
-weff*	Code has no effect.
-wias*	Array variable identifier is near.
-will*	Ill-formed pragma.
-wnod	No declaration for function <i>function</i> .
-wpar*	Parameter <i>parameter</i> is never used.
-wpia*	Possibly incorrect assignment.
-wpro	Call to function with no prototype.
-wrch*	Unreachable code.
-wrvl*	Function should return a value.
-wstv	Structure passed by value.
-wuse	<i>Identifier</i> is declared but never used.

Portability warnings

-wcln	Constant is long.
-wcpt*	Nonportable pointer comparison.
-wrng*	Constant out of range in comparison.
-wrpt*	Nonportable pointer conversion.
-wsig	Conversion may lose significant digits.
-wucp	Mixing pointers to signed and unsigned char .

C++ warnings

-wbei*	Initializing enumeration with <i>type</i> .
-wdsz*	Array size for 'delete' ignored.
-whid*	<i>Function1</i> hides virtual function <i>function2</i> .
-wibc*	Base class <i>base1</i> is inaccessible because also in <i>base2</i> .
-winl*	Functions containing <i>identifier</i> are not expanded inline.
-wlin*	Temporary used to initialize <i>identifier</i> .
-wlvc*	Temporary used for parameter in call to <i>identifier</i> .
-wmpc*	Conversion to <i>type</i> fail for members of virtual base class <i>base</i> .
-wmpd*	Maximum precision used for member pointer type <i>type</i> .
-wncf*	Non-const function <i>function</i> called const object.
-wnci*	Constant member <i>identifier</i> is not initialized.

- wnst*** Use qualified name to access nested type *type*.
- wnvf*** Non-volatile function *function* called for volatile object.
- wobi*** Base initialization without a class name is now obsolete.
- wofp*** Style of function definition is now obsolete.
- wovl*** Overload is now unnecessary and obsolete.
- wpre** Overloaded prefix operator ++/-- used as a postfix operator.

Segment-naming control

Segment-naming control options allow you to rename segments and to reassign their groups and classes.

Don't use these options unless you have a good understanding of segmentation on the 8086 processor. Under normal circumstances, you don't need to specify segment names.

- zAname** This option changes the name of the code segment class to *name*. By default, the code segment is assigned to class CODE.
- zBname** This option changes the name of the uninitialized data segment class to *name*. By default, the uninitialized data segments are assigned to class BSS.
- zCname** This option changes the name of the code segment to *name*. By default, the code segment is named `_TEXT`, except for the medium, large and huge models, where the name is *filename_TEXT*. (*filename* here is the source file name.)
- zDname** This option changes the name of the uninitialized data segment to *name*. By default, the uninitialized data segment is named `_BSS`, except in the huge model, where no uninitialized data segment is generated.
- zEname** This option changes the name of the segment where far objects are put to *name*. By default, the segment name is the name of the far object followed by `_FAR`. A name beginning with an asterisk (*) indicates that the default string should be used.
- zFname** This option changes the name of the class for far objects to *name*. By default, the name is `FAR_DATA`. A name beginning with an asterisk (*) indicates that the default string should be used.

See Chapter 18, "Memory management" for more on far objects.

- zGname** This option changes the name of the uninitialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
- zHname** This option causes far objects to be put into group *name*. By default, far objects are not put into a group. A name beginning with an asterisk (*) indicates that the default string should be used.
- zPname** This option causes any output files to be generated with a code group for the code segment named *name*.
- zRname** This option sets the name of the initialized data segment to *name*. By default, the initialized data segment is named _DATA, except in the huge model, where the segment is named *filename*_DATA.
- zSname** This option changes the name of the initialized data segment group to *name*. By default, the data group is named DGROUP, except in the huge model, where there is no data group.
- zTname** This option sets the name of the initialized data segment class to *name*. By default the initialized data segment class is named DATA.
- zVname** This option sets the name of the far virtual table segment to *name*. By default far virtual tables are generated in the code segment.
- zWname** This option sets the name of the far virtual table class segment to *name*. By default far virtual table classes are generated in the CODE segment.
- zX*** This option uses the default name for X. For example, **-zA*** assigns the default class name CODE to the code segment.

Compilation control options

Compilation control options allow you to control compilation of source files, such as whether your code is compiled as C or C++ or whether to use precompiled headers.

- B** This option compiles and calls the assembler to process inline assembly code.

- c** This option compiles and assembles the named .C and .CPP, files, but does not execute a link command.
- Efilename** This option uses *name* as the name of the assembler to use. By default, TASM is used.
- H** This option causes the compiler to generate and use precompiled headers, using the default filename TCDEF.SYM.
- H-** This option turns off generation and use of precompiled headers (this is the default).
- Hu** This option tells the compiler to use but not generate precompiled headers.
- H=filename** This option sets the name of the file for precompiled headers, if you wish to save this information in a file other than TCDEF.SYM. This option also turns on generation and use of precompiled headers; that is, it also has the effect of **-H**.
- ofilename** This option compiles the named file to the specified *filename.obj*.
- P** This option causes the compiler to compile your code as C++ always, regardless of extension. The compiler assumes that all files have .CPP extensions unless a different extension is specified with the **-Pext** option as described later.
- Pext** This option causes the compiler to compile all files as C++; it changes the default extension to whatever you specify with *ext*. This option is available because some programmers use .C or another extension as their default extension for C++ code.
- P-** This option tells the compiler to compile a file as either C or C++, based on its extension. The default extension is .CPP. This option is the default.
- P-ext** This option also tells the compiler to compile code based on the extension (.CPP as C++ code, all other file-name extensions as C code). It further specifies what the default extension is to be.

See Appendix B for more on precompiled headers.

Note that this option behaves differently from the **-P** option in Turbo C++ 1.x.

- S** This option compiles the named source files and produces assembly language output files (.ASM), but does not assemble. When you use this option, Turbo C++ includes the C or C++ source lines as comments in the produced .ASM file.
- Tstring** This option passes *string* as an option to TASM (or as an option to the assembler defined with **-E**).
- T-** This option removes all previously defined assembler options.

EMS and expanded memory options

If you have expanded (EMS) memory, you may want to make this memory available to the compiler for “swap” space in the event that your computer’s extended (protected mode) memory is exhausted during compilation. These options give you the ability to control the compiler’s use of EMS memory. You can also control the amount of expanded (protected mode) memory Turbo C++ uses.

- Qe** This option instructs the compiler to use all EMS memory it can find. This is on by default for the command-line compiler (TCC). It speeds up your compilations, especially for large source files.
- Qe=yyyy** This option instructs the compiler to use *yyyy* pages (in 16K page sizes) of EMS memory for itself.
- Qe-** This option instructs the compiler not to use any EMS memory.
- Qx=nnnn** This option instructs the compiler to use *nnnn* bytes of extended memory.

C++ virtual tables

The **-V** option controls the C++ virtual tables. There are five variations of the **-V** option:

- V** Use this option when you want to generate C++ virtual tables (and inline functions not expanded inline) so that only one instance of a given virtual table or inline function are included in the program. This produces

the smallest executables, but uses .OBJ and .ASM extensions only available with TLINK 3.0 and TASM 2.0 (or later).

-Vs Use this option when you want Turbo C++ to generate local virtual tables (and inline functions not expanded inline) such that each module gets its own private copy of each virtual table (or inline function) it uses. This option uses only standard .OBJ (and .ASM) constructs, but produces larger executables.

-V0, -V1 These options work together to create global virtual tables. If you don't want to use the Smart or Local options (**-V** or **-Vs**), you can use **-V0** and **-V1** to produce and reference global virtual tables. **-V0** generates external references to virtual tables; **-V1** produces public definitions for virtual tables.

When using these two options, at least one of the modules in the program must be compiled with the **-V1** option to supply the definitions for the virtual tables. All other modules should be compiled with the **-V0** option to refer to that Public copy of the virtual tables.

-Vf You can use this option independently of or in conjunction with any of the other virtual table options. It causes virtual tables to be created in the code segment instead of the data segment (unless changed using the **-zV** and **-zW** options), and makes virtual table pointers into full 32-bit pointers (the latter is done automatically if you are using the huge memory model).

There are two primary reasons for using this option: to remove the virtual tables from the data segment, which may be getting full, and to be able to share objects (of classes with virtual functions) between modules that use different data segments. You must compile all modules that may share objects either entirely with or entirely without this option. You can achieve the same effect by using the **huge** modifier on a class-by-class basis.

C++ member pointers

The **-Vm** options control C++ member pointer types. There are five variations of the **-Vm** option:

The Turbo C++ compiler supports three different kinds of member pointer types, with varying degrees of complexity and generality. By default, the compiler uses the most general (but in some contexts also the least efficient) kind for all member pointer types; this default behavior can be changed via the **-Vm** family of switches.

- Vmv** Member pointers declared while this option is in effect have no restriction on what members they point to; they use the most general representation.
- Vmm** Member pointers declared while this option is in effect are allowed to point to members of multiple inheritance classes, except that members of virtual base classes cannot be pointed to.
- Vms** Member pointers declared while this option is in effect are not allowed to point to members of some base classes of classes that use multiple inheritance (in general, they can be used with single inheritance classes only).
- Vmd** Member pointers declared while this option is in effect use the smallest possible representation that allows member pointers to point to all members of their class. If the class is not fully defined at the point where the member pointer type is declared, the most general representation has to be chosen by the compiler (and a warning is issued about this).
- Vmp** Whenever a member pointer is dereferenced or called, the compiler treat the member pointer as if it were of the least general case needed for that particular pointer type. For example, a call through a pointer to member of a class that is declared without any base classes treat the member pointer as having the simplest representation, regardless of how it's been declared. This works correctly (and produces the most efficient code) in all cases except for one: when a pointer to a derived class is explicitly cast to a pointer to member of a 'simpler' base class, when the pointer is actually

pointing to a derived class member. This is a non-portable (and dubious) construct, but if you need to compile code that uses it, use the `-Vmp` option. It forces the compiler to honor the declared precision for all member pointer types.

Template generation options

The `-Jg` option controls the generation of template instances in C++. There are three variations of the `-Jg` option:

-Jg Public definitions of all template instances encountered when this switch value is in effect are generated, and if more than one module generates the same template instance, the linker merges them to produce a single copy of the instance. This option (the default) is the most convenient approach to generating template instances. In order to generate the instances, however, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class).

-Jgd This option tells the compiler to generate public definitions for all template instances encountered. Unlike the `-Jg` option, however, duplicate instances are *not* merged, causing the linker to report public symbol redefinition errors if more than one module defines the same template instance.

-Jgx This option instructs the compiler to generate external references to template instances. If you use this option you must make sure that the instances are publicly defined in some other module (using the `-Jgd` option), so that the external references are satisfied.

For more information about templates, see Chapter 13, "C++ specifics."

Linker options

See the section on TLINK for a list of linker options.

-efilename This option derives the executable program's name from *filename* by adding the file extension .EXE (the program name is *filename.EXE*). *filename* must immediately follow the `-e`, with no intervening whitespace. Without this option, the linker derives

the .EXE file's name from the name of the first source or object file in the file name list.

- tDe** This specifies that the target (output) file is a DOS .EXE file.
- tDc** This specifies that the target (output) file is a DOS .COM file.
- lx** This option (which is a lowercase l) passes option *x* to the linker. The option **-l-x** suppresses option *x*. More than one option can appear after the **-l**.
- M** This option forces the linker to produce a full link map. The default is to produce no link map.

Environment options

When working with environment options, bear in mind that Turbo C++ recognizes two types of library files: *implicit* and *user-specified* (also known as *explicit* library files). These are defined and discussed on page 294.

- lpath** This option (which is an uppercase I) causes the compiler to search *path* (the drive specifier or path name of a subdirectory) for include files (in addition to searching the standard places). A drive specifier is a single letter, either uppercase or lowercase, followed by a colon (:). A directory is any valid directory or directory path. You can use more than one **-l** directory option.
- Lpath** This option forces the linker to get the C0x.OBJ start-up object file and the Turbo C++ library files (Cx.LIB, MATHx.LIB, EMU.LIB, and FP87.LIB) from the named directory. By default, the linker looks for them in the current directory.
- npath** This option places any .OBJ or .ASM files created by the compiler in the directory or drive named by *path*.

Backward compatibility options

Turbo C++ version 3.0 introduces a number of improvements in the way some C++ operations are implemented, resulting in smaller, faster code with fewer restrictions and less overhead. In some cases, the new implementation is not fully compatible with previous versions of Turbo C++. Where such compatibility is needed, the following options are provided:

- Va** When an argument of type class with constructors is passed by value to a function, this option instructs the compiler to create a temporary variable at the calling site, initialize this temporary with the argument value, and pass a reference to this temporary to the function. This behavior is compatible with previous versions of Turbo C++. By default, version 3.0 copy-constructs such argument values directly to the stack, thus avoiding the introduction of the temporary (and also making access to the argument value faster).
- Vb** When a class inherits virtually from a base class, the compiler stores a hidden pointer in the class object to access the virtual base class subobject. The Turbo C++ 3.0 compiler makes this pointer always 'near', which allows it to generate more efficient code. For backward compatibility, the **-Vb** option directs the TC++ 3.0 compiler to match the hidden pointer to the size of the 'this' pointer used by the class itself.
- Vc** To correctly implement the case when a derived class overrides a virtual function that it inherits from a virtual base class, and a constructor or destructor for the derived class calls that virtual function using a pointer to the virtual base class, the compiler may add hidden members to the derived class, and add more code to its constructors and destructors. This option directs the compiler *not* to add the hidden members and code, so that class instance layout is the same as with previous versions of Turbo C++.
- Vp** This option directs the compiler to pass the 'this' parameter to 'pascal' member functions as the first parameter on the stack, for compatibility with previous versions of Turbo C++. By default, version 3.0 always

pushes 'this' as the last parameter regardless of calling convention.

- Vt** This option instructs the compiler to place the virtual table pointer after any non-static data members of the particular class, to ensure compatibility when class instances are to be shared with non-C++ code and when sharing classes with code compiled with previous versions of Turbo C++. By default, version 3.0 adds this pointer *before* any non-static data members of the class, thus making virtual member function calls smaller and faster.
- Vv** This option directs the compiler not to change the layout of any classes (which it may need to do in order to allow pointers to virtual base class members, which were not supported in previous versions of Turbo C++). If this option is used, the compiler isn't able to create a pointer to a member of a base class that can only be reached from the derived class through two or more levels of virtual inheritance.
- Vo** This option is a "master switch" that turns on all of the backward-compatibility options listed in this section. It can be used as a handy shortcut when linking with libraries built with older versions of Turbo C++.

Searching for include and library files

Turbo C++ can search multiple directories for include and library files. This means that the syntax for the library directories (**-L**) and include directories (**-I**) command-line options, like that of the **#define** option (**-D**), allows multiple listings of a given option.

Here is the syntax for these options:

Library directories: `-Ldirname[;dirname;...]`
Include directories: `-Idirname[;dirname;...]`

The parameter *dirname* used with **-L** and **-I** can be any directory or directory path.

You can enter these multiple directories on the command line in the following ways:

- You can "gang" multiple entries with a single **-L** or **-I** option, separating ganged entries with a semicolon, like this:


```
TCC -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
TCC -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix ganged and multiple listings, like this:

```
TCC -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfile.c
```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative: The compiler searches all the directories listed, in order from left to right.

Note The IDE also supports multiple library directories through the “ganged entry” syntax.

File-search algorithms

The Turbo C++ include-file search algorithms search for the **#include** files listed in your source code in the following way:

- If you put an **#include <somefile.h>** statement in your source code, Turbo C++ searches for **somefile.h** only in the specified include directories.
- If, on the other hand, you put an **#include "somefile.h"** statement in your code, Turbo C++ searches for **somefile.h** first in the current directory; if it does not find the header file there, it then searches in the include directories specified in the command line.

The library file search algorithms are similar to those for include files:

- **Implicit libraries:** Turbo C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for **#include <somefile.h>**. [Implicit library files are the ones Turbo C++ automatically links in. These are **Cx.LIB**, **EMU.LIB** or **FP87.LIB**, **MATHx.LIB**, **OVERLAY.LIB**, **C0F.OBJ**, and the start-up object file (**C0x.OBJ**).]
- **Explicit libraries:** Where Turbo C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name. (Explicit library files are the ones you list on the command line or in a project file; these are file names with a **.LIB** extension.)
 - If you list an explicit library file name with no drive or directory (like this: **mylib.lib**), Turbo C++ searches for that library

Your code written under older versions of Turbo C or Turbo C++ works without problems in this version of Turbo C++.

in the current directory first. Then (if the first search was unsuccessful), it looks in the specified library directories. This is similar to the search algorithm for `#include "somefile.h"`.

- If you list a user-specified library with drive and/or directory information (like this: `c:mystuff\mylib1.lib`), Turbo C++ searches *only* in the location you explicitly listed as part of the library path name and not in the specified library directories.

An annotated
example

Here is an example of a Turbo C++ command line that incorporates multiple library and include directory options.

1. Your current drive is C:, and your current directory is C:\TC. This example makes the following assumptions: A drive's current position is A:\ASTROLIB; C:\TC\BIN is in your DOS path.
2. Your include files (.h or "header" files) are located in C:\TC\INCLUDE.
3. Your startup files (C0T.OBJ, C0S.OBJ, ... , C0H.OBJ) are in C:\TC\LIB.
4. Your standard Turbo C++ library files (CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ... , EMU.LIB, FP87.LIB, and so forth) are in C:\TC\LIB.
5. Your custom library files for star systems (which you created and manage with TLIB) are in C:\TC\STARLIB. One of these libraries is PARX.LIB.
6. Your third-party-generated library files for quasars are in the A drive in \ASTROLIB. One of these libraries is WARP.LIB.

Under this configuration, you enter the following command:

```
TCC -mm -Llib;starlib -Iinclude orion.c umaj.c parx.lib a:\astrolib\warp.lib
```

Turbo C++ compiles ORION.C and UMAJ.C to .OBJ files, searching C:\TC\INCLUDE for any `#include` files in your source code. It then links ORION.OBJ and UMAJ.OBJ with the medium model start-up code (COM.OBJ), the medium model libraries (CM.LIB, MATHM.LIB), the standard floating-point emulation library (EMU.LIB), and the user-specified libraries (PARX.LIB and WARP.LIB), producing an executable file named ORION.EXE.

It searches for the startup code in C:\TC (then stops because they're there); it searches for the standard libraries in C:\TC\LIB (and stops because they're there).

When it searches for the user-specified library PARX.LIB, the compiler first looks in the current directory, C:\TC. Not finding the library there, the compiler then searches the library directories in order: first C:\TC\LIB, then C:\TC\STARLIB (where it locates PARX.LIB).

Since an explicit path is given for the library WARP.LIB (A:\ASTROLIB\WARP.LIB), the compiler only looks there.

MAKE: The program manager

Borland's command-line MAKE helps you keep the executable versions of your programs current. A program typically consists of many source files. Each file might have to pass through preprocessors, assemblers, compilers, and other utilities before being combined with the rest of the program. Forgetting to recompile a module that has changed or a module that's dependent on a changed module produces erroneous results. On the other hand, recompiling *everything* is time-consuming.

MAKE's usefulness extends beyond programming applications. MAKE controls any process that involves selecting files by name and processing them to produce a finished product, for example text processing, automatic backups, sorting files by extension into other directories, and cleaning temporary files out of your directory.

How MAKE works

MAKE keeps your program up-to-date by performing the following tasks:

- Reads a special file (called a makefile) that you created. The file tells MAKE which .OBJ and library files have to be linked in order to create your executable file and which source and header files have to be compiled to create each .OBJ file.

- Checks the time and date of each .OBJ file against the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the source file must be recompiled.
- Calls the compiler to recompile the source file.
- Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of your executable file.
- If any of the .OBJ files is later than the .EXE file, calls the linker to recreate the .EXE file.

MAKE relies upon the time stamp DOS places on files. For MAKE to do its job, your system's time and date *must* be set correctly. Check the system battery. Weak batteries can cause your system's clock to lose track of the date and time, and MAKE won't work right.

Starting MAKE

There are two versions of MAKE, MAKE.EXE, the protected-mode version and MAKER.EXE, the real-mode version. They work identically, except the protected-mode version processes larger makefiles. Throughout this document, MAKE refers to either version.

To use MAKE, type `make` at the DOS prompt.

MAKE looks for the following files until it finds one of them, or it halts and displays an error message:

BUILTINS.MAK (described later)
MAKEFILE
MAKEFILE.MAK

To use a file with a name other than MAKEFILE or MAKEFILE.MAK, give MAKE the file (`-f`) option, as shown in the following example:

```
MAKE -f MYFILE.MAK
```

The general syntax for MAKE is

```
make [option...] [target...]
```

where *option* is a MAKE option (discussed later), and *target* is the name of a target file to make.

Here are the MAKE syntax rules:

- The word *make* is followed by a space and a list of make options.
- Each make option must be separated from its adjacent options by a space. The order and number of options don't matter, as long as they fit on the command line. You can type a *-* or a *+* after options that don't specify a string, such as *-s*, to turn the option off or on, respectively.
- The list of MAKE options is followed by a space, then an optional list of targets.
- Each target must be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, re-compiling their constituents if necessary.

A *Ctrl-Break* stops the currently executing command and MAKE as well.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they are built if necessary.

Command-line options

The following table contains a list of MAKE's command-line options. MAKE is case-sensitive. For example, the option *-d* isn't a valid substitution for *-D*.

You can use either a *-* or a */* to introduce the options.

Table 9.1: MAKE options

Option	What it does
-? or -h	Prints a help message. The default options end with plus sign.
-a	Causes an automatic dependency check on .OBJ files.
-B	Builds all targets regardless of file dates.
-ddirectory	When used with the -S options, tells MAKE to write its swap file in the specified directory. <i>directory</i> can include a drive letter. Has no effect with the protected-mode version of MAKE.
-Didentifier	Defines the named identifier to the string consisting of the single character 1 (one).
[-D]iden=string	Defines the named identifier <i>iden</i> to the string after the equal sign. If the string contains any spaces or tabs, it must be enclosed in quotes. The -D option is optional.
-e	Ignores any attempt to redefine a macro whose name is the same as an environment variable. (In other words, causes the environment variable to take precedence.)
-f filename	Uses <i>filename</i> as the MAKE file. If <i>filename</i> does not exist and no extension is given, tries <i>filename</i> .MAK. The space after the -f is optional.
-i	Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. This is equivalent to putting '-' in front of all commands in the MAKEFILE (described below).
-ldirectory	Searches for include files in the indicated directory (as well as in the current directory).
-K	Keeps (does not erase) temporary files created by MAKE. All temporary files have the form MAKE $nnnn$.\$\$\$, where <i>nnnn</i> ranges from 0000 to 9999. See page 305 for more on temporary files.
-m	Displays the date and time stamp of each file as MAKE processes it.
-n	Prints the commands but does not actually perform them. This is useful for debugging a makefile.
-N	Increases MAKE's compatibility by resolving conflicts between MAKE's syntax and the syntax of Microsoft's NMAKE. See the rest of this chapter and Chapter 17, "Converting from Microsoft C" for the exact differences.
-p	Displays all macro definitions, implicit rules, and macro definitions before executing the makefile.
-r	Ignores the rules (if any) defined in BUILTINS.MAK.
-s	Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed.
-S	Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules. This option has no effect on the protected-mode version of MAKE.
-Uidentifier	Undefines any previous definitions of the named identifier.
-W	Writes the current specified non-string options (like -s and -a) to MAKE.EXE. (This makes them default.)

The BUILTINS.MAK file

Certain MAKE macros and rules are used more frequently than others and can be handled in the following ways:

- First, you can put them in every makefile you create.
- Second, you can put them all in one file and use the **!include** directive in each makefile you create. (See page 322 for more on directives.)
- Third, you can put them all in a BUILTINS.MAK file.

As previously mentioned, each time you run MAKE, it looks for a BUILTINS.MAK file; however, there is no requirement that any BUILTINS.MAK file exist.

MAKE searches for BUILTINS.MAK in the current directory first. Next, MAKE searches the directory where MAKE.EXE was invoked. Put BUILTINS.MAK in the same directory as the MAKE.EXE file.

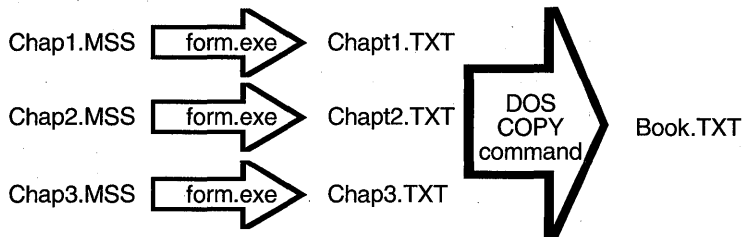
MAKE searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. Both BUILTINS.MAK and the makefile files have identical syntax rules.

MAKE also searches for any **!include** files (see page 324 for more information) in the current directory. If you use the **-I** (include) option, it searches the directory specified with the **-I** option, also.

A simple use of MAKE

MAKE can also back up files, pull files out of different subdirectories, and even automatically run your programs if the associated data files are modified.

Let's look at an example of using MAKE that doesn't involve a C program. Suppose you're writing a book and putting each chapter of the manuscript in a separate file. The book has three chapters in the files CHAP1.MSS, CHAP2.MSS, and CHAP3.MSS. To produce a current draft of the book, you run each chapter through a formatting program, called FORM.EXE, then use the DOS COPY command to concatenate the outputs to make a single file containing the draft, as represented by the following diagram:



As work on the book progresses, you modify one or more of the manuscript files. Next, you create a file, `MAKEFILE`, which tells `MAKE` what files `BOOK.TXT` depends on and how to process them. This file contains rules that explain how to rebuild `BOOK.TXT` when some of the files it depends on have been changed.

The first line in your makefile is the following rule:

```
BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT
COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

The line beginning with `BOOK.TXT:` says that `BOOK.TXT` depends on the formatted text of each of the three chapters. If any of the files that `BOOK.TXT` depends on are newer than `BOOK.TXT` itself, `MAKE` must rebuild `BOOK.TXT` by executing the `COPY` command on the subsequent line.

Furthermore, each of the chapter files depends on a manuscript (`.MSS`) file. If any of the `CHAP?.TXT` files is newer than the corresponding `.MSS` file, the `.MSS` file must be recreated. Thus, you need to add more rules to the makefile as follows:

```
CHAP1.TXT: CHAP1.MSS
FORM CHAP1.MSS

CHAP2.TXT: CHAP2.MSS
FORM CHAP2.MSS

CHAP3.TXT: CHAP3.MSS
FORM CHAP3.MSS
```

The rules show how to format one of the chapters, if necessary, from the original manuscript file.

If the first file depends on the second file, `MAKE` updates the second file before updating the first file. For example, if you

change CHAP3.MSS, MAKE reformats Chapter 3 before combining the .TXT files to create BOOK.TXT.

Explicit and implicit rules are discussed following the section on commands.

We can refine the example with an *implicit* rule. An implicit rule shows how to make one type of file from another, based on the files' extensions. In this case, you can replace the three rules for the chapters with one implicit rule:

```
.MSS.TXT:  
    FORM $*.MSS
```

The rule says "If you need to make a .TXT file out of an .MSS file to make things current, here's how to do it." (You'll still have to update the first rule—the one that makes BOOK.TXT, so that MAKE knows to concatenate the new chapters into the output file. This rule makes use of a *macro*. See page ??? for an in-depth discussion of macros.)

Once you create the makefile, type MAKE at the DOS command line to update the book.

Creating makefiles

Creating a program from an assortment of program files, include files, header files, object files, and so on, is similar to the previous text-processing example. The main difference is that the commands used at each step of the process invoke preprocessors, compilers, assemblers, and linkers instead of a text formatter and the DOS COPY command. Let's explore how to create makefiles—the files that tell MAKE what to do—in greater depth.

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up-to-date. You can create as many makefiles as you want and name them whatever you want; MAKEFILE is the name that MAKE looks for by default (if you don't specify a makefile) when you run MAKE.

You create a makefile with any ASCII text editor, such as the IDE built-in editor. All rules, definitions, and directives end at the end of a line. If a line is too long, you can continue it to the next line by placing a backslash (\) at the end of the line.

Use whitespace (blanks and tabs) to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Components of a makefile

Creating a makefile is like writing a program, with definitions, commands, and directives. The following constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules
- macros
- directives:
 - file inclusion directives
 - conditional execution directives
 - error detection directives
 - macro undefinition directives

Let's look at each of these in more detail.

Comments

Comments begin with a pound sign (#) character; the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere; they don't have to start in a particular column.

A backslash does *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. If the backslash precedes the #, it is no longer the last character on the line; if it follows the #, then it is part of the comment itself.

Here are some examples of comments in a makefile:

```
# Makefile for my book

# This file updates the file BOOK.TXT each time I
# change one of the .MSS files

# Explicit rule to make BOOK.TXT from six chapters. Note the
# continuation lines.

BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT\
          CHAP4.TXT CHAP5.TXT CHAP6.TXT
          COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT+CHAP4.TXT+\
              CHAP5.TXT+CHAP6.TXT BOOK.TXT

# Implicit rule to format individual chapters
```

```
.MSS.TXT:
FORM $*.MSS
```

Command lists for implicit and explicit rules

Both explicit and implicit rules can have lists of commands. This section describes how these commands are processed by MAKE.

Commands in a command list take the form

`[prefix ...] command_body`

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

Prefixes

The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at-sign (@) or a hyphen (-) followed immediately by a number.

Table 9.2
MAKE prefixes

Prefix	What it does
@	Prevents MAKE from displaying the command before executing it. The display is hidden even if the <code>-s</code> option is not given on the MAKE command line. This prefix applies only to the command on which it appears.
<code>-num</code>	Affects how MAKE treats exit codes. If a number (<i>num</i>) is provided, then MAKE aborts processing only if the exit status exceeds the number given. In this example, MAKE aborts only if the exit status exceeds 4: <code>-4 MYPROG SAMPLE.X</code> If no <code>-num</code> prefix is given and the status is nonzero, MAKE stops and deletes the current target file.
-	With a hyphen but no number, MAKE doesn't check the exit status at all. Regardless of the exit status, MAKE continues.
&	The ampersand operator causes MAKE to execute the command for each of the dependents the <code>\$\$*</code> or <code>\$\$?</code> macros in an explicit rule expands to. See page 321 for more information on these macros.

Exit codes are status codes returned by the executed commands.

Command body and operators

The command body is treated exactly as if it were entered as a line to the DOS command line, with the exception that pipes (|) are not supported.

In addition to the `<`, `>`, and `>>` redirection operators, MAKE adds the `<<` and `&&` operators. These operators create a file on the fly for input to a command. The `<<` operator creates a temporary file and redirects the command's standard input so that it comes from

the created file. If you have a program that accepts input from *stdin*, the command

```
MYPROG <<!
This is a test
!
```

would create a temporary file containing the string "This is a test \n", redirecting it to be the sole input to *myprog*. The exclamation point (!) is a delimiter in this example; you can use any character except # or \ as a delimiter for the file. The first line containing the delimiter character as its first character ends the file. The rest of the line following the delimiter character (in this case, an exclamation point) is considered part of the preceding command.

The **&&** operator is similar to **<<**. It creates a temporary file, but instead of making the file the standard input to the command, the **&&** operator is replaced with the temporary file's name. This is useful when you want MAKE to create a file that's going to be used as input to a program. The following example creates a "response file" for TLINK:

Macros are covered starting on page 315.

```
MYPROG.EXE: $(MYOBS)
TLINK /c @&&!
COS $(MYOBS)
$*
$*
$(MYLIBS) EMU.LIB MATHS.LIB CS.LIB
!
```

Note that macros (indicated by \$ signs) are expanded when the file is created. The \$* is replaced with the name of the file being built, without the extension, and \$(MYOBS) and \$(MYLIBS) are replaced with the values of the macros MYOBS and MYLIBS. Thus, TLINK might see a file that looks like this:

```
COS A.OBJ B.OBJ C.OBJ D.OBJ
MYPROG
MYPROG
W.LIB X.LIB Y.LIB Z.LIB EMU.LIB MATHS.LIB CS.LIB
```

The KEEP option for the << operator in compatibility mode tells MAKE not to delete specific temporary files. See the next section.

All temporary files are deleted unless you use the **-K** command-line option. Use the **-K** option to "debug" your temporary files if they don't appear to be working correctly.

Compatibility option

If you specified **-N** on the MAKE command line, the **<<** operator changes its behavior to be more like that of the **&&** operator; that is, the temporary file isn't redirected to standard input, it's just created on the fly for use mainly as a response file. This behavior is consistent with Microsoft's NMAKE.

The format for this version of the **<<** operator is:

```
command <<[filename1] ... <<[filenameN]
text
:
<<[KEEP | NOKEEP]
text
:
<<[KEEP | NOKEEP]
```

Note that there must be no space after the << and before the KEEP or NOKEEP option.

The **KEEP** option tells MAKE to not delete the file after it's been used. If you don't specify anything or specify **NOKEEP**, MAKE deletes the temporary file (unless you specified the **-K** option to keep temporary files).

Batching programs MAKE allows utilities that can operate on a list of files to be batched. Suppose, for example, that MAKE needs to submit several C files to Turbo C++ for processing. MAKE could run TCC once for each file, but it's much more efficient to invoke TCC with a list of all the files to be compiled on the command line. This saves the overhead of reloading Turbo C++ each time.

MAKE's batching feature lets you accumulate the names of files to be processed by a command, combine them into a list, and invoke that command only once for the whole list.

To cause MAKE to batch commands, you use braces in the command line:

```
command { batch-item } ...rest-of-command
```

This command syntax delays the execution of the command until MAKE determines what command (if any) it has to invoke next. If the next command is identical except for what's in the braces, the two commands are combined by appending the parts of the commands that appeared inside the braces.

Here's an example that shows how batching works. Suppose MAKE decides to invoke the following three commands in succession:

```
TCC {file1.c }  
TCC {file2.c }  
TCC {file3.c }
```

Rather than invoking Turbo C++ three times, MAKE issues the single command

```
TCC file1.c file2.c file3.c
```

Note that the spaces at the ends of the file names in braces are essential to keep them apart, since the contents of the braces in each command are concatenated exactly as is.

Here's an example that uses an implicit rule. Suppose your makefile had an implicit rule to compile C programs to .OBJ files:

```
.c.obj:  
TCC -c {< }
```

As MAKE uses the implicit rule on each C file, it expands the macro \$< into the actual name of the file and adds that name to the list of files to compile. (Again, note the space inside the braces to keep the names separate.) The list grows until one of three things happens:

- MAKE discovers that it has to run a program other than TCC
- there are no more commands to process
- MAKE runs out of room on the command line

If MAKE runs out of room on the command line, it puts as much as it can on one command line, then puts the rest on the next command line. When the list is done, MAKE invokes TCC (with the `-c` option) on the whole list of files at once.

Executing commands MAKE searches for any other command name using the DOS search algorithm:

1. MAKE first searches for the file in the current directory, then searches each directory in the path.
2. In each directory, MAKE first searches for a file of the specified name with the extension .COM. If it doesn't find it, it searches for the same file name with an .EXE extension. Failing that, MAKE searches for a file by the specified name with a .BAT extension.
3. If MAKE finds a .BAT file, it invokes a copy of COM-MAND.COM to execute the batch file.

4. If MAKE can't find a .COM, .EXE, or .BAT file matching the command to be executed, it invokes a copy of the DOS command processor (COMMAND.COM by default) to execute the command.

If you supply a file name extension in the command line, MAKE searches only for that extension. Here are some examples:

- This command causes COMMAND.COM to change the current directory to C:\INCLUDE:

```
cd c:\include
```

- MAKE uses the full search algorithm in searching for the appropriate files to perform this command:

```
tlink lib\c0s x y,z,z,lib\cs
```

- MAKE searches for this file using only the .COM extension:

```
myprog.com geo.xyz
```

- MAKE executes this command using the explicit file name provided:

```
c:\myprogs\fil.exe -r
```

Explicit rules

The first rule in the example on page 304 is an explicit rule—a rule that specifies complete file names explicitly. Explicit rules take the form

Note that the braces must be included if you use the paths parameter.

```
target [target ... ] : [{paths}] [dependent ... ]  
[command]  
:
```

where *target* is the file to be updated, *dependent* is a file on which *target* depends, *paths* is a list of directories, separated by semicolons and enclosed in braces, in which dependent files might reside, and *command* is any valid DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more dependent files, and an optional list of commands to be performed. Target and dependent file names listed in explicit rules can contain normal DOS drive and directory specifications; they can also contain wildcards.



Syntax here is important.

- *target* must be at the start of a line (in column 1).

- The *dependent* file(s) must be preceded by at least one space or tab, after the colon.
- *paths*, if included, must be enclosed in braces.
- Each *command* must be indented, (must be preceded by at least one blank or tab). As mentioned before, the backslash can be used as a continuation character if the list of dependent files or a given command is too long for one line.

Both the dependent files and the commands are optional; it is possible to have an explicit rule consisting only of *target* [*target* ...] followed by a colon.

An explicit rule creates or updates *target*, usually using the *dependent* files, from one or more commands in the makefile. When MAKE encounters an explicit rule, it first checks to see if any of the *dependent* files are themselves target files elsewhere in the makefile. If so, MAKE evaluates that rule first.

MAKE checks for dependent files in the current directory first. If it can't find them, MAKE checks each of the directories specified in the path list.

Once all the *dependent* files have been created or updated based on other rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *dependent*. If any *dependent* has been modified more recently than *target*, the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or to the end of the file. Blank lines are ignored.

Special considerations

An explicit rule with no command lines following it is treated a little differently than an explicit rule with command lines.

- If an explicit rule includes commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on two sets of files: the files given in the explicit rule, and any file that

matches an implicit rule for the target(s). This lets you specify a dependency to be handled by an implicit rule. For example in

```
.c.obj:
    TCC -c $<
prog.obj:
```

PROG.OBJ depends on PROG.C; If PROG.OBJ is out of date, MAKE executes the command line

```
TCC -c prog.c
```

Multiple explicit rules for a single target

A single target may have more than a single explicit rule. You might use multiple explicit rules to create a module library with TLIB, for example, since the .OBJ object module files might be built differently (for example, some with TCC and some with an assembler such as TASM).

The format is the same as for normal explicit rules, except there are two colons following the target. The second colon tells MAKE to expect additional explicit rules for this target.

In the following example, MYLIB.LIB consists of four object modules, two of which are C++ modules. The other two are assembly modules. The first explicit rule compiles the C++ modules and updates the library. The second explicit rule assembles the ASM files and also updates the library.

```
mylib.lib:: f1.cpp f2.cpp
    tcc -c f1.cpp f2.cpp
    tlib mylib --f1.obj --f2.obj

mylib.lib:: f3.asm f4.asm
    tasm /mx f3.asm f4.asm
    tlib mylib --f3.obj --f4.obj
```

Examples Here are some examples of explicit rules:

1. prog.exe: myprog.obj prog2.obj
TCC myprog.obj prog2.obj
2. myprog.obj: myprog.c include\stdio.h
TCC -c myprog.c
3. prog2.obj: prog2.c include\stdio.h
TCC -c -K prog2.c

The three examples are from the same makefile. Only the modules affected by a change are rebuilt. If PROG2.C is changed, it's the only one recompiled; the same holds true for MYPROG.C. But if

the include file `stdio.h` is changed, both are recompiled. (The link step is done if any of the `.OBJ` files in the dependency list have changed, which happens when a recompile results from a change to a source file.)

Automatic dependency checking

Turbo C++ works with MAKE to provide automatic dependency checking for include files. TC produces `.OBJ` files that tell MAKE what include files were used to create those `.OBJ` files. MAKE's `-a` command-line option checks this information and makes sure that everything is up-to-date.

When MAKE does an automatic dependency check, it reads the include files' names, times, and dates from the `.OBJ` file. The autodependency check also works for include files inside of include files. If any include files have been modified, MAKE causes the `.OBJ` file to be recompiled. For example, consider the following explicit rule:

```
myprog.obj: myprog.c include\stdio.h
    TCC -c myprog.c
```

Now assume that the following source file, called `MYPROG.C`, has been compiled with TC:

```
#include <stdio.h>
#include "dcl.h"

void myprog() {}
```

If you then invoke MAKE with the following command line

```
make -a myprog.obj
```

it checks the time and date of `MYPROG.C`, and also of `stdio.h` and `dcl.h`.

Implicit rules

MAKE allows you to define *implicit* rules as well as explicit ones. Implicit rules are generalizations of explicit rules; they apply to all files that have certain identifying extensions.

Here's an example that illustrates the relationship between the two rules. Consider this explicit rule from the preceding example. The rule is typical because it follows a general principle: An `.OBJ` file is dependent on the `.C` file with the same file name and is created by executing TC. In fact, you might have a makefile where

you have several (or even several dozen) explicit rules following this same format.

By rewriting the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.c.obj:  
TCC -c $<
```

The symbol `$<` is a special macro. Macros are discussed starting on page 315. The `$<` macro is replaced by the full name of the appropriate `.C` source file each time the command executes.

This rule means “Any file with the extension `.C` can be translated to a file of the same name with the extension `.OBJ` using this sequence of commands.” The `.OBJ` file is created with the second line of the rule, where `$<` represents the file’s name with the source (`.C`) extension.

Here’s the syntax for an implicit rule:

```
[[source_dir]].source_extension. [[target_dir]]target_extension:  
[command]  
⋮
```

As before, the commands are optional and must be indented.

`source_dir` (which must be enclosed by braces) tells MAKE to search for source files in the specified directory. `target_dir` gives MAKE a location for the target files.

`source_extension` is the extension of the source file; that is, it applies to any file having the format

```
fname.source_extension
```

Likewise, the `target_extension` refers to the file

```
fname.target_extension
```

where `fname` is the same for both files. In other words, this implicit rule replaces all explicit rules having the format

```
fname.target_extension: fname.source_extension  
[command]  
⋮
```

for any `fname`.



MAKE uses implicit rules if it can’t find any explicit rules for a given target, or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is

found with the same name as the target, but with the mentioned source extension.

For example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.c.obj:  
    TCC -c $<
```

If you had a C program named RATIO.C that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
TCC -c ratio.c
```

which, of course, does the compile step necessary to create RATIO.OBJ.

MAKE also uses implicit rules if you give it an explicit rule with no commands. Suppose you had the following implicit rule at the start of your makefile:

```
.c.obj:  
    TCC -c $<
```

You could then remove the command from the rule:

```
myprog.obj: myprog.c include\stdio.h  
            TCC -c myprog.c
```

and it would execute exactly as before.

If you're using Turbo C++ and you enable automatic dependency checking in MAKE, you can remove all explicit dependencies that have .OBJ files as targets. With automatic dependency checking enabled and implicit rules, the three-rule C example shown in the section on explicit rules becomes

```
.c.obj:  
    TCC -c $<  
  
prog.exe: myprog.obj prog2.obj  
         tlink lib\c0s myprog prog2, prog, , lib\cs
```

Note that with the **-N** compatibility option, the searches go in the **opposite** direction: from the bottom of the makefile up.

You can write several implicit rules with the same target extension. If more than one implicit rule exists for a given target extension, the rules are checked in the order in which they appear in the makefile, until a match is found for the source extension, or until MAKE has checked all applicable rules.

MAKE uses the first implicit rule that involves a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule, up to the next line that begins without whitespace or to the end of the file, are considered to be part of the command list for the rule.

Macros

Often, you'll find yourself using certain commands, file names, or options again and again in your makefile. For instance, if you're writing a C program that uses the medium memory model, all TC commands use the option **-mm**, which means to compile to the medium memory model. But suppose you wanted to switch to the large memory model. You could go through and change all the **-mm** options to **-ml**. Or, you could define a macro.

A *macro* is a name that represents some string of characters. A macro definition gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
MODEL = m
```

This line defines the macro `MODEL`, which is now equivalent to the string `m`. Using this macro, you could write each command to invoke the C compiler to look something like this:

```
TCC -c -m$(MODEL) myprog.c
```

When you run MAKE, each macro (in this case, `$(MODEL)`) is replaced with its expansion text (here, `m`). The command that's actually executed would be

```
TCC -c -mm myprog.c
```

Now, changing memory models is easy. If you change the first line to

```
MODEL = 1
```

you've changed all the commands to use the large memory model. In fact, if you leave out the first line altogether, you can specify which memory model you want each time you run MAKE, using the **-D** (define) command-line option:

```
make -DMODEL=1
```

This tells MAKE to treat **MODEL** as a macro with the expansion text *l*.

Defining macros Macro definitions take the form

```
macro_name = expansion text
```

where *macro_name* is the name of the macro. *macro_name* should be a string of letters and digits with no whitespace in it, although you can have whitespace between *macro_name* and the equal sign (=). The *expansion text* is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by newline.

If *macro_name* has previously been defined, either by a macro definition in the makefile or on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macro names **model**, **Model**, and **MODEL** are all different.

Using macros You invoke macros in your makefile using this format

```
$(macro_name)
```

You need the parentheses for all invocations, except when the macro name is just one character long. This construct—*\$(macro_name)*—is known as a *macro invocation*.

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Using environment variables as macros

If you invoke a macro where *macro_name* hasn't been defined in the makefile or on the command line, MAKE tries to find *macro_name* as a DOS environment variable. If MAKE finds it in the environment, the expansion text is the value of the environment variable.



Macros defined in the makefile or on the command line override environment variables of the same name unless the `-e` option was specified.

Substitution within macros

You can invoke a macro while simultaneously changing some of its text by using a special format of the macro invocation format. Instead of the standard macro invocation form, use

```
$(macro_name:text1=text2)
```

In this form, every occurrence of *text1* in *macro_name* is replaced with *text2*. *macro_name* can also be one of the predefined macros. This is useful if you'd prefer to keep only a single list of files in a macro. For example, in the following example, the macro **SOURCE** contains a list of all the C++ source files a target depends on. The TLINK command line changes all the .CPP extensions to the matching .OBJ object files and links those.

Note that no extraneous whitespace should appear between the : and =. If spaces appear after the colon, MAKE attempts to find the string, including the preceding space.

```
SOURCE = f1.cpp f2.cpp f3.cpp
myapp.exe: $(SOURCE)
tcc -c $(SOURCE)
tlink c0s $(SOURCE:.cpp=.obj),myapp,,cs
```

Special considerations

Macros in macros: Macros cannot be invoked on the left side (*macro_name*) of a macro definition. They can be used on the right side (*expansion text*), but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

See page 322 for information on directives.

Macros in directives: Macro invocations are expanded immediately in **!if** and **!elif** directives. If the macro being invoked in an **!if** or **!elif** directive is not currently defined, it is expanded to the value 0 (FALSE).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

Predefined macros

MAKE comes with several special macros built in: **\$d**, **\$***, **\$<**, **\$:**, **\$.**, **\$&**, **\$@**, ******, and **\$?**. The first is a test to see if a macro name is defined; it's used in the conditional directives **!if** and **!elif**. The others are file name macros, used in explicit and implicit rules. Finally, MAKE defines several other macros; see Table 9.3.

Table 9.3
MAKE predefined macros

__MSDOS__	"1" if running MAKE under DOS
__MAKE__	MAKE's version number in hexadecimal (for this version, "0x0360")
MAKE	MAKE's executable filename, (usually MAKE or MAKER)
MAKEFLAGS	Any options used on the MAKE command line
MAKEDIR	The directory from which MAKE was run

Table 9.4
MAKE filename macros

Macro	What part of the file name it returns in an	
	implicit rule	explicit rule
\$*	Dependent base with path	Target base with path
\$<	Dependent full with path	Target full with path
\$:	Dependent path only	Target path only
\$.	Dependent full without path	Target full without path
\$&	Dependent base without path	Target base without path
\$@	Target full with path	Target full with path
**	Dependent full with path	All dependents
\$?	Dependent full with path	All out of date dependents

Defined Test Macro (\$d)

The defined test macro (**\$d**) expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in **!if** and **!elif** directives.

For example, suppose you want to modify your makefile so that if you don't specify a memory model, it'll use the medium one. You could put this at the start of your makefile:

```

!if !$d(MODEL) # if MODEL is not defined
MODEL=m      # define it to m (MEDIUM)
!endif

```

If you then invoke MAKE with the command line

```
make -DMODEL=1
```

then **MODEL** is defined as *1*. If, however, you just invoke MAKE by itself,

make

then **MODEL** is defined as *m*, your “default” memory model.

File name macros The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built.

Base file name macro (\$*)

The base file name macro is allowed in the commands for an explicit or an implicit rule. This macro (**\$***) expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.C
$* expands to A:\P\TESTFILE
```

For example, you could modify this explicit rule

```
prog.exe: myprog.obj prog2.obj
         tlink lib\c0s myprog prog2, prog, , lib\cs
```

to look like this:

```
prog.exe: myprog.obj prog2.obj
         tlink lib\c0s myprog prog2, $*, , lib\cs
```

When the command in this rule is executed, the macro **\$*** is replaced by the target file name without an extension and with a path. For implicit rules, this macro is very useful.

For example, an implicit rule might look like this:

```
.cpp.obj:
         TCC -c $*
```

Full file name macro (\$<)

The full file name macro (**\$<**) is also used in the commands for an explicit or implicit rule. In an explicit rule, **\$<** expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.C
$< expands to A:\P\TESTFILE.C
```

For example, the rule

```
mylib.obj: mylib.c
         copy $< \oldobjs
         TCC -c $*.c
```

copies MYLIB.OBJ to the directory \OLDOBJs before compiling MYLIB.C.

In an implicit rule, $\$<$ takes on the file name plus the source extension. For example, the implicit rule

```
.c.obj:  
    TCC -c $*.c
```

produces exactly the same result as

```
.c.obj:  
    TCC -c $<
```

because the extension of the target file name *must* be .C.

File name path macro (\$:)

This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.C  
$: expands to A:\P\
```

File name and extension macro (\$.)

This macro expands to the file name, with an extension but without the path name, like this:

```
File name is A:\P\TESTFILE.C  
$. expands to TESTFILE.C
```

File name only macro (\$&)

This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.C  
$& expands to TESTFILE
```

Full target name with path macro (\$@)

This macro expands to the full target file name with path and extension, like this:

```
File name is A:\P\TESTFILE.C  
$@ expands to A:\P\TESTFILE.C
```

The `$@` macro is similar to the `$<` macro, except that `$@` expands to the full target file name in *both* implicit and explicit rules, which expands to the target in an explicit rule and the dependent in an implicit rule.

All dependents macro (`$**`)

In an explicit rule, this macro expands to all the dependents of the target, including the full filename with path and extension. For example, in the following explicit rule, the `$**` is replaced with `myprog.obj prog2.obj`, the two dependents of `prog.exe`:

```
prog.exe: myprog.obj prog2.obj
        tlink lib\c0s $**, $*, , lib\cs
```

All out of date dependents macro (`$?`)

In an explicit rule, this macro expands to all the out of date dependents of the target, including the full filename with path and extension. Out of date dependents are those that have been modified since the target was last made. Therefore, in the following example explicit rule, the `$?` is replaced with `f1.cpp` and/or `f2.cpp` depending on which dependent(s) were out of date:

```
mylib.lib: f1.cpp f2.cpp
        tcc -c $?
        &tlib mylib -+$(?:.cpp=.obj)
```

Note the use of the `&` prefix so MAKE repeats the command for each of the out-of-date dependents.

Macro modifiers If there isn't a predefined filename macro to give you the parts of a filename you need, you can use *macro modifiers* to extract any part of a filename macro. The format is:

`$(macro[D | F | B | R])`

where *macro* is one of the predefined filename macros and D, F, B, and R are the modifiers. Note that since the macro is now longer than a single character, parentheses are necessary. The following table describes what each modifier does. The examples assume that `$<` returns `C:\OBJ\BOB.OBJ`.

Table 9.5
MAKE macro modifiers

Modifier	What part of the filename	Example
D	Drive and directory	\$(<D) = C:\OBJJS\
F	Base and extension	\$(<F) = BOB.OBJ
B	Base only	\$(<B) = BOB
R	Drive, directory, and base	\$(<R) = C:\OBJJS\BOB

Directives

Turbo's MAKE allows something that other versions of MAKE don't: directives similar to those allowed in C, assembler, and Turbo Pascal. You can use these directives to perform a variety of useful and powerful actions. Some directives in a makefile begin with an exclamation point (!) as the first character of the line. Others begin with a period. Here is the complete list of MAKE directives:

Table 9.6
MAKE directives

.autodepend	Turns on autodependency checking.
!elif	Conditional execution.
!else	Conditional execution.
!endif	Conditional execution.
!error	Causes MAKE to stop and print an error message.
!if	Conditional execution.
!ifdef	Conditional execution.
!ifndef	Conditional execution.
.ignore	Tells MAKE to ignore return value of a command.
!include	Specifies a file to include in the makefile.
.noautodepend	Turns off autodependency checking.
.noignore	Turns off .ignore .
.nosilent	Tells MAKE to print commands before executing them.
.noswap	Tells the real mode version of MAKE, MAKER, to not swap itself in and out of memory. Has no effect in the protected-mode version of MAKE.
.path.ext	Gives MAKE a path to search for files with extension .EXT .
.precious	Tells MAKE to not delete the specified target even if the commands to build the target fail.
.silent	Tells MAKE to not print commands before executing them.
.swap	Tells the real mode version of MAKE, MAKER, to swap itself in and out of memory. Has no effect in the protected-mode version of MAKE.
.suffixes	Tells MAKE the implicit rule to use when a target's dependent is ambiguous.
!undef	Causes the definition for a specified macro to be forgotten.

Dot directives

Each of the following directives has a corresponding command-line option, but takes precedence over that option. For example, if you invoke MAKE with the following command, and if the makefile has a **.noautodepend** directive, then autodependency checking is turned off:

```
make -a
```

.autodepend and **.noautodepend** turn on or off autodependency checking. They correspond to the **-a** command-line option.

.ignore and **.noignore** tell MAKE whether or not to ignore the return value of a command, much like placing the prefix **-** in front of it (described earlier). They correspond to the **-i** command-line option.

.silent and **.nosilent** tell MAKE whether or not to print commands before executing them. They correspond to the **-s** command-line option.

.swap and **.noswap** tell MAKE whether or not to swap itself out of memory. They correspond to the **-S** option.

.precious The syntax for the **.precious** directive is:

```
.precious: target [...]
```

where *target* is one or more target files. **.precious** prevents MAKE from deleting the target if the commands building the target fail. In some cases, the target is still viable. For example, if an object module can't be added to a library, the library shouldn't be deleted.

.path.ext This directive, placed in a makefile, tells MAKE where to look for files of the given extension. For example, if the following instructions are in a makefile:

```
.path.c = C:\CSOURCE  
.c.obj:  
    TCC -c $<  
tmp.exe: tmp.obj  
    TCC tmp.obj
```

MAKE looks for TMP.C, the implied source file for TMP.OBJ, in C:\CSOURCE instead of the current directory.

The **.path** is also a macro that has the value of the path. The following is an example of the use of **.path**. The source files are contained in one directory, the .OBJ files in another, and all the .EXE files in the current directory.

```
.path.c = C:\CSOURCE
.path.obj = C:\OBSJ
.c.obj:
    TCC -c -o$(.path.obj)\$& $<
.obj.exe:
    TCC -e$&.exe $<
tmp.exe: tmp.obj
```

.suffixes In the following example, MYPROG.OBJ can be created from MYPROG.ASM, MYPROG.CPP, and MYPROG.C.

```
myprog.exe: myprog.obj
    tlink myprog.obj
.asm.obj:
    tasm /mx $<
.cpp.obj:
    tcc -P $<
.c.obj:
    tcc -P- $<
```

If more than one of these sources is available, the **.suffixes** directive determines which are used. The syntax for **.suffixes** is:

```
.suffixes: source_extension ...
```

where *source_extension* is a list of the extensions for which there are implicit rules, in order of which source extension implicit rule should be used.

For example, if we add **.suffixes: .asm .c .cpp** to the top of the previous makefile example, MAKE would first look for MYPROG.ASM, then MYPROG.C, and finally MYPROG.CPP.

File-inclusion directive

A file-inclusion directive (**!include**) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

!include filename

filename can be surrounded by quotes ("*filename*") or angle brackets (<*filename*>). You can nest these directives to any depth. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file MODEL.MAC that contained the following:

```
!ifndef MODEL
MODEL=m
!endif
```

You could use this conditional macro definition in any makefile by including the directive.

```
!include "MODEL.MAC"
```

When MAKE encounters **!include**, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional execution directives

Conditional execution directives (**!if**, **!ifdef**, **!ifndef**, **!elif**, **!else**, and **!endif**) give you a measure of flexibility in constructing makefiles. Rules and macros can be made conditional, so that a command-line macro definition (using the **-D** option) can enable or disable sections of the makefile.

The format of these directives parallels those in C, assembly language, and Turbo Pascal:

```
!if expression
[lines]
:
!endif

!if expression
[lines]
:
!else
[lines]
:
!endif

!if expression
[lines]
```



```

:
!elif expression
[lines]
:
!endif

!ifdef macro
[lines]
:
!endif

!ifndef macro
[lines]
:
!endif

```

[lines] can be any of the following statement types:

- macro definition
- explicit rule
- implicit rule
- include directive
- if group
- error directive
- undef directive

The conditional directives form a group, with at least an **!if**, **!ifdef**, or **!ifndef** directive beginning the group and an **!endif** directive closing the group.

- One **!else** directive can appear in the group.
- **!elif** directives can appear between the **!if** (or **!ifdef** and **!ifndef**) and any **!else** directives.
- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested to any depth.

Any rules, commands, or directives must be complete within a single source file.

All **!if**, **!ifdef**, and **!ifndef** directives must have matching **!endif** directives within the same source file. Thus the following include file is illegal, regardless of what's in any file that might include it, because it doesn't have a matching **!endif** directive:

```

!if $(FILE_COUNT) > 5
    some rules
!else
    other rules
<end-of-file>

```

The **lifdef** directive is another way of testing whether a macro is defined. `!lifdef MACRO` is equivalent to `!if $d(MACRO)`. The same holds true for **!lifndef**; `!lifndef MACRO` is equivalent to `!if ! $d(MACRO)`.

Expressions allowed in conditional directives

Expressions are allowed in an **lif** or an **!lif** directive; they use a C-like syntax. The expression is evaluated as a simple 32-bit signed integer or strings of characters.

You can enter numbers as decimal, octal, or hexadecimal constants. If you know the C language, you already know how to write constants in MAKE; the formats are exactly the same. These are legal constants in a MAKE expression:

```

4536 # decimal constant
0677 # octal constant (distinguished by leading 0)
0x23aF # hexadecimal constant (distinguished by leading 0x)

```

Any expression that doesn't follow one of those formats is considered a string.

An expression can use any of the following operators (an asterisk indicates the operator is also available with string expressions):

Table 9.7
MAKE operators

See Chapter 12 for complete descriptions of these operators.

Operator	Operation	Operator	Operation
<i>Unary operators</i>			
-	Negation	&	Bitwise AND
~	Bit complement		Bitwise OR
!	Logical NOT	^	Bitwise XOR
		&&	Logical AND
			Logical OR
<i>Binary operators</i>			
+	Addition	>	Greater than*
-	Subtraction	<	Less than*
*	Multiplication	>=	Greater than or equal*
/	Division	<=	Less than or equal*
%	Remainder	==	Equality*
>>	Right shift	!=	Inequality*
<<	Left shift	<i>Ternary operator</i>	
		?:	Conditional expression

The operators have the same precedences as they do in the C language. Parentheses can be used to group operands in an expression. Unlike the C language, MAKE can compare strings using the normal ==, !=, >, <, >=, and => operators. You can't compare a string expression with a numeric expression, nor can you use numeric operators (like + or *) with strings.

A string expression may contain spaces, but if it does it must be enclosed in quotes:

```
Model = "Medium model"
:
!if $(Model) == "Medium model"
  CFLAGS = -mm
!elif $(Model) == "Large model"
  CFLAGS = -ml
!endif
```

You can invoke macros within an expression; the special macro **\$d()** is recognized. After all macros have been expanded, the expression must have proper syntax.

Error directive

The error directive (**!error**) causes MAKE to stop and print a fatal diagnostic containing the text after **!error**. It takes the format

```
!error any_text
```

This directive is designed to be included in conditional directives to allow a user-defined error condition to abort MAKE. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(MODEL)
# if MODEL is not defined
!error MODEL not defined
!endif
```

If you reach this spot without having defined **MODEL**, then MAKE stops with this error message:

```
Fatal makefile 4: Error directive: MODEL not defined
```

Macro undefinition directive

The macro “undefinition” directive (**!undef**) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is

```
!undef macro_name
```

The compatibility option **-N**

The **-N** command line option increases compatibility with Microsoft's NMAKE. You should use it only when you need to build a project using makefiles created for NMAKE tools. Running MAKE without the **-N** option is preferred, since **-N** introduces some subtle differences in how makefiles work:

- `$$` expands to a single `$` and a single `$` expands to nothing.
- The caret character `^` causes the character that follows, if a special character, to be treated literally. For example,

```
TEST = this is ^  
a test
```

causes **TEST** to expand to `this is \na test` where the `\n` is the C symbol for a new line. It's especially useful when you need to end a line with the line continuation character:

```
SOURCEDIR = C:\BOB\OBJS^\
```



- If the caret is followed by a normal character (one without a special meaning), the caret is ignored.
- The **\$d** macro won't be the special defined test macro. Use the **!ifdef** and **!ifndef** directives instead.
- Predefined macros that return paths only do *not* end in a trailing backslash. For example, without the **-N** switch `$(<D)` might return `C:\OBJS\`, but with the **-N** switch, the same `$(<D)` macro would return `C:\OBJS`.

- Unless there's a matching **.suffixes** directive, MAKE searches for implicit rules from the bottom of the makefile up.
- The **\$*** macro always expands to the target name. (In normal mode, **\$*** expands to the dependent in an implicit rule.)

TLINK: The Turbo linker

Appendix C, "Error messages," lists linker messages generated by TLINK and by the built-in IDE linker.

The IDE has its own built-in linker. When you invoke the command-line compiler TCC, TLINK is invoked automatically unless you suppress the linking stage. If you suppress the linking stage, you must invoke TLINK manually. This chapter describes how to use TLINK as a standalone linker.

By default, the command-line compiler calls TLINK when compilation is successful; TLINK then combines object modules and library files to produce the executable file.

Invoking TLINK

Note that this version of TLINK is sensitive to the case of its options; /f is not the same option as /F.

You can invoke TLINK at the command line by typing `tlink` with or without parameters. When invoked without parameters, TLINK displays a summary of parameters and options. Table 10.1 briefly describes the TLINK options.

Table 10.1
TLINK options

Option	What it does
<code>/3</code>	Enables processing of 32-bit modules.
<code>/c</code>	Treats case as significant in symbols.
<code>/d</code>	Warns if duplicate symbols in libraries.
<code>/e</code>	Ignores Extended Dictionary.
<code>/i</code>	Initializes all segments.
<code>/l</code>	Includes source line numbers.
<code>/L</code>	Specifies library search paths.
<code>/m</code>	Creates map file with publics.
<code>/n</code>	Doesn't use default libraries.

You can use either a hyphen or a slash to precede TLINK's commands.

Table 10.1: TLINK options (continued)

/b	Overlays following modules or libraries.
/P	Packs code segments.
/s	Creates detailed map of segments.
/t	Generates .COM file. (Also /Tdc.)
/Td	Creates target DOS executable.
/Tdc	Creates target DOS .COM file.
/Tde	Creates target DOS .EXE file.
/v	Includes full symbolic debug information.
/x	Doesn't create map file.
/ye	Uses expanded memory for swapping.
/yx	Configures TLINK's use of extended memory.

The general syntax of a TLINK command line is

TLINK *objfiles, exe file, mapfile, libfiles*

This syntax specifies that you supply file names *in the given order*, separating the file *types* with commas.

An example of linking

If you supply the TLINK command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

TLINK will interpret it to mean that

- Case is significant during linking (**/c**).
- The .OBJ files to be linked are MAINLINE.OBJ, WD.OBJ, LN.OBJ, and TX.OBJ.
- The executable program name will be FIN.EXE.
- The map file is MFIN.MAP.
- The library files to be linked in are COMM.LIB and SUPPORT.LIB, both of which are in subdirectory WORK\LIB.

The **/c** option tells TLINK to be case-sensitive during linking.

File names on the TLINK command line

If you don't specify an executable file name, TLINK derives the name of the executable by appending .EXE to the first object file name listed.

If you specify a complete file name for the executable file, TLINK will create the file with that name, but the actual nature of that executable depends on other options.

If no map file name is given, TLINK adds a .MAP extension to the .EXE file name. If no libraries are included, none will be linked.

TLINK assumes or appends extensions to file names that have none:

- .OBJ for object files
- .EXE executable files when you use the */t* option; the executable file extension defaults to .COM rather than .EXE
- .MAP for map files
- .LIB for library files

All of the file names *except* object files are optional. So, for instance,

```
TLINK app app2
```

links the files APP.OBJ and APP2.OBJ, creates an executable file called APP.EXE, creates a map file called APP.MAP, links no libraries.

Using response files

TLINK lets you supply the various parameters on the command line, in a response file, or in any combination of the two.

A response file is just a text file that contains the options and file names that you would usually type in after the name TLINK on your command line.

Unlike the command line, however, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line. When a plus occurs at the end of a line but it immediately follows one of the TLINK options that uses + to enable the option (such as */ye+*), the + is not treated as a line continuation character.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

with the following response file, FINRESP:


```
/c mainline wd+
  ln tx,fin
  mfin
  work\lib\comm work\lib\support
```

You would then enter your TLINK command as

```
tlink @finresp
```

Note that you must precede the file name with an "at" character (@) to indicate that the next name is a response file.

Alternately, you may break your link command into multiple response files. For example, you can break the previous command line into the following two response files:

File name	Contents
LISTOBS	mainline+ wd+
LISTLIBS	ln tx lib\comm+ lib\support

You would then enter the TLINK command as

```
tlink /c @listobjs,fin,mfin,@listlibs
```

The TLINK configuration file

The command line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded.

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked.

Using TLINK with Turbo C++ modules

Turbo C++ supports six different memory models: tiny, small, compact, medium, large, and huge. When you create an executable Turbo C++ file using TLINK, you must include the initialization module and libraries for the memory model being used.

The general format for linking Turbo C++ programs with TLINK is

```
tlink C0[F]x myobjs, exe,[map], [mylibs]
[OVERLAY] [EMU | FP87 mathx] Cx
```

where

- *myobjs* is the .OBJ files you want linked, specify path if not in current directory
- *exe* is the name to be given the executable file
- (optional) ■ *map* is the name to be given the map file
- (optional) ■ *mylibs* is the library files you want included at link time. You must specify the path if not in current directory, or use **/L** option to specify search paths

Be sure to include paths for the startup code and libraries (or use the **/L** option to specify a list of search paths for startup and library files). The other file names on this general TLINK command line represent Turbo C++ files, as follows:

If you are using the tiny model and you want TLINK to produce a .COM file, you must also specify the /t or /tdc option.

- C0x | C0Fx is the initialization module for DOS executable, DOS executable written for another compiler, with memory model **t, s, c, m, l, or h**.
- OVERLAY is the overlay manager library; needed only for overlaid programs.
- EMU | FP87 is the floating-point libraries (choose one).
- MATHx is the math library for memory model **s, c, m, l, or h**.
- Cx is the run-time library for memory model **s, c, m, l, or h**.

Startup code

The initialization modules have the name C0x.OBJ, where *x* is a single letter corresponding to the model: *t* for tint, *s* for small, *c* for compact, *m* for medium, *l* for large, and *h* for huge.

The C0Fx.OBJ modules are provided for compatibility with source files intended for compilers from other vendors. The C0Fx.OBJ modules substitute for the C0x.OBJ modules. These initialization modules alter the memory model so that the stack segment is inside the data segment. The appropriate C0Fx.OBJ module will be used automatically if you use either the **-Fs** or the **-Fm** command-line compiler option.

Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, that no stack has been created, or that fixup overflows occurred.

The initialization module must also appear as the first object file in the list. The initialization module arranges the order of the various segments of the program. If it is not first, the program segments may not be placed in memory properly, causing some frustrating program bugs.

Be sure that you give an explicit name for the executable file name on the TLINK command line. Otherwise, your program name will be something like C0x.EXE—probably not what you wanted!

Libraries



The order of objects and libraries is very important. You must always put the Turbo C++ start-up module (C0x.OBJ or C0Fx,) first in the list of objects. Then, the library list should contain, in this order:

- your own libraries (if any)
- if you want to overlay your program, you must include OVERLAY.LIB; this library must precede the Cx.LIB library
- Cx.LIB, run-time library
- if you are using floating point math, FP87.LIB or EMU.LIB, followed by MATHx.LIB

BGI graphics library

If you are using any Turbo C++ BGI graphics functions, you must link in GRAPHICS.LIB anywhere in the list. The BGI graphics library is independent of memory models.

Math libraries

If your program uses any floating-point, you must include the math library (MATHx.LIB) in the link command. You'll also need to include either the EMU.LIB or FP87.LIB floating-point libraries. Turbo C++'s two floating-point libraries are independent of the program's memory model.

- Use EMU.LIB if you want to include floating-point emulation logic. With EMU.LIB the program will work on machines whether they have a math coprocessor (80x87) chip or not.
- If you know that the program will always be run on a machine with a math coprocessor chip, the FP87.LIB library will produce a smaller and faster executable program.

The math libraries have the name MATHx.LIB, where *x* is a single letter corresponding to the model: s, c, m, l, h (the tiny and small models share the library MATHS.LIB).

You can always include the emulator and math libraries in a link command line. If you do so, and if your program does no floating-point work, nothing from those libraries will be added to your executable program file. However, if you know there is no floating-point work in your program, you can save some time in your links by excluding those libraries from the command line.

Run-time libraries

If you don't use all six memory models, you may want to keep only the files for the model(s) you use.

You must always include the C run-time library for the program's memory model. The C run-time libraries have the name Cx.LIB, where *x* is a single letter corresponding to the model, as before.

Here's a list of the library files needed for each memory model (you'll also need FP87.LIB or EMU.LIB :

Table 10.2
.OBJ and .LIB files

Model	Regular Startup Module	Compatibility Startup Module	Math Library	Run-time Library
Tiny	C0T.OBJ	C0FT.OBJ	MATHS.LIB	CS.LIB
Small	C0S.OBJ	C0FS.OBJ	MATHS.LIB	CS.LIB
Compact	C0C.OBJ	C0FC.OBJ	MATHC.LIB	CC.LIB
Medium	C0M.OBJ	C0FM.OBJ	MATHM.LIB	CM.LIB
Large	C0L.OBJ	C0FL.OBJ	MATHL.LIB	CL.LIB
Huge	C0H.OBJ	C0FH.OBJ	MATHH.LIB	CH.LIB

Note that the tiny and small models use the same libraries, but have different startup files (C0T.OBJ vs. C0S.OBJ).

Using TLINK with TCC

You can also use TCC, the standalone Turbo C++ compiler, as a "front end" to TLINK that will invoke TLINK with the correct startup file, libraries, and executable program name.

See Chapter 8, "The command-line compiler," for more on TCC.

To do this, you give file names on the TCC command line with explicit .OBJ and .LIB extensions. For example, given the following TCC command line,

```
TCC -mx MAINFILE.OBJ SUB1.OBJ MYLIB.LIB
```

TCC will invoke TLINK with the files C0x.OBJ, EMU.LIB, MATHx.LIB and Cx.LIB (initialization module, default 8087 emulation library, math library and run-time library for memory

model *x*). TLINK will link these along with your own modules MAINLINE.OBJ and SUB1.OBJ, and your own library MYLIB.LIB.

When TCC invokes TLINK, it uses the **/c** (case-sensitive link) option by default. You can override this default with **-l -c**.

TLINK options

TLINK options can occur anywhere on the command line. The options consist of a slash (/), a hyphen (-), or the DOS switch character, followed by the option.

If you have more than one option, spaces are not significant (**/m/c** is the same as **/m /c**), and you can have them appear in different places on the command line. The following sections describe each of the options.

The TLINK configuration file

The command-line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded.

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked. Whitespace is ignored.

/3 (32-bit code)

This option increases the memory requirements of TLINK and slows down linking, so it should be used only when necessary.

The **/3** option should be used when one or more of the object modules linked has been produced by TASM or a compatible assembler, and contains 32-bit code for the 80386 or the i486 processor.

/c (case sensitivity)

The **/c** option forces the case to be significant in public and external symbols.

/d (duplicate symbols)

Normally, TLINK will not warn you if a symbol appears in more than one library file. If the symbol must be included in the program, TLINK will use the copy of that symbol in the first file on the command line in which it is found. Since this is a commonly used feature, TLINK does not normally warn about the duplicate symbols. The following hypothetical situation illustrates how you might want to use this feature.

Suppose you have two libraries: one called SUPPORT.LIB, and a supplemental one called DEBUGSUP.LIB. Suppose also that DEBUGSUP.LIB contains duplicates of some of the routines in SUPPORT.LIB (but the duplicate routines in DEBUGSUP.LIB include slightly different functionality, such as debugging versions of the routines). If you include DEBUGSUP.LIB *first* in the link command, you will get the debugging routines and *not* the routines in SUPPORT.LIB.

If you are not using this feature or are not sure which routines are duplicated, you may include the /d option. TLINK will list all symbols duplicated in libraries, even if those symbols are not going to be used in the program.

Given this option, TLINK will also warn about symbols that appear both in an .OBJ and a .LIB file. In this case, since the symbol that appears in the first (left-most) file listed on the command line is the one linked in, the symbol in the .OBJ file is the one that will be used.

The exception to this rule is OVERLAY.LIB. OVERLAY.LIB does duplicate some symbols found in other libraries; that's why OVERLAY.LIB must be the first standard library specified on the TLINK command line.

With Turbo C++, the distributed libraries you would use in any given link command do not contain any duplicated symbols. So while EMU.LIB and FP87.LIB (or CS.LIB and CL.LIB) obviously have duplicate symbols, they would never rightfully be used together in a single link. There are no symbols duplicated between EMU.LIB, MATHS.LIB, and CS.LIB, for example.

/e (no extended dictionary)

The library files that are shipped with Turbo C++ all contain an *extended dictionary* with information that enables TLINK to link faster with those libraries. This extended dictionary can also be added to any other library file using the /E option with TLIB. The TLINK /e option disables the use of this dictionary.

Although linking with libraries that contain an extended dictionary is faster, you might want to use the **/e** option if you have a program that needs slightly more memory to link when an extended dictionary is used.



Unless you use **/e** to turn off extended dictionary use, TLINK will ignore any debugging information contained in a library that has an extended dictionary.

/i (uninitialized trailing segments)

The **/i** option causes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This option is not normally necessary.

/l (line numbers)

The **/l** option creates a section in the .MAP file for source code line numbers. To use it, you must have created the .OBJ files by compiling with the **-y** or **-v** option. If you use the **/x** to tell TLINK to create no map at all, this option will have no effect.

/L (library search paths)

The **/L** option lets you specify a list of directories that TLINK searches for libraries if an explicit path is not specified. TLINK searches the current directory before those specified with the **/L** option. For example,

```
TLINK /Lc:\TC\lib;c:\mylibs splash logo,,,utils .\logolib
```

With this command line, TLINK first searches the current directory for UTILS.LIB, then searches C:\TC\LIB and C:\MYLIBS. Because .\LOGOLIB explicitly names the current directory, TLINK does not search the libraries specified with the **/L** option to find LOGOLIB.LIB.

TLINK also searches for the C or C++ initialization module (C0x.OBJ or C0F.OBJ) on the specified library search path.

/m, /s, and /x (map options)

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error

messages produced during the link. If you don't want to create a map, turn it off with the `/x` option.

If you want to create a more complete map, the `/m` option will add a list of public symbols to the map file, sorted alphabetically as well as in increasing address order. This kind of map file is useful in debugging. Many debuggers can use the list of public symbols to allow you to refer to symbolic addresses when you are debugging.

The `/s` option creates a map file with segments, public symbols and the program start address just like the `/m` option did, but also adds a detailed segment map. Figure 10.1 is an example of a detailed segment map.

For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Except for the ACBP field, the information in the detailed segment map is self-explanatory.

Figure 10.1
Detailed map of segments

Address	Length (Bytes)	Class	Segment Name	Group	Module	Alignment/ Combining
0000:0000	0E5B	C=CODE	S=SYMB TEXT	G=(none)	M=SYMB.C	ACBP=28
00E5:000B	2735	C=CODE	S=QUAL TEXT	G=(none)	M=QUAL.C	ACBP=28
0359:0000	002B	C=CODE	S=SCOPY TEXT	G=(none)	M=SCOPY	ACBP=28
035B:000B	003A	C=CODE	S=LRSH TEXT	G=(none)	M=LRSH	ACBP=20
035F:0005	0083	C=CODE	S=PADA TEXT	G=(none)	M=PADA	ACBP=20
0367:0008	005B	C=CODE	S=PADD TEXT	G=(none)	M=PADD	ACBP=20
036D:0003	0025	C=CODE	S=PSBP TEXT	G=(none)	M=PSBP	ACBP=20
036F:0008	05CE	C=CODE	S=BRK TEXT	G=(none)	M=BRK	ACBP=28
03CC:0006	066F	C=CODE	S=FLOAT TEXT	G=(none)	M=FLOAT	ACBP=20
0433:0006	000B	C=DATA	S=_DATA	G=DGROUP	M=SYMB.C	ACBP=48
0433:0012	00D3	C=DATA	S=_DATA	G=DGROUP	M=QUAL.C	ACBP=48
0433:00E6	000E	C=DATA	S=_DATA	G=DGROUP	M=BRK	ACBP=48
0442:0004	0004	C=BSS	S=_BSS	G=DGROUP	M=SYMB.C	ACBP=48
0442:0008	0002	C=BSS	S=_BSS	G=DGROUP	M=QUAL.C	ACBP=48
0442:000A	000E	C=BSS	S=_BSS	G=DGROUP	M=BRK	ACBP=48

The ACBP field encodes the A (*alignment*), C (*combination*), and B (*big*) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (alignment)	00	An absolute segment.
	20	A byte-aligned segment.
	40	A word-aligned segment.
	60	A paragraph-aligned segment.
	80	A page-aligned segment.
	A0	An unnamed absolute portion of storage.
The C field (combination)	00	May not be combined.
	08	A public combining segment.
The B field (big)	00	Segment less than 64K.
	02	Segment exactly 64K.

When you request a detailed map with the **/s** option, the list of public symbols (if it appears) has public symbols flagged with "idle" if there are no references to that symbol. For example, this fragment from the public symbol section of a map file indicates that symbols *Symbol1* and *Symbol3* are not referenced by the image being linked:

```
0C7F:031E  idle  Symbol1
0000:3EA2                Symbol2
0C7F:0320  idle  Symbol3
```

/n (ignore default libraries)

The **/n** option causes the linker to ignore default libraries specified by some compilers. You may want to use this option when linking modules written in another language.

/o (overlays)

The **o** option causes the code in all modules or libraries specified after the option to be overlaid. It remains in effect until the next comma (explicit or implicit) or **o-** on the command line. **o-** turns off overlaying. (Chapter 18, "Memory management," covers overlays in more detail.)

The **o** option can be optionally followed by a segment class name; this will cause all segments of that class to be overlaid. When no such name is specified, all segments of classes ending with CODE will be overlaid. Multiple **o** options can be given, thus overlaying segments of several classes; all **o** options remain in effect until the next comma or **o-** is encountered.

The syntax **b#xx**, where *xx* is a two-digit hexadecimal number, overrides the overlay interrupt number, which by default is 3FH.

Here are some examples of **b** options:

Table 10.3
TLINK overlay options

Option	Result
b	Overlay all code segments until next comma or b- .
b-	Stop overlaying.
bOVY	Overlay segments of class OVY until the next comma or b- .
bCODE bOVLY	Overlay segments of class CODE or class OVLY until next comma or b- .
b#F0	Use interrupt vector 0F0H for overlays.

If you use the **b** option, it will be turned off automatically before the libraries are processed. If you want to overlay a library, you must use another **b** right before all the libraries or right before the library you want to overlay.

/t (tiny model .COM file)

When you use **/t**, the default extension for the executable file is .COM. This works the same as the **/Tdc** option.

Note: .COM files may not exceed 64K in size, cannot have any segment-relative fixups, cannot define a stack segment, and must have a starting address equal to 0:100H. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100H.

TLINK can't generate debugging information for a .COM file. If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **-c** option; this creates a .COM file from an .EXE.

/Td

These options are called target options. You use them (with **c**, **e**, or **d**) to produce a .COM or .EXE file.

- **/Td** creates an .EXE file.
- **/Tdc** creates a .COM file.
- **/Tde** creates an .EXE file.

/v (debugging information)

The **/v** option directs TLINK to include debugging information in the executable file. If this option is found anywhere on the command line, debugging information will be included executable for all object modules that contained debugging information. You can use the **/v+** and **/v-** options to selectively enable or disable inclusion of debugging information on a module-by-module basis (but not on the same command line as **/v**). For example, this command

```
tlink mod1 /v+ mod2 mod3 /v- mod4
```

includes debugging information for modules *mod2* and *mod3*, but not for *mod1* and *mod4*.

TLINK can't generate debugging information for a .COM file (one created with the **/t** or **/Tdc** options). If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **-c** option; this creates a .COM file from an .EXE.

/ye (expanded memory)

This option controls TLINK's use of expanded memory for I/O buffering. If, while reading object files or while writing the executable file, TLINK needs more memory for active data structures, it will either purge buffers or swap them to expanded memory.

In the case of input file buffering, purging simply means throwing away the input buffer so that its space can be used for other data structures. In the case of output file buffering, purging means writing the buffer to its correct place in the executable file. In either case, you can substantially increase the speed of a link by allowing these buffers to be swapped to expanded memory.

TLINK's capacity is not increased by swapping; only its performance is improved. By default, swapping to extended memory is enabled, while swapping to expanded memory is disabled. If swapping is enabled and no appropriate memory exists in which to swap, then swapping does not occur.

This option has several forms, shown below

/ye or **/ye+** enable expanded memory swapping (default)
/ye- disable expanded memory swapping

/yx (extended
memory)

The **/yx** option controls TLINK's use of extended memory for I/O buffering. By default, TLINK will take up to 8MB of extended memory. You can change TLINK's use of extended memory with one of the following forms of this option:

/yx+ Use all available extended memory.
/yx*n* Use only up to *n* KB extended memory.

P A R T

2

Programming reference

Lexical elements

This chapter provides a formal definition of the Turbo C++ lexical elements. It is concerned with the different categories of word-like units, known as *tokens*, recognized by a language. By contrast, language structure (covered in Chapter 12) details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

The tokens in Turbo C++ are derived from a series of operations performed on your programs by the compiler and its built-in pre-processor.

A Turbo C++ program starts life as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Turbo C++ editor). The basic program unit in Turbo C++ is the file. This usually corresponds to a named DOS file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see page 501). For example, the directive **#include** *<inc_file>* adds (or includes) the contents of the file *inc_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

Whitespace

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and *whitespace*. *Whitespace* is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i ;  
float f ;
```

are lexically equivalent and parse identically to give the six tokens:

1. **int**
2. **i**
3. **;**
4. **float**
5. **f**
6. **;**

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process; in other words, they remain as part of the string:

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International".

Line splicing with \

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \  
International"
```

is parsed as "Borland International" (see page 362, "String literals," for more information).

Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. Both are supported by Turbo C++, with an additional, optional extension permitting nested comments. You may use either kind of comment in both C and C++ programs.

C comments A C comment is any sequence of characters placed after the symbol pair */**. The comment terminates at the first occurrence of the pair **/* following the initial */**. The entire sequence, including the four comment delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

See page 507 for a description of token pasting.

Turbo C++ does not support the nonportable *token pasting* strategy using */**/*. Token pasting in Turbo C++ is performed with the ANSI-specified pair *##*, as follows:

```
#define VAR(i,j) (i/**/j)    /* won't work */
#define VAR(i,j) (i##j)    /* OK in Turbo C++ */
#define VAR(i,j) (i ## j)  /* Also OK */
```

In Turbo C++,

```
int /* declaration */ i /* counter */;
```

parses as

```
int i ;
```

to give the three tokens: **int i ;**

Nested comments ANSI C doesn't allow nested comments. Attempting to comment out the preceding line with

```
/* int /* declaration */ i /* counter */; */
```

fails, since the scope of the first */** ends at the first **/*. This gives

```
i ; /*
```

which would generate a syntax error.

By default, Turbo C++ won't allow nested comments, but you can override this with compiler options. You can enable nested comments via the Source Options dialog box (O|C|Source) in the IDE or with the `-C` option (for the command-line compiler).

C++ comments

You can also use // to create comments in C code. This is specific to Turbo C++.

C++ allows a single-line comment using two adjacent slashes (`//`). The comment can start in any position, and extends until the next new line:

```
class X { // this is a comment
... };
```

Comment delimiters and whitespace

In rare cases, some whitespace before `/*` and `//`, and after `*/`, although not syntactically mandatory, can avoid portability problems. For example, this C++ code

```
int i = j///* divide by k*/k;
+m;
```

parses as `int i = j +m;` not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

Tokens

Turbo C++ recognizes six classes of tokens. The formal definition of a token is as follows:

token:

keyword
identifier
constant
string-literal
operator
punctuator

Punctuators are also known as separators.

As the source code is parsed, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, **external** would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al*.

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names. The following two tables list the Turbo C++ keywords. You can use options in the IDE (or command-line compiler options) to select ANSI keywords only, UNIX keywords, and so on; see Chapter 2, "IDE Basics" and Chapter 8, "The command-line compiler," for information on these options.

Table 11.1
All Turbo C++ keywords

_asm	_ds	int	_seg
asm	else	_interrupt	short
auto	enum	interrupt	signed
break	_es	_loadds	sizeof
case	extern	long	_ss
_cdecl	_far	_near	static
cdecl	far	near	struct
char	_fastcall	new	switch
class	float	operator	template
const	for	_pascal	this
continue	friend	pascal	typedef
_cs	goto	private	union
default	_huge	protected	unsigned
delete	huge	public	virtual
do	if	register	void
double	inline	return	volatile
		_saveregs	while

Table 11.2
Turbo C++ extensions to C

_cdecl	_es	interrupt	pascal
cdecl	_far	_loadds	_saveregs
_cs	far	_near	_seg
_ds	_fastcall	near	_ss
	huge	_pascal	

Table 11.3
Keywords specific to C++

asm	operator
class	private
delete	protected
friend	public
inline	template
new	this
	virtual

Table 11.4
Turbo C++ register
pseudovariabiles

_AH	_BP	_CX	_DX
_AL	_BX	_DH	_ES
_AX	_CH	_DI	_FLAGS
_BH	_CL	_DL	_SI
_BL	_CS	_DS	_SP
			_SS

Identifiers

The formal definition of an identifier is as follows:

identifier:

nondigit

identifier nondigit

identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. Identifiers can contain the letters *A* to *Z* and *a* to *z*, the underscore character “_”, and the digits 0 to 9. There are only two restrictions:

1. The first character must be a letter or an underscore.
2. By default, Turbo C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu and command-line options, but not increased. Use the **-in** command-line option (where $1 \leq n \leq 32$) or Identifier Length in the Source Options dialog box (O|C|Source).

Identifiers in C++ programs are significant to 32 characters.

Identifiers and case sensitivity

Turbo C++ identifiers are case sensitive, so that *Sum*, *sum*, and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, Turbo C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. By checking Case-sensitive Link in the Linker dialog box

(Options | Linker | Settings), or using the `/c` command-line switch with `TLINK`, you can ensure that global identifiers are *case insensitive*. Under this regime, the globals `Sum` and `sum` are considered identical, resulting in a possible “Duplicate symbol” warning during linking.

An exception to these rules is that identifiers of type **pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are always legal for *different* name spaces regardless of scope. The rules are covered in the discussion on scope starting on page 371.

Constants

Constants are tokens representing fixed numeric or character values. Turbo C++ supports four classes of constants: floating point, integer, enumeration, and character.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in Table 11.5.

Integer constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Table 11.6. Note that the rules vary between decimal and nondecimal constants.

Decimal constants

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit will be truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10; /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0; /*decimal 0 = octal 0 */
```

Table 11.5: Constants—formal definitions

<i>constant:</i>	0 X hexadecimal-digit hexadecimal-constant hexadecimal-digit
<i>floating-constant</i>	
<i>integer-constant</i>	
<i>enumeration-constant</i>	nonzero-digit: one of 1 2 3 4 5 6 7 8 9
<i>character-constant</i>	
<i>floating-constant:</i>	octal-digit: one of 0 1 2 3 4 5 6 7
<i>fractional-constant</i> <exponent-part> <floating-suffix>	hexadecimal-digit: one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<i>digit-sequence</i> <i>exponent-part</i> <floating-suffix>	<i>integer-suffix:</i> unsigned-suffix <long-suffix> long-suffix <unsigned-suffix>
<i>fractional-constant:</i>	unsigned-suffix: one of u U
<digit-sequence> . <i>digit-sequence</i>	long-suffix: one of l L
<i>digit-sequence</i> .	enumeration-constant: identifier
<i>exponent-part:</i>	character-constant: c-char-sequence
e <sign> <i>digit-sequence</i>	c-char-sequence: c-char c-char-sequence c-char
E <sign> <i>digit-sequence</i>	c-char: Any character in the source character set except the single-quote ('), backslash (\), or newline character <i>escape-sequence</i> .
<i>sign:</i> one of + -	<i>escape-sequence:</i> one of \" \' \? \\ \a \b \f \n \o \oo \ooo \r \t \v \Xh... \xh...
<i>digit-sequence:</i> digit <i>digit-sequence</i> digit	
<i>floating-suffix:</i> one of f l F L	
<i>integer-constant:</i> decimal-constant <integer-suffix> octal-constant <integer-suffix> hexadecimal-constant <integer-suffix>	
<i>decimal-constant:</i> nonzero-digit decimal-constant digit	
<i>octal-constant:</i> 0 octal-constant octal-digit	
<i>hexadecimal-constant:</i> 0 x hexadecimal-digit	

Octal constants

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 will be truncated.

Hexadecimal constants

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF will be truncated.

long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces it to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul*, *lu*, *UL*, and so on.

Table 11.6
Turbo C++ integer constants
without L or U

Decimal constants	
0 to 32,767	int
32,768 to 2,147,483,647	long
2,147,483,648 to 4,294,967,295	unsigned long
> 4294967295	truncated
Octal constants	
00 to 077777	int
0100000 to 0177777	unsigned int
02000000 to 01777777777	long
020000000000 to 03777777777	unsigned long
> 03777777777	truncated
Hexadecimal constants	
0x0000 to 0x7FFF	int
0x8000 to 0xFFFF	unsigned int
0x10000 to 0x7FFFFFFF	long
0x80000000 to 0xFFFFFFFF	unsigned long
> 0xFFFFFFFF	truncated

The data type of a constant in the absence of any suffix (*U*, *u*, *L*, or *l*) is the first of the following types that can accommodate its value:

decimal	int, long int, unsigned long int
octal	int, unsigned int, long int, unsigned long int
hexadecimal	int, unsigned int, long int, unsigned long int

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int, unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int**, **unsigned long int** that can accommodate its value.

If the constant has both *u* and *l* suffixes (*ul*, *lu*, *Ul*, *lU*, *uL*, *Lu*, *LU*, or *UL*), its data type will be **unsigned long int**.

Table 11.6 summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Character constants

A *character constant* is one or more characters enclosed in single quotes, such as 'A', '=', '\n'. In C, single character constants have data type **int**; they are represented internally with 16 bits, with the upper byte zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

Escape sequences

The backslash character (\) is used to introduce an *escape sequence*, allowing the visual representation of certain nongraphic characters. For example, the constant **\n** is used for the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for *Ctrl-C* or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for Turbo C++). Larger numbers generate the compiler error, "Numeric constant too large." For example, the octal number \777 is larger than the maximum value allowed, \377, and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Originally, Turbo C allowed only three digits in a hexadecimal escape sequence. The ANSI C rules adopted in Turbo C++ might cause problems with old code that assumes only the first three characters are converted. For example, using Turbo C 1.x to define a string with a bell (ASCII 7) followed by numeric characters, a programmer might write:

```
printf("\x007.1A Simple Operating System");
```

This is intended to be interpreted as `\x007` and "2.1A Simple Operating System". However, Turbo C++ compiles it as the hexadecimal number `\x0072` and the literal string ".1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities may also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant `\258` would be interpreted as a two-character constant made up of the characters `\25` and `8`.

The next table shows the available escape sequences.

Table 11.7
Turbo C++ escape
sequences

The \\ must be used to represent a real ASCII backslash, as used in DOS paths.

Sequence	Value	Char	What it does
<code>\a</code>	0x07	BEL	Audible bell
<code>\b</code>	0x08	BS	Backspace
<code>\f</code>	0x0C	FF	Formfeed
<code>\n</code>	0x0A	LF	Newline (linefeed)
<code>\r</code>	0x0D	CR	Carriage return
<code>\t</code>	0x09	HT	Tab (horizontal)
<code>\v</code>	0x0B	VT	Vertical tab
<code>\\</code>	0x5c	<code>\</code>	Backslash
<code>\'</code>	0x27	<code>'</code>	Single quote (apostrophe)
<code>\"</code>	0x22	<code>"</code>	Double quote
<code>\?</code>	0x3F	<code>?</code>	Question mark
<code>\O</code>		any	O = a string of up to three octal digits
<code>\xH</code>		any	H = a string of hex digits
<code>\XH</code>		any	H = a string of hex digits

Turbo C++ special two-character constants

Turbo C++ also supports two-character constants (for example, `'An'`, `'\n\t'`, and `'\007\007'`). These constants are represented as 16-bit `int` values, with the first character in the low-order byte and the second character in the high-order byte. These constants are not portable to other C compilers.

signed and unsigned char

In C, one-character constants, such as `'A'`, `'\t'`, and `'\007'`, are also represented as 16-bit `int` values. In this case, the low-order byte is *sign extended* into the high byte; that is, if the value is greater than 127 (base 10), the upper byte is set to `-1` (`=0xFF`). This

can be disabled by declaring that the default **char** type is **unsigned** (use the **-K** command-line compiler option or choose Unsigned Characters in the Options | Compiler | Code Generation dialog box), which forces the high byte to be zero regardless of the value of the low byte.

Wide character constants

A character constant preceded by an *L* is a wide-character constant of data type **wchar_t** (an integral type defined in `stddef.h`). For example,

```
x = L 'A';
```

Floating-point constants

A floating constant consists of:

- decimal integer
- decimal point
- decimal fraction
- *e* or *E* and a signed integer exponent (optional)
- type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Examples:

Constant	Value
23.45e6	23.45×10^6
.0	0
0.	0
1.	$1.0 \times 10^0 = 1.0$
-1.23	-1.23
2e-5	2.0×10^{-5}
3E+10	3.0×10^{10}
.09E34	0.09×10^{34}

Floating-point constants — data types

In the absence of any suffixes, floating-point constants are of type **double**. However, you can coerce a floating constant to be of type

float by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The next table shows the ranges available for **float**, **double**, and **long double**.

Table 11.8
Turbo C++ floating constant
sizes and ranges

Type	Size (bits)	Range
float	32	3.4×10^{-38} to 3.4×10^{38}
double	64	1.7×10^{-308} to 1.7×10^{308}
long double	80	3.4×10^{-4932} to 1.1×10^{4932}

Enumeration constants

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed.

See page 417 for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, **cubs**, and **dodgers** are enumeration constants of type **team** that can be assigned to any variables of type **team** or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

String literals String literals, also known as string constants, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type **array of char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences (see page 358). This code, for example,

```
"\t\t\"Name\"\\\tAddress\n\n"
```

prints out like this:

```
      "Name" \      Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \ provides interior double quotes.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>

int main()
{
    char    *p;

    p = "This is an example of how Turbo C++"
        " will automatically\ndo the concatenation for"
        " you on very long strings,\nresulting in nicer"
        " looking programs.";
    printf(p);
    return(0);
}
```

The output of the program is

```
This is an example of how Turbo C++ will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character in order to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

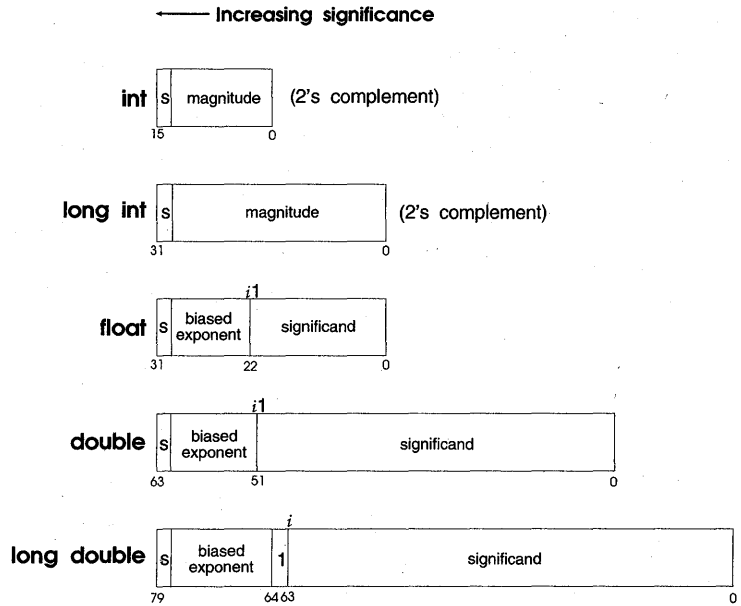
Constants and internal representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation specific and usually derive from the architecture of the host computer. For Turbo C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of inner representations for the various data types. The next table lists the sizes and resulting ranges of the data types for Turbo C++; see page 383 for more information on these data types. Figure 11.1 shows how these types are represented internally.

Table 11.9: Data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	16	-32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4×10^{-38} to 3.4×10^{38}	Scientific (7-digit precision)
double	64	1.7×10^{-308} to 1.7×10^{308}	Scientific (15-digit precision)
long double	80	3.4×10^{-4932} to 1.1×10^{4932}	Financial (19-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

Figure 11.1
Internal representations of
data types



s = Sign bit (0 = positive, 1 = negative)

i = Position of implicit binary point

1 = Integer bit of significand:

Stored in **long double**
Implicit (always 1) in **float**, **double**

Exponent bias (normalized values):

float : 127 (7FH)
double : 1023 (3FFH)
long double : 16,383 (3FFFH)

Constant expressions

A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is

constant-expression:

Conditional-expression

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- assignment
- comma
- decrement
- function call
- increment

Punctuators

The punctuators (also known as separators) in Turbo C++ are defined as follows:

punctuator: one of

[] () { } , ; : ... * = #

Brackets [] (open and close brackets) indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];      /* 3 x 4 matrix */
ch = str[3];        /* 4th element */
...
```

Parentheses () (open and close parentheses) group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);    /* override normal precedence */
if (d == z) ++x;    /* essential with conditional statement */

func();             /* function call, no args */
int (*fptr)();      /* function pointer declaration */
fptr = func;        /* no () means func pointer */

void func2(int n);  /* function declaration with args */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered on page 423.

Braces **{ }** (open and close braces) indicate the start and end of a compound statement:

```
if (d == z)
{
    ++x;
    func();
}
```

The closing brace serves as a terminator for the compound statement, so a **;** (semicolon) is not required after the **}**, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
    {}; /*illegal semicolon*/
else
```

Comma The comma (,) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j); /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5)); /* also calls func
                                         with two args! */
```

Semicolon The semicolon (;) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by ; is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, Turbo C++ may ignore it.

```
a + b; /* maybe evaluate a + b, but discard value */
++a; /* side effect on a, but discard value of ++a */
; /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement*:

```
for (i = 0; i < n; i++)
{
    ;
}
```

Colon Use the colon (:) to indicate a labeled statement:

```
start:   x=0;
...
goto start;
...
switch (a) {
    case 1: puts("One");
           break;
    case 2: puts("Two");
           break;
    ...
    default: puts("None of the above!");
           break;
}
```

Labels are covered on page 442.

Ellipsis Ellipsis (...) are three successive periods with no whitespace intervening. Ellipsis are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that **func** will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.



In C++, you can omit the comma preceding the ellipsis.

Asterisk (pointer declaration)

The * (asterisk) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr; /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;      /* a pointer to an integer array */
double ***double_ptr; /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer) The = (equal sign) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... } /* parameter i has default value of
                        zero */
```

The equal sign is also used as the assignment operator in expressions:

```
a = b + c;
ptr = farmalloc(sizeof(float)*100);
```

Pound sign
(preprocessor
directive)

The # (pound sign) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See page 501 for more on the preprocessor directives.

and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

Language structure

This chapter provides a formal definition of Turbo C++'s language structure. It details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units. By contrast, lexical elements (described in Chapter 11) are concerned with the different categories of word-like units, known as tokens, recognized by a language.

Declarations

Scope is discussed starting on page 371; visibility on page 373; duration on page 373; and linkage on page 375.

This section briefly reviews concepts related to declarations: objects, types, storage classes, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is not to be confused with the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely "points" to the object. The type is used

- to determine the correct memory allocation required initially
- to interpret the bit patterns found in the object during subsequent accesses
- in many type-checking situations, to ensure that illegal assignments are trapped

Turbo C++ supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in various memory models.

The Turbo C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Turbo C++ can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object, that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this rule, known as *forward references*, are labels, calls to undeclared functions, and class, struct, or union tags.

Lvalues

An *lvalue* is an object locator: An expression that designates an object. An example of an lvalue expression is **P*, where *P* is any expression evaluating to a nonnull pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the *l* stood for “left,” meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if *a* and *b* are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as $a = 1$; and $b = a + b$ are legal.

Rvalues The expression $a + b$ is not an lvalue: $a + b = a$ is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

Types and storage classes

Associating identifiers with objects requires that each identifier has at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Turbo C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type, as explained earlier, determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object’s storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type; see page 430 for more on this operator.

Scope

The *scope* of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

Block scope The *scope* of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.

Function scope The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label_name*: followed by a statement. Label names must be unique within a function.

Function prototype scope Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.

File scope File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.

Class scope (C++) For now, think of a class as a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see pages 455 to 468.

Scope and name spaces *Name space* is the scope within which an identifier must be unique. There are four distinct classes of identifiers in C:

1. **goto** label names. These must be unique within the function in which they are declared.
2. Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all tags defined externally.
3. Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.

Structures, classes, and enumerations are in the same name space in C++.

4. Variables, **typedefs**, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Visibility

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: The object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Visibility cannot exceed scope, but scope can exceed visibility.

```
...
{
    int i; char ch; // auto by default
    i = 3;         // int i and char ch in scope and visible
...
{
    double i;
    i = 3.0e3;    // double i in scope and visible
                // int i=3 in scope but hidden.
    ch = 'A';    // char ch in scope and visible
}
                // double i out of scope
i += 1;        // int i visible and = 4
...           // char ch still in scope & visible = 'A'
}
...           // int i and char ch out of scope
```



Again, special rules apply to hidden class names and class member names: Special C++ operators allow hidden identifiers to be accessed under certain conditions (see page 456).

Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedefs** and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

Static duration Objects with *static* duration are allocated memory as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Static duration must not be confused with file or global scope. An object can have static duration and local scope.

Local duration *Local* duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects always must have local or function scope. The storage class specifier **auto** may be used when declaring local duration variables, but is usually redundant, since **auto** is the default for variables declared within a block.

An object with local duration also has local scope, since it does not exist outside of its enclosing block. The converse is not true: A local scope object can have static duration.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. Turbo C++ can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Dynamic duration *Dynamic* duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the *heap*, using either standard library functions such as **malloc()**, or by using the C++ operator **new**. The corresponding deallocations are made using **free()** or **delete**.

Translation units

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration
    function-definition
    declaration
```

For more details, see
"External declarations and
definitions" on page 380.

The word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the following section, "Linkage.") Any declaration that also reserves storage for an object or function is called a definition (or defining declaration).

Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function only within one file. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules are as follows:

1. Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.

For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.

2. If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
3. If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
4. If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

The following identifiers have no linkage attribute:

- any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
- function parameters
- block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .OBJ files compiled with a C compiler.

To tell the C++ compiler not to mangle the name of a function, simply declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler that references to the function **Cfunc()** should not be mangled.

You can also apply the `extern "C"` declaration to a block of names:

```
extern "C" {
    void Cfunc1( int );
    void Cfunc2( int );
    void Cfunc3( int );
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions **Cfunc1()**, **Cfunc2()**, and **Cfunc3()** should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {
    #include "locallib.h"
};
```

Declaration syntax

All six interrelated attributes (storage class, type, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known simply as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration simply introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative definitions

The ANSI C standard introduces a new concept: that of the *tentative definition*. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative

definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;  
int x;          /*legal, one copy of x is reserved */  
  
int y;  
int y = 4;     /* legal, y is initialized to 4 */  
  
int z = 5;  
int z = 6;     /* not legal, both are initialized definitions */
```



Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible declarations

The range of objects that can be declared includes

- variables
- functions
- classes and class members (C++)
- types
- structure, union, and enumeration tags
- structure members
- union members
- arrays of other types
- enumeration constants
- statement labels
- preprocessor macros

The full syntax for declarations is shown in the following tables. The recursive nature of the declarator syntax allows complex declarators. We encourage the use of **typedefs** to improve legibility.

Table 12.1
Turbo C++ declaration syntax

<p><i>declaration:</i> <i><decl-specifiers> <declarator-list>;</i> <i>asm-declaration</i> <i>function-declaration</i> <i>linkage-specification</i></p> <p><i>decl-specifier:</i> <i>storage-class-specifier</i> <i>type-specifier</i> <i>fmt-specifier</i> friend (C++ specific) typedef</p> <p><i>decl-specifiers:</i> <i><decl-specifiers> decl-specifier</i></p> <p><i>storage-class-specifier:</i> auto register static extern</p> <p><i>fmt-specifier:</i> (C++ specific) inline virtual</p> <p><i>type-specifier:</i> <i>simple-type-name</i> <i>class-specifier</i> <i>enum-specifier</i> <i>elaborated-type-specifier</i> const volatile</p> <p><i>simple-type-name:</i> <i>class-name</i> typedef-name char short</p>	<p>int long signed unsigned float double void</p> <p><i>elaborated-type-specifier:</i> <i>class-key identifier</i> <i>class-key class-name</i> enum <i>enum-name</i></p> <p><i>class-key:</i> (C++ specific) class struct union</p> <p><i>enum-specifier:</i> enum <i><identifier> { <enum-list> }</i></p> <p><i>enum-list:</i> <i>enumerator</i> <i>enumerator-list , enumerator</i></p> <p><i>enumerator:</i> <i>identifier</i> <i>identifier = constant-expression</i></p> <p><i>constant-expression:</i> <i>conditional-expression</i></p> <p><i>linkage-specification:</i> (C++ specific) extern <i>string { <declaration-list> }</i> extern <i>string declaration</i></p> <p><i>declaration-list:</i> <i>declaration</i> <i>declaration-list ; declaration</i></p>
---	--

For the following table, note that there are restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail starting on page 391.

Table 12.2: Turbo C++ declarator syntax

<p><i>declarator-list:</i> <i>init-declarator</i> <i>declarator-list</i> , <i>init-declarator</i></p>	<p><i>class-name</i> (C++ specific) ~ <i>class-name</i> (C++ specific) typedef-name</p>
<p><i>init-declarator:</i> <i>declarator</i> <<i>initializer</i>></p>	<p><i>type-name:</i> <i>type-specifier</i> <<i>abstract-declarator</i>></p>
<p><i>declarator:</i> <i>dname</i> <i>modifier-list</i> <i>ptr-operator declarator</i> <i>declarator</i> (<i>parameter-declaration-list</i>) <<i>cv-qualifier-list</i>> (The <<i>cv-qualifier-list</i>> is for C++ only.) <i>declarator</i> [<<i>constant-expression</i>>] (<i>declarator</i>)</p>	<p><i>abstract-declarator:</i> <i>ptr-operator</i> <<i>abstract-declarator</i>> <<i>abstract-declarator</i>> (<i>argument-declaration-list</i>) <<i>cv-qualifier-list</i>> <<i>abstract-declarator</i>> [<<i>constant-expression</i>>] (<i>abstract-declarator</i>)</p>
<p><i>modifier-list:</i> <i>modifier</i> <i>modifier-list modifier</i></p>	<p><i>argument-declaration-list:</i> <<i>arg-declaration-list</i>> <i>arg-declaration-list</i> , ... <<i>arg-declaration-list</i>> ... (C++ specific)</p>
<p><i>modifier:</i> cdecl pascal interrupt near far huge</p>	<p><i>arg-declaration-list:</i> <i>argument-declaration</i> <i>arg-declaration-list</i> , <i>argument-declaration</i></p>
<p><i>ptr-operator:</i> * <<i>cv-qualifier-list</i>> & <<i>cv-qualifier-list</i>> (C++ specific) <i>class-name</i> :: * <<i>cv-qualifier-list</i>> (C++ specific)</p>	<p><i>argument-declaration:</i> <i>decl-specifiers declarator</i> <i>decl-specifiers declarator</i> = <i>expression</i> (C++ specific) <i>decl-specifiers</i> <<i>abstract-declarator</i>> <i>decl-specifiers</i> <<i>abstract-declarator</i>> = <i>expression</i> (C++ specific)</p>
<p><i>cv-qualifier-list:</i> <i>cv-qualifier</i> <<i>cv-qualifier-list</i>></p>	<p><i>fct-definition:</i> <<i>decl-specifiers</i>> <i>declarator</i> <<i>ctor-initializer</i>> <i>fct-body</i></p>
<p><i>cv-qualifier</i> const volatile</p>	<p><i>fct-body:</i> <i>compound-statement</i></p>
<p><i>dname:</i> <i>name</i></p>	<p><i>initializer:</i> = <i>expression</i> = { <i>initializer-list</i> } (<i>expression-list</i>) (C++ specific)</p>
	<p><i>initializer-list:</i> <i>expression</i> <i>initializer-list</i> , <i>expression</i> { <i>initializer-list</i> <,> }</p>

External declarations and definitions

The storage class specifiers **auto** and **register** cannot appear in an external declaration (see "Translation units," page 375). For each identifier in a translation unit declared with internal linkage, there can be no more than one external definition.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of **sizeof**), there must be exactly one external definition of that identifier somewhere in the entire program.

Turbo C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. For example,

```
int a[];           // no size
struct mystruct; // tag only, no member declarators
...
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
    int i, j;
};                // add member declarators
```

The following table covers class declaration syntax. Page 449 covers C++ reference types (closely related to pointer types) in detail.

Table 12.3: Turbo C++ class declarations (C++ only)

<i>class-specifier:</i> <i>class-head</i> { < <i>member-list</i> > }	<i>access-specifier</i> < virtual > <i>class-name</i>																																										
<i>class-head:</i> <i>class-key</i> < <i>identifier</i> > < <i>base-spec</i> > <i>class-key</i> <i>class-name</i> < <i>base-spec</i> >	<i>access-specifier:</i> private protected public																																										
<i>member-list:</i> <i>member-declaration</i> < <i>member-list</i> > <i>access-specifier</i> : < <i>member-list</i> >	<i>conversion-function-name:</i> operator <i>conversion-type-name</i>																																										
<i>member-declaration:</i> < <i>decl-specifiers</i> > < <i>member-declarator-list</i> > ; <i>function-definition</i> <;> <i>qualified-name</i> ;	<i>conversion-type-name:</i> <i>type-specifiers</i> < <i>ptr-operator</i> >																																										
<i>member-declarator-list:</i> <i>member-declarator</i> <i>member-declarator-list</i> , <i>member-declarator</i>	<i>ctor-initializer:</i> : <i>mem-initializer-list</i>																																										
<i>member-declarator:</i> <i>declarator</i> < <i>pure-specifier</i> > < <i>identifier</i> > : <i>constant-expression</i>	<i>mem-initializer-list:</i> <i>mem-initializer</i> <i>mem-initializer</i> , <i>mem-initializer-list</i>																																										
<i>pure-specifier:</i> = 0	<i>mem-initializer:</i> <i>class name</i> (< <i>argument-list</i> >) <i>identifier</i> (< <i>argument-list</i> >)																																										
<i>base-spec:</i> : <i>base-list</i>	<i>operator-function-name:</i> operator <i>operator</i>																																										
<i>base-list:</i> <i>base-specifier</i> <i>base-list</i> , <i>base-specifier</i>	<i>operator:</i> one of new delete sizeof																																										
<i>base-specifier:</i> <i>class-name</i> virtual < <i>access-specifier</i> > <i>class-name</i>	<table border="0"> <tbody> <tr> <td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr> <tr> <td>&</td><td> </td><td>~</td><td>!</td><td>=</td><td><></td></tr> <tr> <td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td></tr> <tr> <td>&=</td><td> =</td><td><<</td><td>>></td><td>>>=</td><td><<=</td></tr> <tr> <td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td><td> </td></tr> <tr> <td>++</td><td>--</td><td>,</td><td>->*</td><td>-></td><td>()</td></tr> <tr> <td>[]</td><td>.*</td><td></td><td></td><td></td><td></td></tr> </tbody> </table>	+	-	*	/	%	^	&		~	!	=	<>	+=	-=	*=	/=	%=	^=	&=	=	<<	>>	>>=	<<=	==	!=	<=	>=	&&		++	--	,	->*	->	()	[]	.*				
+	-	*	/	%	^																																						
&		~	!	=	<>																																						
+=	-=	*=	/=	%=	^=																																						
&=	=	<<	>>	>>=	<<=																																						
==	!=	<=	>=	&&																																							
++	--	,	->*	->	()																																						
[]	.*																																										

Type specifiers

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i; // declare i as a signed integer
unsigned char ch1, ch2; // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++ there are some situations where a missing type specifier leads to syntactic ambiguity, so C++ practice uses the explicit entry of all **int** type specifiers.

Type taxonomy

There are four basic type categories: *void*, *scalar*, *function*, and *aggregate*. The scalar and aggregate types can be further divided as follows:

- Scalar: arithmetic, enumeration, pointer, and reference types (C++)
- Aggregate: array, structure, union, and class types (C++)

Types can also be divided into *fundamental* and *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.



A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes.

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Table 12.4
Declaring types

type <i>t</i> ;	An object of type type
type <i>array</i> [10];	Ten types : <i>array</i> [0] – <i>array</i> [9]
type * <i>ptr</i> ;	<i>ptr</i> is a pointer to type
type & <i>ref</i> = <i>t</i> ;	<i>ref</i> is a reference to type (C++)
type func (void);	func returns value of type type
void func1 (type <i>t</i>);	func1 takes a type type parameter
struct <i>st</i> (type <i>t1</i> ; type <i>t2</i>);	structure <i>st</i> holds two types

Note that `type& var`, `type &var`, and `type & var` are all equivalent.

And here's how you could declare derived types in a class:

```
class ct {           // class ct holds pointer to type plus a
                    // function taking a "pointer to type" parameter.
    type *ptr;
public:
    void func(type*);
}
```

Type void

void is a special type specifier indicating the absence of any values. It is used in the following situations:

C++ handles **func()** in a special manner. See "Declarations and prototypes" on page 405 and code examples on page 406.

- An empty parameter list in a function declaration:

```
int func(void); // func takes no arguments
```

- When the declared function does not return a value:

```
void func(int n); // return value
```

- As a generic pointer: A pointer to **void** is a generic pointer to anything:

```
void *ptr; // ptr can later be set to point to any object
```

- In *typecasting* expressions:

```
extern int errfunc(); // returns an error code
...
(void) errfunc(); // discard return value
```

The fundamental types

long, **signed**, and **unsigned** are modifiers that can be applied to the integral types.

The fundamental type specifiers are built from the following keywords:

char	int	signed
double	long	unsigned
float	short	

From these keywords you can build the integral and floating-point types, which are together known as the *arithmetic* types. The include file `limits.h` contains definitions of the value ranges for all the fundamental types.

Integral types

char, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. The integral type specifiers are as follows, with synonyms listed on the same line:

Table 12.5
Integral types

char, signed char unsigned char	Synonyms if default char set to signed
char, unsigned char signed char	Synonyms if default char set to unsigned
int, signed int unsigned, unsigned int	
short, short int, signed short int unsigned short, unsigned short int	
long, long int, signed long int unsigned long, unsigned long int	

At most, one of **signed** and **unsigned** can be used with **char**, **short**, **int**, or **long**. If you use the keywords **signed** and **unsigned** on their own, they mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Turbo C++ lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares `ch` as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

At most, one of **long** and **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to insist that **short**, **int**, and **long** form a non-decreasing sequence with "**short** <= **int** <= **long**." All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In Turbo C++, the types **int** and **short** are equivalent, both being 16 bits. **long** is a 32-bit object. The signed varieties are all stored in 2's complement format using the most significant bit (MSB) as a

sign bit: 0 for positive, 1 for negative (which explains the ranges shown in Table 3.1 on page 363). In the unsigned versions, all bits are used to give a range of $0 - (2^n - 1)$, where n is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. Turbo C++ uses the IEEE floating-point formats. The online documentation, "ANSI.DOC," tells more about implementation-specific items.

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double test_case**, for example.

Table 3.1 on page 363 indicates the storage allocations for the floating-point types.

Standard conversions

When you use an arithmetic expression, such as $a + b$, where a and b are different arithmetic types, Turbo C++ performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps Turbo C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Table 12.6. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers, **double**, **float**, or **long double**).
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Table 12.6
Methods used in standard
arithmetic conversions

Type	Converts to	Method
char	int	Zero or sign-extended (depends on default char type)
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	Same value
unsigned short	unsigned int	Same value
enum	int	Same value

Special **char**, **int**, and **enum** conversions

The conversions discussed in this section are specific to Turbo C++.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged.

Converting a shorter integral type to a longer type either sign extends or zero fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

Initialization

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

If it has automatic storage duration, its value is indeterminate.

- to zero if it is of an arithmetic type
- to null if it is a pointer type

The syntax for initializers is as follows:

```
initializer
= expression
= {initializer-list} <,>
(expression list)
initializer-list
```

expression



```
initializer-list, expression
{initializer-list} <,>
```

Rules governing initializers are

1. The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
2. The item to be initialized must be an object type or an array of unknown size.
3. For C (not required for C++), all expressions must be constants if they appear in one of these places:
 - a. in an initializer for an object that has static duration
 - b. in an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed)
4. If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
5. If there are fewer initializers in a brace-enclosed list than there are members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- an initializer list as described in the following section
- a single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, intended to count how many times each day of the week appears in a month (and assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

Use these rules to initialize character arrays and wide character arrays:

1. You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for *name*[0]), 'n' (for *name*[1]), and so on (and including a null terminator).

2. You can initialize a wide character array (one that is compatible with **wchar_t**) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
    int i;
    char str[21];
    double d;
} s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces. You can eliminate the braces, but you must follow certain rules, and it isn't recommended practice.

Simple declarations

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where *var1*, *var2*,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.



In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions.

Storage class specifiers

A *storage class specifier*, or a type specifier, must be present in a declaration. The storage class specifiers can be one of the following:

auto	register	typedef
extern	static	

Use of storage class specifier **auto**

The storage class specifier **auto** is used only with local scope variable declarations. It conveys local (automatic) duration, but since this is the default for all local scope variable declarations, its use is rare.

Use of storage class specifier **extern**

The storage class specifier **extern** can be used with function and variable file scope and local scope declarations to indicate external linkage. With file scope variables, the default storage class specifier is **extern**. When used with variables, **extern** indicates that the variable has static duration. (Remember that functions always have static duration.) See page 376 for information on using **extern** to prevent name mangling when combining C and C++ code.

Use of storage class specifier **register**

The storage class specifier **register** is allowed only for local variable and function parameter declarations. It is equivalent to **auto**, but it makes a request to the compiler that the variable should be allocated to a register if possible. The allocation of a register can significantly reduce the size and improve the performance of programs in many situations. However, since Turbo C++ does a good job of placing variables in registers, it is rarely necessary to use the **register** keyword.

Turbo C++ lets you select register variable options from the Options | Compiler | Optimizations Options dialog box. If you check Automatic, Turbo C++ will try to allocate registers even if you have not used the **register** storage class specifiers.

Use of storage class specifier **static**

The storage class specifier **static** can be used with function and variable file scope and local scope declarations to indicate internal linkage. **static** also indicates that the variable has static duration. In the absence of constructors or explicit initializers, static variables are initialized with 0 or null.



In C++, a static data member of a class has the same value for all instances of a class. A static member function of a class can be invoked independently of any class instance.

Use of storage class specifier **typedef**

The keyword **typedef** indicates that you are defining a new data type specifier rather than declaring an object. **typedef** is included as a storage class specifier because of syntactical rather than functional similarities.

```
static long int biggy;
typedef long int BIGGY;
```

The first declaration creates a 32-bit, **long int**, static-duration object called *biggy*. The second declaration establishes the identifier *BIGGY* as a new type specifier, but does not create any run-time object. *BIGGY* can be used in any subsequent declaration where a type specifier would be legal. For example,

```
extern BIGGY salary;
```

has the same effect as

```
extern long int salary;
```

Although this simple example can be achieved by #define BIGGY long int, more complex **typedef** applications achieve more than is possible with textual substitutions.

Important!

typedef does not create new data types; it merely creates useful mnemonic synonyms or aliases for existing types. It is especially valuable in simplifying complex declarations:

```
typedef double (*PFD)();
PFD array_pfd[10];
/* array_pfd is an array of 10 pointers to functions
   returning double */
```

You can't use **typedef** identifiers with other data-type specifiers:

```
unsigned BIGGY pay;      /* ILLEGAL */
```

Modifiers

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier/object mapping. The modifiers available with Turbo C++ are summarized in Table 12.7.

The **const** modifier

The **const** modifier prevents any assignments to the object or any other side effects, such as increment or decrement. A **const** pointer cannot be modified, though the object to which it points can be. Consider the following examples:

The modifier **const** used by itself is equivalent to **const int**.

```
const float pi = 3.1415926;
const maxint = 32767;
char *const str = "Hello, world"; // A constant pointer
char const *str2 = "Hello, world"; /* A pointer to a constant
char */
```

Given these, the following statements are illegal:

```
pi = 3.0;          /* Assigns a value to a const */
i = maxint++;     /* Increments a const */
str = "Hi, there!"; /* Points str to something else */
```

Note, however, that the function call `strcpy(str, "Hi, there!")` is legal, since it does a character-by-character copy from the string literal "Hi, there!" into the memory locations pointed to by `str`.



In C++, **const** also hides the **const** object and prevents external linkage. You need to use **extern const**. A pointer to a **const** can't be assigned to a pointer to a non-**const** (otherwise, the **const** value could be assigned to using the non-**const** pointer). For example,

```
char *str3 = str2 /* disallowed */
```

Only **const** member functions can be called for a **const** object.

Table 12.7
Turbo C++ modifiers

C++ extends **const** and **volatile** to include classes and member functions.

Modifier	Use with	Use
const	Variables only	Prevents changes to object.
volatile	Variables only	Prevents register allocation and some optimization. Warns compiler that object may be subject to outside change during evaluation.

Table 12.7: Turbo C++ modifiers (continued)

Turbo C++ extensions		
cdecl	Functions	Forces C argument-passing convention. Affects Linker and link-time names.
cdecl	Variables	Forces global identifier case-sensitivity and leading underscores.
pascal	Functions	Forces Pascal argument-passing convention. Affects Linker and link-time names.
pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
interrupt	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt handlers.
near, far, huge	Pointer types	Overrides the default pointer type specified by the current memory model.
_cs, _ds, _es, _seg, _ss	Pointer types	Segment pointers. See page 582.
near, far, huge	Functions	Overrides the default function type specified by the current memory model.
near, far	Variables	Directs the placement of the object in memory.
_loadds	Functions	Sets DS to point to the current data segment.
_saveregs	Functions	Preserves all register values (except for return values) during execution of the function.
_fastcall	Functions	Forces register parameter passing convention. Affects the linker and link-time names.

The **interrupt** function modifier

The **interrupt** modifier is specific to Turbo C++. **interrupt** functions are designed to be used with the 8086/8088 interrupt vectors. Turbo C++ will compile an **interrupt** function with extra function entry and exit code so that registers AX, BX, CX, DX, SI, DI, ES, and DS are preserved. The other registers (BP, SP, SS, CS, and IP) are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function will use an **iret** instruction to return, so that the function can be used to service hardware or software interrupts. Here is an example of a typical **interrupt** definition:

```
void interrupt myhandler()
{
    ...
}
```

You should declare interrupt functions to be of type **void**. **interrupt** functions can be declared in any memory model. For all memory models except huge, DS is set to the program data segment. For the huge model, DS is set to the module's data segment.

The **volatile** modifier

*In C++, **volatile** has a special meaning for class member functions. If you've declared a **volatile** object, you can only use its **volatile** member functions.*

The **volatile** modifier indicates that the object may be modified; not only by you, but also by something outside of your program, such as an interrupt routine or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions containing it, since the value could (in theory) change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile int ticks;
interrupt timer()
{
    ticks++;
}

wait(int interval)
{
    ticks = 0;
    while (ticks < interval);    // Do nothing
}
```

These routines (assuming **timer()** has been properly associated with a hardware clock interrupt) implement a timed wait of ticks

specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop, since the loop doesn't change the value of *ticks*.

The **cdecl** and **pascal** modifiers

*Page 375 tells how to use **extern**, which allows C names to be referenced from a C++ program.*

Turbo C++ allows your programs to easily call routines written in other languages, and vice versa. When you mix languages like this, you have to deal with two important issues: identifiers and parameter passing.

In Turbo C++, all global identifiers are saved in their original case (lower, upper, or mixed) with an underscore “_” prepended to the front of the identifier, unless you have selected the **-u** option or unchecked the Generate Underbars box in the Options | Compiler | Advanced Code Generation dialog box.

pascal

In Pascal, global identifiers are not saved in their original case, nor are underscores prepended to them. Turbo C++ lets you declare any identifier to be of type **pascal**; the identifier is converted to uppercase, and no underscore is prepended. (If the identifier is a function, this also affects the parameter-passing sequence used; see “Function type modifiers,” page 396, for more details.)

*The **-p** compiler option or Calling Convention Pascal in the Options | Compiler | Entry | Exit Code dialog box causes all functions (and pointers to those functions) to be treated as if they were of type **pascal**.*

The **pascal** modifier is specific to Turbo C++; it is intended for functions (and pointers to functions) that use the Pascal parameter-passing sequence. Also, functions declared to be of type **pascal** can still be called from C routines, so long as the C routine sees that the function is of type **pascal**.

```
pascal putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

Functions of type **pascal** cannot take a variable number of arguments, unlike functions such as **printf()**. For this reason, you cannot use an ellipsis (...) in a **pascal** function definition.

cdecl

Once you have compiled with Pascal calling convention turned on (using the **-p** option or IDE Options | Compiler | Entry/Exit Code), you may want to ensure that certain identifiers have their

case preserved and keep the underscore on the front, especially if they're C identifiers from another file. You can do so by declaring those identifiers to be **cdecl**. (This also has an effect on parameter passing for functions).

main() must be declared as **cdecl**; this is because the C start-up code always tries to call **main()** with the C calling convention.

Like **pascal**, the **cdecl** modifier is specific to Turbo C++. It is used with functions and pointers to functions. It overrides the **-p** option or IDE Options | Compiler | Entry/Exit Code compiler directive and allows a function to be called as a regular C function. For example, if you were to compile the previous program with the Pascal calling option set but wanted to use **printf()**, you might do something like this:

```
extern cdecl printf();
void putnums(int i, int j, int k);

cdecl main()
{
    putnums(1,4,9);
}

void putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

If you compile a program with the **-p** option or IDE Options | Compiler | Entry/Exit Code, all functions used from the run-time library will need to have **cdecl** declarations. If you look at the header files (such as **stdio.h**), you'll see that every function is explicitly defined as **cdecl** in anticipation of this.

The pointer modifiers

Turbo C++ has eight modifiers that affect the pointer declarator (*); that is, they modify pointers to data. These are **near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_seg**, and **_ss**.

C lets you compile using one of several memory models. The model you use determines (among other things) the internal format of pointers. For example, if you use a small data model (tiny, small, medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact, large, huge), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes, when using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the pointer modifiers.

See the discussion starting on page 576 in Chapter 18 for an in-depth explanation of **near**, **far**, and **huge** pointers, and page 577 for a description of normalized pointers. Also see the discussion starting on page 582 for more on **_cs**, **_ds**, **_es**, **_seg**, and **_ss**.

Function type modifiers. The **near**, **far**, and **huge** modifiers can also be used as function type modifiers; that is, they can modify functions and function pointers as well as data pointers. In addition, you can use the **_loadds**, and **_saveregs** modifiers to modify functions.

The **near**, **far**, and **huge** function modifiers can be combined with **cdecl** or **pascal**, but not with **interrupt**.

Functions of type **huge** are useful when interfacing with code in assembly language that doesn't use the same memory allocation as Turbo C++.

A non-**interrupt** function can be declared to be **near**, **far**, or **huge** in order to override the default settings for the current memory model.

A **near** function uses **near** calls; a **far** or **huge** function uses **far** call instructions.

In the tiny, small, and compact memory models, an unqualified function defaults to type **near**. In the medium and large models, an unqualified function defaults to type **far**. In the huge memory model, it defaults to type **huge**.

A **huge** function is the same as a **far** function, except that the DS register is set to the data segment address of the source module when a **huge** function is entered, but left unset for a **far** function.

The **_loadds** modifier indicates that a function should set the DS register, just as a huge function does, but does not imply **near** or **far** calls. Thus, **_loadds far** is equivalent to **huge**.

The **_saveregs** modifier causes the function to preserve all register values and restore them before returning (except for explicit return values passed in registers such as AX or DX).

The **_loadds** and **_saveregs** modifiers are useful for writing low-level interface routines, such as mouse support routines.

Complex declarations and declarators

See Table 12.0 on page 379 for the declarator syntax. The definition covers both identifier and function declarators.

Simple declarations have a list of comma-delimited identifiers following the optional storage class specifiers, type specifiers, and other modifiers.

A complex declaration uses a comma-delimited list of declarators following the various specifiers and modifiers. Within each declarator, there exists just one identifier, namely the identifier being declared. Each of the declarators in the list is associated with the leading storage class and type specifier.

The format of the declarator indicates how the declared *dname* is to be interpreted when used in an expression. If **type** is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

storage-class-specifier type D1, D2;

indicates that each occurrence of *D1* or *D2* in an expression will be treated as an object of type **type** and storage class *storage class specifier*. The type of the *dname* embedded in the declarator will be some phrase containing **type**, such as "**type**," "pointer to **type**," "array of **type**," "function returning **type**," or "pointer to function returning **type**," and so on.

For example, in the declarations

```
int n, nao[], naf[3], *pn, *apn[], (*pan)[], &nr=n;
int f(void), *fnp(void), (*pfn)(void);
```

each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Table 12.8: Complex declarations

Declarator syntax	Implied <i>type</i> of <i>name</i>	Example
type <i>name</i> ;	type	int count;
type <i>name</i> [];	(open) array of type	int count[];
type <i>name</i> [3];	Fixed array of three elements, all of type (<i>name</i> [0], <i>name</i> [1], and <i>name</i> [2])	int count[3];
type * <i>name</i> ;	Pointer to type	int *count;
type * <i>name</i> [];	(open) array of pointers to type	int *count[];
type *(<i>name</i> [])	Same as above	int *(count[]);
type (* <i>name</i>)[];	Pointer to an (open) array of type	int (*count) [];
type & <i>name</i> ;	Reference to type (C++ only)	int &count;
type <i>name</i> ()	Function returning type	int count();
type * <i>name</i> ()	Function returning pointer to type	int *count();
type *(<i>name</i> ())	Same as above	int *(count());
type (* <i>name</i>)()	Pointer to function returning type	int (*count) ();

Note the need for parentheses in *(*name)[]* and *(*name)()*, since the precedence of both the array declarator `[]` and the function declarator `()` is higher than the pointer declarator `*`. The parentheses in **name[]* are optional.

Pointers

See page 429 for a discussion of referencing and dereferencing.

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain Turbo C++ operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Pointers to objects

A pointer of type “pointer to object of **type**” holds the address of (that is, points to) an object of **type**. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

The size of pointers to objects is dependent on the memory model and the size and disposition of your data segments, possibly influenced by the optional pointer modifiers (discussed starting on page 395).

Pointers to functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function’s executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called “pointer to function returning **type**,” where **type** is the function’s return type.



Under C++, which has stronger type checking, a pointer to a function has type “pointer to function taking argument types **type** and returning **type**.” In fact, under C, a function defined with argument types will also have this narrower type. For example,

```
void (*func)();
```

In C, this is a pointer to a function returning nothing. In C++, it’s a pointer to a function taking no arguments and returning nothing. In this example,

```
void (*func)(int);
```

func* is a pointer to a function taking an **int argument and returning nothing.

Pointer

declarations

See page 383 for details on **void**.

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. Turbo C++ lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to **void**. And in C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer.

If **type** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr; /* Danger--uninitialized pointer */
```

declares *ptr* to be of type “pointer to **type**.” All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic **NULL** (defined in the standard library header files, such as `stdio.h`) can be used for legibility. All pointers can be successfully tested for equality or inequality to **NULL**.

The pointer type “pointer to void” must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any “pointer to **type**” value, including null, without complaint. Assignments without proper casting between a “pointer to **type1**” and a “pointer to **type2**,” where **type1** and **type2** are different types, can invoke a compiler warning or error. If **type1** is a function and **type2** isn’t (or vice versa), pointer assignments are illegal. If **type1** is a pointer to **void**, no cast is needed. Under C, if **type2** is a pointer to **void**, no cast is needed.

Assignment restrictions also apply to pointers of different sizes (**near**, **far**, and **huge**). You can assign a smaller pointer to a larger one without error, but you can’t assign a larger pointer to a smaller one unless you are using an explicit cast. For example,

```

char near *ncp;
char far *fcp;
char huge *hcp;
fcp = ncp;           // legal
hcp = fcp;          // legal
fcp = hcp;          // not legal
ncp = fcp;          // not legal
ncp = (char near*)fcp; // now legal

```

Pointers and constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be assigned to. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```

int i;                // i is an int
int * pi;             // pi is a pointer to int
(uninitialized)
int * const cp = &i;  // cp is a constant pointer to int.
const int ci = 7;     // ci is a constant int
const int * pci;      // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
// constant int

```

The following assignments are legal:

```

i = ci;               // Assign const-int to int
*cp = ci;             // Assign const-int to
// object-pointed-at-by-a-const-pointer
++pci;               // Increment a pointer-to-const
pci = cpc;           // Assign a const-pointer-to-a-const to a
// pointer-to-const

```

The following assignments are illegal:

```

ci = 0;              // NO--cannot assign to a const-int
ci--;               // NO--cannot change a const-int
*pci = 3;           // NO--cannot assign to an object
// pointed at by pointer-to-const
cp = &ci;           // NO--cannot assign to a const-pointer,
// even if value would be unchanged
cpc++;             // NO--cannot change const-pointer

```

```

pi = pci; // NO--if this assignment were allowed,
          // you would be able to assign to *pci
          // (a const value) by assigning to *pi.

```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

Pointer arithmetic

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

The difference between two pointers only has meaning if both pointers point into the same array.

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type “pointer to **type**” automatically take into account the size of **type**; that is, the number of bytes needed to store a **type** object.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to **type**, adding an integral value to the pointer advances the pointer by that number of objects of **type**. If **type** has size 10 bytes, then adding an integer 5 to a pointer to **type** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of *ptr2* - *ptr1* would be 7.

When an integral value is added to or subtracted from a “pointer to **type**,” the result is also of type “pointer to **type**.”

There is no such element as “one past the last element”, of course, but a pointer is allowed to assume such a value. If *P* points to the last array element, *P* + 1 is legal, but *P* + 2 is undefined. If *P* points to one past the last array element, *P* - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a “pointer to one past the last element” leads to undefined behavior.

Informally, you can think of *P* + *n* as advancing the pointer by (*n* * **sizeof(type)**) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff_t* defined in `stddef.h` (**signed long** for huge and far pointers; **signed int** for all others). This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff_t*. In the expression *P1* - *P2*, where *P1* and *P2* are of type pointer to **type** (or pointer to qualified **type**), *P1* and *P2* must point to existing elements or to one past the last element. If *P1* points to the *i*-th

element, and $P2$ points to the j -th element, $P1 - P2$ has the value $(i - j)$.

Pointer conversions

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (**type***) will convert a pointer to type "pointer to **type**."

C++ reference declarations

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See page 449, "Referencing," for complete details.

Arrays

The declaration

```
type declarator [<constant-expression>]
```

declares an array composed of elements of **type**. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. Thus, a two-dimensional array of five rows and seven columns called *alpha* is declared as

```
type alpha [5] [7];
```

In certain contexts, the first array declarator of a series may have no expression inside the brackets. Such an array is of indeter-

minate size. The contexts where this is legitimate are ones in which the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array, nor does an array function parameter. As a special extension to ANSI C, Turbo C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

Functions

Functions are central to C and C++ programming. Languages such as Pascal distinguish between procedure and function. Turbo C++ functions play both roles.

Declarations and definitions

Each program must have a single external function named **main** marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see page 375).

Functions are defined in your source files or made available by linking precompiled libraries.

In C++ you must always use function prototypes. We recommend that you also use them in C.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide Turbo C++ with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a

definition and a declaration is that the definition has a function body.)

Declarations and prototypes

In C++, this declaration means `<type> func(void)`

You can enable a warning within the IDE or with the command-line compiler: "Function called without a prototype."

In the original Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows:

```
<type> func()
```

where **type** is the optional return type defaulting to **int**. A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

```
<type> func(parameter-declarator-list);
```

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
    int limit = 32;
    char ch = 'A';
    long mval;
    mval = lmax(limit, ch); /* function call */
}
```

Since it has the function prototype for **lmax()**, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to **lmax()**. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to **lmax()** would not match in size or content what **lmax()** was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function **strcpy()** takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is only used for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```



In C++, **func()** also declares a function taking no arguments.

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as **printf()**), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Here are some more examples of function declarators and prototypes:

```
int f();           /* In C, a function returning an int with no
                  information about parameters. This is the K&R
                  "classic style." */

int f();           /* In C++, a function taking no arguments */

int f(void);       /* A function returning an int that takes no
                  parameters. */

int p(int, long);  /* A function returning an int that accepts two
                  parameters: the first, an int; the second, a
                  long. */

int pascal q(void); /* A pascal function returning an int that takes
                  no parameters at all. */

char far *s(char *source, int kind); /* A function returning a far
                  pointer to a char and accepting two parameters: the
                  first, a pointer to a char; the second, an int. */
```

stdarg.h contains macros that you can use in user-defined functions with variable numbers of parameters.

```

int printf(char *format,...); /* A function returning an int and
                               accepting a pointer to a char fixed parameter and
                               any number of additional parameters of unknown
                               type. */

int (*fp)(int); /* A pointer to a function returning an int and
                 accepting a single int parameter. */

```

Definitions

The general syntax for external function definitions is given in the following table:

Table 12.9
External function definitions

```

file
  external-definition
  file external-definition

external-definition:
  function-definition
  declaration
  asm-statement

function-definition:
  <declaration-specifiers> declarator <declaration-list>
  compound-statement

```

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):


You can mix elements from 1 and 2.

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **pascal**, **cdecl**, **interrupt**, **near**, **far**, **huge**, **_loadds**, **_saveregs**. The defaults depend on the memory model and compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is **func(void)**. The old style of **func()** is legal in C but antiquated and possibly unsafe.
6. A function body representing the code to be executed when the function is called.

Formal parameter declarations

The formal parameter declaration list follows a similar syntax to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) { // no args
int func(T1 t1, T2 t2, T3 t3=1) { // three simple parameters, one
// with default argument
int func(T1* ptr1, T2& tref) { // a pointer and a reference arg
int func(register int i) { // request register for arg
int func(char *str,...) { /* one string arg with a variable
number of other args, or with a fixed number of args with
varying types */
```

 In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all enjoy automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal argument declarators.

Function calls and argument conversions

A function is called with actual arguments placed in the same sequence as their matching formal arguments. The actual arguments are converted as if by initialization to the declared types of the formal arguments.

Here is a summary of the rules governing how Turbo C++ deals with language modifiers and formal parameters in function calls, both with and without prototypes:

1. The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
2. A function may modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, Turbo C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in the section "Standard conversions," starting on page 385. When a function prototype is in scope, Turbo C++ converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Turbo C++ converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need only be compatible to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Important! If your function prototype does not match the actual function definition, Turbo C++ will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

Structure initialization is discussed on page 386.

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure

member can be a bit field type not allowed elsewhere. The Turbo C++ structure type lets you handle complex data structures almost as easily as single variables.



In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example,

```
struct mystruct { ... }; // mystruct is the structure tag
...
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged structures and typedefs

Untagged structure and union members are ignored during initialization.

If you omit the structure tag, you can get an *untagged* structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere:

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10]; // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, array[20];
```

You don't usually need both a tag and a **typedef**: either can be used in structure declarations.

Structure member declarations

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in Table 12.2 on page 380.

A structure member can be of any type, with two exceptions:

1. The member type cannot be the same as the **struct** type being currently declared:

You can omit the **struct** keyword in C++.

```
struct mystruct { mystruct s } s1, s2; // illegal
```

A member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct { mystruct *ps } s1, s2; // OK
```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

2. Except in C++, a member cannot have the type “function returning...,” but the type “pointer to function returning...” is allowed. In C++, a **struct** can have member functions.

Structures and functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void); // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s); // directly
void func2(mystruct *sptr); // via a pointer
void func3(mystruct &sref); // as a reference (C++ only)
```

Structure member access

Structure and union members are accessed using the selection operators `.` and `->`. Suppose that the object `s` is of struct type `S`, and `sptr` is a pointer to `S`. Then if `m` is a member identifier of type `M` declared in `S`, the expressions `s.m` and `sptr->m` are of type `M`, and both represent the member object `m` in `s`. The expression `sptr->m` is a convenient synonym for `(*sptr).m`.

The operator `.` is called the direct member selector; the operator `->` is called the indirect (or pointer) member selector; for example,

```
struct mystruct
{
    int i;
    char str[21];
    double d;
```

```

} s, *sptr=&s;
...
s.i = 3;           // assign to the i member of mystruct s
sptr->d = 1.23;    // assign to the d member of mystruct s

```

The expression *s.m* is an lvalue, provided that *s* is not an lvalue and *m* is not an array type. The expression *sptr->m* is an lvalue unless *m* is an array type.

If structure *B* contains a field whose type is structure *A*, the members of *A* can be accessed by two applications of the member selectors:

```

struct A {
    int j;
    double x;
};

struct B {
    int i;
    struct A a;
    double d;
} s, *sptr;

...
s.i = 3;           // assign to the i member of B
s.a.j = 2;        // assign to the j member of A
sptr->d = 1.23;    // assign to the d member of B
(sptr->a).x = 3.14 // assign to x member of A

```

Each structure declaration introduces a unique structure type, so that in

```

struct A {
    int i,j;
    double d;
} a, a1;

struct B {
    int i,j;
    double d;
} b;

```

the objects *a* and *a1* are both of type struct *A*, but the objects *a* and *b* are of different structure types. Structures can be assigned only if the source and destination have the same type:

```

a = a1; // OK: same type, so member by member assignment
a = b;  // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign
                                member-by-member */

```

Structure word alignment

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```
struct mystruct {  
    int i;  
    char str[21];  
  
    double d;  
} s;
```

the object *s* occupies sufficient memory to hold a 2-byte integer, a 21-byte string, and an 8-byte double. The format of this object in memory is determined by the Turbo C++ word alignment option. With this option off (the default), *s* will be allocated 31 contiguous bytes.

If you turn on word alignment with the Options | Compiler | Code Generation dialog box or with the `-a` compiler option, Turbo C++ pads the structure with bytes to ensure the structure is aligned as follows:

1. The structure will start on a word boundary (even address).
2. Any non-**char** member will have an even byte offset from the start of the structure.
3. A final byte is added (if necessary) at the end to ensure that the whole structure contains an even number of bytes.

With word alignment on, the structure would therefore have a byte added before the **double**, making a 32-byte object.

Structure name spaces

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example,

```
goto s;
...
s:
struct s { // OK: tag and label name spaces different
    int s; // OK: label, tag and member name spaces different
    float s; // ILLEGAL: member name duplicated
} s; // OK: var name space different. In C++, this can only
// be done if s does not have a constructor.

union s { // ILLEGAL: tag space duplicate
    int s; // OK: new member space
    float f;
} f; // OK: var name space

struct t {
    int s; // OK: different member space
    ...
} s; // ILLEGAL: var name duplicate
```

Incomplete declarations

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before *A* has been declared:

```
struct A; // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of *A* is called *incomplete* because there is no definition for it at that point. An incomplete declaration is allowed here, since the definition of *B* doesn't need the size of *A*.

Bit fields

A structure can contain any mixture of bit field and non-bit field types.

You can declare **signed** or **unsigned** integer members as bit fields from 1 to 16 bits wide. You specify the bit field width and optional identifier as follows:

```
type-specifier <bitfield-id> : width;
```

where *type-specifier* is **char**, **unsigned char**, **int**, or **unsigned int**. Bit fields are allocated from low-order to high-order bits within a word. The expression *width* must be present and must evaluate to a constant integer in the range 1 to 16.

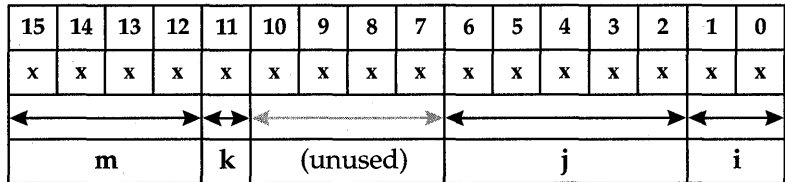
If the bit field identifier is omitted, the number of bits specified in *width* is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused. For example,

```

struct mystruct {
    int      i : 2;
    unsigned j : 5;
    int      : 4;
    int      k : 1;
    unsigned m : 4;
} a, b, c;

```

produces the following layout:



Integer fields are stored in 2's-complement form, with the leftmost bit being the MSB (most significant bit). With **int** (for example, **signed**) bit fields, the MSB is interpreted as a sign bit. A bit field of width 2 holding binary 11, therefore, would be interpreted as 3 if **unsigned**, but as -1 if **int**. In the previous example, the legal assignment `a.i = 6` would leave binary `10 = -2` in `a.i` with no warning. The signed **int** field `k` of width 1 can hold only the values -1 and 0, since the bit pattern 1 is interpreted as -1.



Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (`.` and `->`) used for non-bit field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent.

The expression `&mystruct.x` is illegal if `x` is a bit field identifier, since there is no guarantee that `mystruct.x` lies at a byte address.

Unions

Unions correspond to the variant record types of Pascal and Modula-2.

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The

value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
    int i;
    double d;
    char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union myunion**, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time.

sizeof(union myunion) and **sizeof(mu)** both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (**.** and **->**), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); // OK: displays mu.d = 4.016
printf("mu.i = %d\n",mu.i); // peculiar result
mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); // OK: displays mu.ch = A
printf("mu.d = %f\n",mu.d); // peculiar result
muptr->i = 3;
printf("mu.i = %d\n",mu.i); // OK: displays mu.i = 3
```

The second **printf()** is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous unions (C++ only)



```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the **x.y** or **p->y** syntax.

Anonymous unions can't have member functions and at file level must be declared static. In other words, an anonymous union may not have external linkage.

Union declarations

The general declaration syntax for unions is pretty much the same as that for structures. Differences are

1. Unions can contain bit fields, but only one can be active. They all start at the beginning of the union (and remember that, because bit fields are machine dependent, they pose several problems when writing portable code).



2. Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.

3. Unions can be initialized only through their first declared member:

```
union local87 {
    int i;
    double d;
} a = { 20 };
```



4. A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.

Enumerations

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum** *days*, a variable *anyday* of this type, and a set of enumerators (*sun, mon,...*) with constant integer values.

Turbo C++ is free to store enumerators in a single byte when `Treat enums as ints` is unchecked (O|C|Code Generation) or the `-b` flag. The default is on (meaning **enums** are always **ints**) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, that is the type of each enumerator.



In C, a variable of an enumerated type can be assigned any value of type **int**—no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;      // OK
anyday = 1;       // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```



In C++, you can omit the **enum** keyword if *days* is not the name of anything else in the same scope.

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

See page 361 for more on enumeration constants.

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on).

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* initializer expression can include previously declared
   enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
            quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
```

```

typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;           // OK
*daysptr = anyday;    // OK
mon = tues;            // ILLEGAL: mon is a constant

```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```

int mon = 11;
{
    enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
    /* enumerator mon hides outer declaration of int mon */
    struct days { int i, j;}; // ILLEGAL: days duplicate tag
    double sat;             // ILLEGAL: redefinition of sat
}
mon = 12;                  // back in int mon scope

```



In C++, enumerators declared within a class are in the scope of that class.

Expressions

Table 12.11 shows how identifiers and operators are combined to form grammatically legal "phrases."

The standard conversions are detailed in Table 12.6 on page 386.

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Table 12.11, indicates that expressions are defined recursively: Subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules which depend on the operators used, the presence of parentheses, and the data types of the operands. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by Turbo C++ (see "Evaluation order" on page 422).

Expressions can produce an lvalue, an rvalue, or no value. Expressions may cause side effects whether they produce a value or not.

We've summarized the precedence and associativity of the operators in Table 12.10. The grammar in Table 12.11 on page 421

completely defines the precedence and associativity of the operators.

There are sixteen precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where there are duplicates of operators in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator category in the following table is indicated by its order in the table. The first category (the first line) has the highest precedence.

Table 12.10
Associativity and
precedence of Turbo C++
operators

Operators	Associativity
() [] -> :: .	Left to right
! ~ + - ++ -- & * (typecast) sizeof new delete	Right to left
.* ->*	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?: (conditional expression)	Right to left
= *= /= %= += -= &= ^= = <<= >>=	Right to left
,	Left to right

Table 12.11: Turbo C++ expressions

<p>primary-expression: <i>literal</i> this (C++ specific) :: identifier (C++ specific) :: operator-function-name (C++ specific) ::qualified-name (C++ specific) <i>(expression)</i> <i>name</i></p> <p>literal: <i>integer-constant</i> <i>character-constant</i> <i>floating-constant</i> <i>string-literal</i></p> <p>name: <i>identifier</i> <i>operator-function-name</i> (C++ specific) <i>conversion-function-name</i> (C++ specific) <i>~ class-name</i> <i>qualified-name</i> (C++ specific)</p> <p>qualified-name: (C++ specific) <i>qualified-class-name :: name</i></p> <p>postfix-expression: <i>primary-expression</i> <i>postfix-expression [expression]</i> <i>postfix-expression (<expression-list>)</i> <i>simple-type-name (<expression-list>)</i> (C++ specific) <i>postfix-expression . name</i> <i>postfix-expression -> name</i> <i>postfix-expression ++</i> <i>postfix-expression --</i></p> <p>expression-list: <i>assignment-expression</i> <i>expression-list , assignment-expression</i></p> <p>unary-expression: <i>postfix-expression</i> ++ unary-expression -- unary-expression <i>unary-operator cast-expression</i> sizeof unary-expression sizeof (type-name) <i>allocation-expression</i> (C++ specific) <i>deallocation-expression</i> (C++ specific)</p> <p>unary-operator: one of & * + - ~ !</p> <p>allocation-expression: (C++ specific) <::> new <placement> new-type-name <initializer> <::> new <placement> (type-name) <initializer></p> <p>placement: (C++ specific) <i>(expression-list)</i></p> <p>new-type-name: (C++ specific) <i>type-specifiers <new-declarator></i></p> <p>new-declarator: (C++ specific) <i>ptr-operator <new-declarator></i> <i>new-declarator [<expression>]</i></p> <p>deallocation-expression: (C++ specific) <::> delete cast-expression <::> delete [] cast-expression</p> <p>cast-expression: <i>unary-expression</i></p>	<p><i>(type-name) cast-expression</i></p> <p>pm-expression: <i>cast-expression</i> <i>pm-expression * cast-expression</i> (C++ specific) <i>pm-expression ->* cast-expression</i> (C++ specific)</p> <p>multiplicative-expression: <i>pm-expression</i> <i>multiplicative-expression * pm-expression</i> <i>multiplicative-expression / pm-expression</i> <i>multiplicative-expression % pm-expression</i></p> <p>additive-expression: <i>multiplicative-expression</i> <i>additive-expression + multiplicative-expression</i> <i>additive-expression - multiplicative-expression</i></p> <p>shift-expression: <i>additive-expression</i> <i>shift-expression << additive-expression</i> <i>shift-expression >> additive-expression</i></p> <p>relational-expression: <i>shift-expression</i> <i>relational-expression < shift-expression</i> <i>relational-expression > shift-expression</i> <i>relational-expression <= shift-expression</i> <i>relational-expression >= shift-expression</i></p> <p>equality-expression: <i>relational-expression</i> <i>equality expression == relational-expression</i> <i>equality expression != relational-expression</i></p> <p>AND-expression: <i>equality-expression</i> <i>AND-expression & equality-expression</i></p> <p>exclusive-OR-expression: <i>AND-expression</i> <i>exclusive-OR-expression ^ AND-expression</i></p> <p>inclusive-OR-expression: <i>exclusive-OR-expression</i> <i>inclusive-OR-expression exclusive-OR-expression</i></p> <p>logical-AND-expression: <i>inclusive-OR-expression</i> <i>logical-AND-expression && inclusive-OR-expression</i></p> <p>logical-OR-expression: <i>logical-AND-expression</i> <i>logical-OR-expression logical-AND-expression</i></p> <p>conditional-expression: <i>logical-OR-expression</i> <i>logical-OR-expression ? expression : conditional-expression</i></p> <p>assignment-expression: <i>conditional-expression</i> <i>unary-expression assignment-operator assignment-expression</i></p> <p>assignment-operator: one of = *= /= %= += -= <<= >>= &= ^= =</p> <p>expression: <i>assignment-expression</i> <i>expression, assignment-expression</i></p> <p>constant-expression: <i>conditional-expression</i></p>
--	--

Expressions and

C++

C++ allows the overloading of certain standard C operators, as explained starting on page 480. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator `==` might be defined in **class complex** to test the equality of two complex numbers without changing its normal usage with non-class data types. An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the rules for operators and conversions discussed in this section may not apply to expressions in C++.

Evaluation order

The order in which Turbo C++ evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. Consider the expression

```
i = v[i++]; // i is undefined
```

The value of *i* depends on whether *i* is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for *sum* and *total*. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value.

Turbo C++ regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the re-arrangement affect the value of the expression.

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables *a*, *b*, *c*, and *f*, then the expression $f = a + (b + c)$ forces $(b + c)$ to be evaluated before adding the result to *a*.

Errors and overflows

See ***matherr()*** and ***signal()*** in *online Help*.

We've summarized the precedence and associativity of the operators in Table 12.10. During the evaluation of an expression, Turbo C++ can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2^n arithmetic on *n*-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.

Operators

The Turbo C++ operators described here are the standard ANSI C operators.

Unless the operators are overloaded, the following information is true in both C and C++. In C++ you can overload all of these operators with the exception of **.** (member operator) and **?:** (conditional operator) (and you also can't overload the C++ operators **::** and **.***).

If an operator is overloaded, the discussion may not be true for it anymore. Table 12.11 on page 421 gives the syntax for all operators and operator expressions.

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Turbo C++ is especially rich in operators, offering not only the common arithmetical and logical operators, but also many for bit-level manipulations, structure and union component access, and pointer operations (referencing and dereferencing).



C++ extensions offer additional operators for accessing class members and their objects, together with a mechanism for over-

Overloading is covered starting on page 479.

loading operators. *Overloading* lets you redefine the action of any standard operators when applied to the objects of a given class. In this section, we confine our discussion to the standard operators of Turbo C++.

After defining the standard operators, we discuss data types and declarations, and explain how these affect the actions of each operator. From there we can proceed with the syntax for building expressions from operators, punctuators, and objects.

The operators in Turbo C++ are defined as follows:

operator: one of

[]	()	.	->	++	--
&	*	+	-	~	!
sizeof	/	%	<<	>>	<
>	<=	>=	==	!=	^
	&&		?:	=	*=
/=	%=	+=	-=	<<=	>>=
&=	^=	=	,	#	##

The operators # and ## are used only by the preprocessor (see page 501).

And the following operators specific to C++:

:: .* ->*

Except for [], (), and ?:, which bracket expressions, the multicharacter operators are considered as single tokens. The same operator token can have more than one interpretation, depending on the context. For example,

A * B	Multiplication
*ptr	Dereference (indirection)
A & B	Bitwise AND
&A	Address operation
int &	Reference modifier (C++)
label:	Statement label
a ? x : y	Conditional statement
void func(int n);	Function declaration
a = (b+c)*d;	Parenthesized expression
a, b, c;	Comma expression
func(a, b, c);	Function call
a = ~b;	Bitwise negation (one's complement)
~func() {delete a;}	Destructor (C++)

Unary operators

&	Address operator
*	Indirection operator
+	Unary plus
-	Unary minus
~	Bitwise complement (1's complement)
!	Logical negation
++	Prefix: preincrement; Postfix: postincrement
--	Prefix: predecrement; Postfix: postdecrement

Binary operators

Additive operators	+	Binary plus (addition)
	-	Binary minus (subtraction)
Multiplicative operators	*	Multiply
	/	Divide
	%	Remainder
Shift operators	<<	Shift left
	>>	Shift right
Bitwise operators	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	 	Bitwise inclusive OR
Logical operators	&&	Logical AND
	 	Logical OR
Assignment operators	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND

	<code>^=</code>	Assign bitwise XOR
	<code> =</code>	Assign bitwise OR
Relational operators	<code><</code>	Less than
	<code>></code>	Greater than
	<code><=</code>	Less than or equal to
	<code>>=</code>	Greater than or equal to
Equality operators	<code>==</code>	Equal to
	<code>!=</code>	Not equal to
Component selection operators	<code>.</code>	Direct component selector
	<code>-></code>	Indirect component selector
Class-member operators	<code>::</code>	Scope access/resolution
	<code>.*</code>	Dereference pointer to class member
	<code>->*</code>	Dereference pointer to class member
	<code>:</code>	Class initializer.
Conditional operator	<code>a ? x : y</code>	"if <i>a</i> then <i>x</i> ; else <i>y</i> "
Comma operator	<code>,</code>	Evaluate; e.g., <i>a</i> , <i>b</i> , <i>c</i> ; from left to right

The operator functions, as well as their syntax, precedences, and associativities, are covered starting on page 419.

Postfix and prefix operators

The six postfix operators `[]` `()` `.` `->` `++` and `--` are used to build postfix expressions as shown in the expressions syntax table (Table 12.11). The increment and decrement operators (`++` and `--`) are also prefix and unary operators; they are discussed starting on page 428.

Array subscript operator `[]`

In the expression

postfix-expression [*expression*]

either *postfix-expression* or *expression* must be a pointer and the other an integral type.

In C, but not necessarily in C++, the expression *exp1*[*exp2*] is defined as

* ((exp1) + (exp2))

where either *exp1* is a pointer and *exp2* is an integer, or *exp1* is an integer and *exp2* is a pointer. (The punctuators [], *, and + can be individually overloaded in C++.)

Function call
operators ()

The expression

postfix-expression(*<arg-expression-list>*)

is a call to the function given by the postfix expression. The *arg-expression-list* is a comma-delimited list of expressions of any type representing the actual (or real) function arguments. The value of the function call expression, if any, is determined by the return statement in the function definition. See “Function calls and argument conversions”, page 408, for more on function calls.

Structure/union
member operator
. (dot)

In the expression

postfix-expression . *identifier*

the postfix expression must be of type structure or union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue. Detailed examples of the use of . and -> for structures are given on page 411.

*lvalues are defined on page
370.*

Structure/union pointer
operator ->

In the expression

postfix-expression -> *identifier*

the postfix expression must be of type pointer to structure or pointer to union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue.

Postfix increment
operator ++

In the expression

postfix-expression ++

the postfix expression is the operand; it must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue (see page 370 for more on modifiable lvalues). The postfix ++ is also

known as the *postincrement* operator. The value of the whole expression is the value of the postfix expression *before* the increment is applied. After the postfix expression is evaluated, the operand is incremented by 1. The increment value is appropriate to the type of the operand. Pointer types are subject to the rules for pointer arithmetic.

Postfix decrement operator -- The postfix decrement, also known as the *postdecrement*, operator follows the same rules as the postfix increment, except that 1 is subtracted from the operand *after* the evaluation.

Increment and decrement operators

The first two unary operators are ++ and --. These are also postfix and prefix operators, so they are discussed here. The remaining six unary operators are covered following this discussion.

Prefix increment operator

In the expression

++ *unary-expression*

the unary expression is the operand; it must be of scalar type and must be a modifiable lvalue. The prefix increment operator is also known as the *preincrement* operator. The operand is incremented by 1 *before* the expression is evaluated; the value of the whole expression is the incremented value of the operand. The 1 used to increment is the appropriate value for the type of the operand. Pointer types follow the rules of pointer arithmetic.

Prefix decrement operator

The prefix decrement, also known as the *predecrement*, operator has the following syntax:

-- *unary-expression*

It follows the same rules as the prefix increment operator, except that the operand is decremented by 1 before the whole expression is evaluated.

Unary operators

The six unary operators (aside from ++ and --) are & * + - ~ and !. The syntax is

unary-operator cast-expression

cast-expression:

unary-expression

(type-name) cast-expression

Address operator & The **&** operator and ***** operator (the ***** operator is described in the next section) work together as the *referencing* and *dereferencing* operators. In the expression

The symbol **&** is also used in C++ to specify reference types; see page 449.

& *cast-expression*

the *cast-expression* operand must be either a function designator or an lvalue designating an object that is not a bit field and is not declared with the **register** storage class specifier. If the operand is of type **type**, the result is of type pointer to **type**.



Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer to X" types when appearing in certain contexts. The **&** operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following extract:

```
type t1 = 1, t2 = 2;
type *ptr = &t1;    // initialized pointer
*ptr = t2;         // same effect as t1 = t2
```

type **ptr = &t1* is treated as

```
T *ptr;
ptr = &t1;
```

so it is *ptr*, not **ptr*, that gets assigned. Once *ptr* has been initialized with the address *&t1*, it can be safely dereferenced to give the lvalue **ptr*.

Indirection operator * In the expression

***** *cast-expression*

the *cast-expression* operand must have type "pointer to **type**," where **type** is any type. The result of the indirection is of type **type**. If the operand is of type "pointer to function," the result is a function designator; if the operand is a pointer to an object, the result is an lvalue designating that object. In the following situations, the result of indirection is undefined:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Unary plus operator + In the expression

+ *cast-expression*

the *cast-expression* operand must be of arithmetic type. The result is the value of the operand after any required integral promotions.

Unary minus operator - In the expression

- *cast-expression*

the *cast-expression* operand must be of arithmetic type. The result is the negative of the value of the operand after any required integral promotions.

Bitwise complement operator ~ In the expression

~ *cast-expression*

the *cast-expression* operand must be of integral type. The result is the bitwise complement of the operand after any required integral promotions. Each 0 bit in the operand is set to 1, and each 1 bit in the operand is set to 0.

Logical negation operator ! In the expression

! *cast-expression*

the *cast-expression* operand must be of scalar type. The result is of type **int** and is the logical negation of the operand: 0 if the operand is nonzero; 1 if the operand is zero. The expression *!E* is equivalent to $(0 == E)$.

The **sizeof** operator

There are two distinct uses of the **sizeof** operator:

sizeof *unary-expression*

sizeof (*type-name*)

How much space is set aside for each type depends on the machine.

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). In the first use,

the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is *not* converted to a pointer type). The number of elements in an array equals **sizeof array / sizeof array[0]**.

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

sizeof cannot be used with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is **size_t**, defined as **unsigned int** in `stddef.h`.

You can use **sizeof** in preprocessor directives; this is specific to Turbo C++.



In C++, **sizeof(class_{type})**, where *class_{type}* is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Multiplicative operators

There are three multiplicative operators: *****, **/** and **%**. The syntax is

multiplicative-expression:

cast-expression

*multiplicative-expression * cast-expression*

multiplicative-expression / cast-expression

multiplicative-expression % cast-expression

The operands for ***** (multiplication) and **/** (division) must be of arithmetical type. The operands for **%** (modulus, or remainder) must be of integral type. The usual arithmetic conversions are made on the operands (see page 385).

The result of (*op1 * op2*) is the product of the two operands. The results of (*op1 / op2*) and (*op1 % op2*) are the quotient and remainder, respectively, when *op1* is divided by *op2*, provided that *op2* is nonzero. Use of **/** or **%** with a zero second operand results in an error.

When *op1* and *op2* are integers and the quotient is not an integer, the results are as follows:

Rounding is always toward zero.

1. If *op1* and *op2* have the same sign, *op1 / op2* is the largest integer less than the true quotient, and *op1 % op2* has the sign of *op1*.
2. If *op1* and *op2* have opposite signs, *op1 / op2* is the smallest integer greater than the true quotient, and *op1 % op2* has the sign of *op1*.

Additive operators

There are two additive operators: + and -. The syntax is

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

The addition operator +

The legal operand types for *op1 + op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. *op1* is of integral type, and *op2* is of pointer to object type.
3. *op2* is of integral type, and *op1* is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In cases 2 and 3, the rules of pointer arithmetic apply. (Pointer arithmetic is covered on page 402.)

The subtraction operator -

The legal operand types for *op1 - op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. Both *op1* and *op2* are pointers to compatible object types. The unqualified type **type** is considered to be compatible with the qualified types **const type**, **volatile type**, and **const volatile type**.
3. *op1* is of pointer to object type, and *op2* is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

Bitwise shift operators

There are two bitwise shift operators: `<<` and `>>`. The syntax is

```
shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
```

Bitwise shift operators (`<<` and `>>`)

In the expressions `E1 << E2` and `E1 >> E2`, the operands `E1` and `E2` must be of integral type. The normal integral promotions are performed on `E1` and `E2`, and the type of the result is the type of the promoted `E1`. If `E2` is negative or is greater than or equal to the width in bits of `E1`, the operation is undefined.

The constants `ULONG_MAX` and `UINT_MAX` are defined in `limits.h`.

The result of `E1 << E2` is the value of `E1` left-shifted by `E2` bit positions, zero-filled from the right if necessary. Left shifts of an **unsigned long** `E1` are equivalent to multiplying `E1` by 2^{E2} , reduced modulo `ULONG_MAX + 1`; left shifts of **unsigned ints** are equivalent to multiplying by 2^{E2} reduced modulo `UINT_MAX + 1`. If `E1` is a signed integer, the result must be interpreted with care, since the sign bit may change.

The result of `E1 >> E2` is the value of `E1` right-shifted by `E2` bit positions. If `E1` is of **unsigned** type, zero-fill occurs from the left if necessary. If `E1` is of **signed** type, the fill from the left uses the sign bit (0 for positive, 1 for negative `E1`). This sign-bit extension ensures that the sign of `E1 >> E2` is the same as the sign of `E1`. Except for signed types, the value of `E1 >> E2` is the integral part of the quotient $E1/2^{E2}$.

Relational operators

There are four relational operators: `<` `>` `<=` and `>=`. The syntax for these operators is:

```
relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression
```

The less-than operator <

In the expression $E1 < E2$, the operands must conform to one of the following sets of conditions:

Qualified names are defined on page 461.

1. Both $E1$ and $E2$ are of arithmetic type.
2. Both $E1$ and $E2$ are pointers to qualified or unqualified versions of compatible object types.
3. Both $E1$ and $E2$ are pointers to qualified or unqualified versions of compatible incomplete types.

In case 1, the usual arithmetic conversions are performed. The result of $E1 < E2$ is of type **int**. If the value of $E1$ is less than the value of $E2$, the result is 1 (true); otherwise, the result is zero (false).

In cases 2 and 3, where $E1$ and $E2$ are pointers to compatible types, the result of $E1 < E2$ depends on the relative locations (addresses) of the two objects being pointed at. When comparing structure members within the same structure, the "higher" pointer indicates a later declaration. Within arrays, the "higher" pointer indicates a larger subscript value. All pointers to members of the same union object compare as equal.

Normally, the comparison of pointers to different structure, array, or union objects, or the comparison of pointers outside the range of an array object give undefined results; however, an exception is made for the "pointer beyond the last element" situation as discussed under "Pointer arithmetic" on page 402. If P points to an element of an array object, and Q points to the last element, the expression $P < Q + 1$ is allowed, evaluating to 1 (true), even though $Q + 1$ does not point to an element of the array object.

The greater-than operator >

The expression $E1 > E2$ gives 1 (true) if the value of $E1$ is greater than the value of $E2$; otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

The less-than or equal-to operator <=

Similarly, the expression $E1 <= E2$ gives 1 (true) if the value of $E1$ is less than or equal to the value of $E2$. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

The greater-than or equal-to operator `>=`

Finally, the expression $E1 \geq E2$ gives 1 (true) if the value of $E1$ is greater than or equal to the value of $E2$. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

Equality operators

There are two equality operators: `==` and `!=`. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.



However, `==` and `!=` have a lower precedence than the relational operators `<` and `>`, `<=`, and `>=`. Also, `==` and `!=` can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

```
equality-expression:  
  relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression
```

The equal-to operator `==`

In the expression $E1 == E2$, the operands must conform to one of the following sets of conditions:

1. Both $E1$ and $E2$ are of arithmetic type.
2. Both $E1$ and $E2$ are pointers to qualified or unqualified versions of compatible types.
3. One of $E1$ and $E2$ is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**.
4. One of $E1$ or $E2$ is a pointer and the other is a null pointer constant.

If $E1$ and $E2$ have types that are valid operand types for a relational operator, the same comparison rules just detailed for $E1 < E2$, $E1 \leq E2$, and so on, apply.

In case 1, for example, the usual arithmetic conversions are performed, and the result of $E1 == E2$ is of type **int**. If the value of $E1$ is equal to the value of $E2$, the result is 1 (true); otherwise, the result is zero (false).

In case 2, $E1 == E2$ gives 1 (true) if $E1$ and $E2$ point to the same object, or both point "one past the last element" of the same array object, or both are null pointers.

If $E1$ and $E2$ are pointers to function types, $E1 == E2$ gives 1 (true) if they are both null or if they both point to the same function. Conversely, if $E1 == E2$ gives 1 (true), then either $E1$ and $E2$ point to the same function, or they are both null.

In case 4, the pointer to an object or incomplete type is converted to the type of the other operand (pointer to a qualified or unqualified version of **void**).

The inequality operator $!=$ The expression $E1 != E2$ follows the same rules as those for $E1 == E2$, except that the result is 1 (true) if the operands are unequal, and 0 (false) if the operands are equal.

Bitwise AND operator &

The syntax is

AND-expression:
equality-expression
AND-expression & equality-expression

In the expression $E1 \& E2$, both operands must be of integral type. The usual arithmetical conversions are performed on $E1$ and $E2$, and the result is the bitwise AND of $E1$ and $E2$. Each bit in the result is determined as shown in Table 12.12.

Table 12.12
Bitwise operators truth table

Bit value in $E1$	Bit value in $E2$	$E1 \& E2$	$E1 \wedge E2$	$E1 E2$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Bitwise exclusive OR operator ^

The syntax is

exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ AND-expression

In the expression $E1 \wedge E2$, both operands must be of integral type. The usual arithmetic conversions are performed on $E1$ and $E2$, and the result is the bitwise exclusive OR of $E1$ and $E2$. Each bit in the result is determined as shown in Table 12.12.

Bitwise inclusive OR operator **|**

The syntax is

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | exclusive-OR-expression

In the expression $E1 | E2$, both operands must be of integral type. The usual arithmetic conversions are performed on $E1$ and $E2$, and the result is the bitwise inclusive OR of $E1$ and $E2$. Each bit in the result is determined as shown in Table 12.12.

Logical AND operator **&&**

The syntax is

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && inclusive-OR-expression

In the expression $E1 \&\& E2$, both operands must be of scalar type. The result is of type **int**, the result is 1 (true) if the values of $E1$ and $E2$ are both nonzero; otherwise, the result is 0 (false).

Unlike the bitwise **&** operator, **&&** guarantees left-to-right evaluation. $E1$ is evaluated first; if $E1$ is zero, $E1 \&\& E2$ gives 0 (false), and $E2$ is not evaluated.

Logical OR operator **||**

The syntax is

logical-OR-expression:
logical-AND-expression
logical-OR-expression || logical-AND-expression

In the expression $E1 || E2$, both operands must be of scalar type. The result is of type **int**, and the result is 1 (true) if either of the values of $E1$ and $E2$ are nonzero. Otherwise, the result is 0 (false).

Unlike the bitwise `|` operator, `||` guarantees left-to-right evaluation. `E1` is evaluated first; if `E1` is nonzero, `E1 || E2` gives 1 (true), and `E2` is not evaluated.

Conditional operator `?` :

The syntax is

```
conditional-expression  
logical-OR-expression  
logical-OR-expression ? expression : conditional-expression
```

In C++, the result is an lvalue.

In the expression `E1 ? E2 : E3`, the operand `E1` must be of scalar type. The operands `E2` and `E3` must obey one of the following sets of rules:

1. Both are of arithmetic type.
2. Both are of compatible structure or union types.
3. Both are of **void** type.
4. Both are of type pointer to qualified or unqualified versions of compatible types.
5. One operand is of pointer type, the other is a null pointer constant.
6. One operand is of type pointer to an object or incomplete type, the other is of type pointer to a qualified or unqualified version of void.

First, `E1` is evaluated; if its value is nonzero (true), then `E2` is evaluated and `E3` is ignored. If `E1` evaluates to zero (false), then `E3` is evaluated and `E2` is ignored. The result of `E1 ? E2 : E3` will be the value of whichever of `E2` and `E3` is evaluated.

In case 1, both `E2` and `E3` are subject to the usual arithmetic conversions, and the type of the result is the common type resulting from these conversions. In case 2, the type of the result is the structure or union type of `E2` and `E3`. In case 3, the result is of type **void**.

In cases 4 and 5, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands. In case 6, the type of the result is that of the nonpointer-to-void operand.

Assignment operators

There are eleven assignment operators. The = operator is the simple assignment operator; the other ten are known as compound assignment operators.

The syntax is

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += -=
<<= >>= &= ^= |=

The simple assignment operator =

In the expression $E1 = E2$, $E1$ must be a modifiable lvalue. The value of $E2$, after conversion to the type of $E1$, is stored in the object designated by $E1$ (replacing $E1$'s previous value). The value of the assignment expression is the value of $E1$ after the assignment. The assignment expression is not itself an lvalue.

In C++, the result is an lvalue.

The operands $E1$ and $E2$ must obey one of the following sets of rules:

1. $E1$ is of qualified or unqualified arithmetic type and $E2$ is of arithmetic type.
2. $E1$ has a qualified or unqualified version of a structure or union type compatible with the type of $E2$.
3. $E1$ and $E2$ are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
4. One of $E1$ or $E2$ is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
5. $E1$ is a pointer and $E2$ is a null pointer constant.

The compound assignment operators

The compound assignments $op=$, where op can be any one of the ten operator symbols * / % + - << >> & ^ |, are interpreted as follows:

$E1\ op =\ E2$

has the same effect as

$$E1 = E1 \text{ op } E2$$

except that the lvalue $E1$ is evaluated only once. For example, $E1 += E2$ is the same as $E1 = E1 + E2$.

The rules for compound assignment are therefore covered in the previous section (on the simple assignment operator =).

Comma operator

The syntax is

expression:
assignment-expression
expression , assignment-expression

In C++, the result is an lvalue.

In the comma expression

$$E1, E2$$

the left operand $E1$ is evaluated as a **void** expression, then $E2$ is evaluated to give the result and type of the comma expression. By recursion, the expression

$$E1, E2, \dots, E_n$$

results in the left-to-right evaluation of each E_i , with the value and type of E_n giving the result of the whole expression. To avoid ambiguity with the commas used in function argument and initializer lists, parentheses must be used. For example,

```
func(i, (j = 1, j + 4), k);
```

calls **func()** with three arguments, not four. The arguments are i , 5, and k .

C++ operators

See page 452 for information
on the scope access
operator ::.

The operators specific to C++ are ::, *, >*. The syntax for the * and >* operators is as follows:

pm-expression
cast-expression
pm expression . cast-expression*
pm expression -> cast-expression*

The * operator dereferences pointers to class members. It binds the *cast-expression*, which must be of type "pointer to member of

class *type*", to the *pm-expression*, which must be of class *type* or of a class publicly derived from class *type*. The result is an object or function of the type specified by the *cast-expression*.

The `->*` operator dereferences pointers to pointers to class members (no, that isn't a typo; it does indeed dereference pointers to pointers). It binds the *cast-expression*, which must be of type "pointer to member of *type*," to the *pm-expression*, which must be of type pointer to *type* or of type "pointer to class publicly derived from *type*." The result is an object or function of the type specified by the *cast-expression*.

If the result of either of these operators is a function, you can only use that result as the operand for the function call operator `()`. For example,

```
(ptr2object->*ptr2memberfunc) (10);
```

calls the member function denoted by *ptr2memberfunc* for the object pointed to be *ptr2object*.

Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. Table 12.13 on page 441 lays out the syntax for statements:

Blocks

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces `{ }`. Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

Table 12.13: Turbo C++ statements

statement:

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement
asm-statement
declaration (C++ specific)

asm-statement:

```
asm tokens newline  
asm tokens;  
asm { tokens; <tokens;>=  
    <tokens;>  
}
```

Table 12.13: Turbo C++ statements (continued)

<p><i>labeled-statement:</i> identifier : <i>statement</i> case <i>constant-expression</i> : <i>statement</i> default : <i>statement</i></p> <p><i>compound-statement:</i> { <<i>declaration-list</i>> <<i>statement-list</i>> }</p> <p><i>declaration-list:</i> <i>declaration</i> <i>declaration-list</i> <i>declaration</i></p> <p><i>statement-list:</i> <i>statement</i> <i>statement-list</i> <i>statement</i></p> <p><i>expression-statement:</i> <<i>expression</i>>;</p>	<p><i>selection-statement:</i> if (<i>expression</i>) <i>statement</i> if (<i>expression</i>) <i>statement</i> else <i>statement</i> switch (<i>expression</i>) <i>statement</i></p> <p><i>iteration-statement:</i> while (<i>expression</i>) <i>statement</i> do <i>statement</i> while (<i>expression</i>); for (<i>for-init-statement</i> <<i>expression</i>> ; <<i>expression</i>>) <i>statement</i></p> <p><i>for-init-statement</i> <i>expression-statement</i> <i>declaration</i> (C++ specific)</p> <p><i>jump-statement:</i> goto <i>identifier</i> ; continue ; break ; return <<i>expression</i>>;</p>
---	---

Labeled statements

A statement can be labeled in the following ways:

1. *label-identifier* : *statement*

The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and enjoy function scope. In C++ you can label both declaration and non-declaration statements.



2. **case** *constant-expression* : *statement* **default** : *statement*

Case and default labeled statements are used only in conjunction with switch statements.

Expression statements

Any expression followed by a semicolon forms an *expression statement*:

```
<expression>;
```

Turbo C++ executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls.

A special case is the *null statement*, consisting of a single semicolon (;). The null statement does nothing. It is nevertheless useful in situations where the Turbo C++ syntax expects a statement but your program does not need one.

Selection statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

if statements

The basic **if** statement has the following pattern:

```
if (cond-expression) t-st <else f-st>
```

The parentheses around *cond-expression* are essential.

The *cond-expression* must be of scalar type. The expression is evaluated. If the value is zero (or null for pointer types), we say that the *cond-expression* is false; otherwise, it is true.

If there is no **else** clause and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored.

If the optional **else** *f-st* is present and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored and *f-st* is executed.



Unlike, say, Pascal, Turbo C++ does not have a specific Boolean data type. Any expression of integer or pointer type can serve a Boolean role in conditional tests. The relational expression ($a > b$) (if legal) evaluates to **int** 1 (true) if ($a > b$), and to **int** 0 (false) if ($a \leq b$). Pointer conversions are such that a pointer can always be correctly compared to a constant expression evaluating to 0. That is, the test for null pointers can be written `if (!ptr) ...` or `if (ptr == 0) ...`.

The *f-st* and *t-st* statements can themselves be **if** statements, allowing for a series of conditional tests nested to any depth. Care is needed with nested **if...else** constructs to ensure that the correct statements are selected. There is no **endif** statement: Any “else” ambiguity is resolved by matching an **else** with the last encountered **if-without-an-else** at the same block level. For example,

```
if (x == 1)
    if (y == 1) puts("x=1 and y=1");
    else puts("x != 1");
```

draws the wrong conclusion! The **else** matches with the second **if**, despite the indentation. The correct conclusion is that $x = 1$ and $y \neq 1$. Note the effect of braces:

```
if (x == 1) {
    if (y == 1) puts("x = 1 and y = 1");
```

```
}  
else puts("x != 1"); // correct conclusion
```

switch statements The **switch** statement uses the following basic format:

switch (*sw-expression*) *case-st*

A **switch** statement lets you transfer control to one of several case-labeled statements, depending on the value of *sw-expression*. The latter must be of integral type (in C++, it can be of class type, provided that there is an unambiguous conversion to integral type available). Any statement in *case-st* (including empty statements) can be labeled with one or more case labels:

It is illegal to have duplicate case constants in the same switch statement.

case *const-exp-i* : *case-st-i*

where each case constant, *const-exp-i*, is a constant expression with a unique integer value (converted to the type of the controlling expression) within its enclosing **switch** statement.

There can also be at most one **default** label:

default : *default-st*

After evaluating *sw-expression*, a match is sought with one of the *const-exp-i*. If a match is found, control passes to the statement *case-st-i* with the matching case label.

If no match is found and there is a **default** label, control passes to *default-st*. If no match is found and there is no **default** label, none of the statements in *case-st* is executed. Program execution is not affected when **case** and **default** labels are encountered. Control simply passes through the labels to the following statement or switch. To stop execution at the end of a group of statements for a particular case, use **break**.

Iteration statements

Iteration statements let you loop a set of statements. There are three forms of iteration in Turbo C++: **while**, **do**, and **for** loops.

while statements

The parentheses are essential.

The general format for this statement is

while (*cond-exp*) *t-st*

The loop statement, *t-st*, will be executed repeatedly until the conditional expression, *cond-exp*, compares equal to zero (false).

The *cond-exp* is evaluated and tested first (as described on page 443). If this value is nonzero (true), *t-st* is executed; if no jump statements that exit from the loop are encountered, *cond-exp* is evaluated again. This cycle repeats until *cond-exp* is zero.

As with **if** statements, pointer type expressions can be compared with the null pointer, so that **while** (*ptr*) ... is equivalent to

```
while (ptr != NULL)...
```

The **while** loop offers a concise method for scanning strings and other null-terminated data structures:

```
char str[10]="Borland";
char *ptr=&str[0];
int count=0;
//...
while (*ptr++) // loop until end of string
    count++;
```

In the absence of jump statements, *t-st* must affect the value of *cond-exp* in some way, or *cond-exp* itself must change during evaluation in order to prevent unwanted endless loops.

do while statements

The general format is

```
do do-st while (cond-exp);
```

The *do-st* statement is executed repeatedly until *cond-exp* compares equal to zero (false). The key difference from the **while** statement is that *cond-exp* is tested *after*, rather than before, each execution of the loop statement. At least one execution of *do-st* is assured. The same restrictions apply to the type of *cond-exp* (scalar).

for statements

The **for** statement format in C is

For C++, <init-exp> can be an expression or a declaration.

```
for (<init-exp>; <test-exp>; <increment-exp>) statement
```

The sequence of events is as follows:

1. The initializing expression *init-exp*, if any, is executed. As the name implies, this usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity (including declarations in C++). Hence the claim that any C program can be written as a single **for** loop. (But don't try this at home. Such stunts are performed by trained professionals.)

2. The expression *test-exp* is evaluated following the rules of the **while** loop. If *test-exp* is nonzero (true), the loop statement is executed. An empty expression here is taken as **while (1)**, that is, always true. If the value of *test-exp* is zero (false), the **for** loop terminates.
3. *increment-exp* advances one or more counters.
4. The expression *statement* (possibly empty) is evaluated and control returns to step 2.

If any of the optional elements are empty, appropriate semicolons are required:

```
for (;;) { // same as for (; 1;)
    // loop forever
}
```



The C rules for **for** statements apply in C++. However, the *init-exp* in C++ can also be a declaration. The scope of a declared identifier extends through the enclosing loop. For example,

```
for (int i = 1; i < 3; ++i) {
    if (i ...) ... // ok to refer to i here
    ...
    for (int x = 0;;); // do nothing
}
if (i...) // legal
if (x...) // illegal; x is now out of scope
```

Jump statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**.

break statements The syntax is

break;

A **break** statement can be used only inside an iteration (**while**, **do**, and **for** loops) or a **switch** statement. It terminates the iteration or **switch** statement. Since iteration and **switch** statements can be intermixed and nested to any depth, take care to ensure that your **break** exits from the correct loop or switch. The rule is that a **break** terminates the *nearest* enclosing iteration or **switch** statement.

continue statements The syntax is

continue;

A **continue** statement can be used only inside an iteration statement; it transfers control to the test condition for **while** and **do** loops, and to the increment expression in a **for** loop.

With nested iteration loops, a **continue** statement is taken as belonging to the *nearest* enclosing iteration.

goto statements The syntax is

goto label;

The **goto** statement transfers control to the statement labeled *label* (see "Labeled statements," page 442), which must be in the same function.



In C++, it is illegal to bypass a declaration having an explicit or implicit initializer unless that declaration is within an inner block that is also bypassed.

return statements Unless the function return type is **void**, a function body must contain at least one **return** statement with the following format:

return *return-expression*;

where *return-expression* must be of type **type** or of a type that is convertible to **type** by assignment. The value of the *return-expression* is the value returned by the function. An expression that calls the function, such as `func(actual-arg-list)`, is an rvalue of type **type**, not an lvalue:

```
t = func(arg);           // OK
func(arg) = t;          /* illegal in C; legal in C++ if return type of
                        func is a reference */
(func(arg))++;         /* illegal in C; legal in C++ if return type of
                        func is a reference */
```

The execution of a function call terminates if a **return** statement is encountered; if no **return** is met, execution continues, ending at the final closing brace of the function body.

If the return type is **void**, the **return** statement can be written as

```
{  
    ...  
    return;  
}
```

with no return expression; alternatively, the **return** statement can be omitted.

C++ specifics

C++ is basically a superset of C. This means that, generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations need special care. For example, the same function **func()** declared twice in C with different argument types will invoke a duplicated name error. Under C++, however, **func()** will be interpreted as an overloaded function — whether this is legal or not will depend on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. We first review these aspects of C++ that can be used independently of classes, then get into the specifics of classes and class mechanisms.

Referencing

Pointer referencing and dereferencing is discussed on page 429.

While in C you pass arguments only by value, in C++ you can pass arguments by value or by reference. C++ reference types, which are closely related to pointer types, create aliases for objects and let you pass arguments to functions by reference.

Simple references

Note that `type& var`, `type &var`, and `type & var` are all equivalent.

The reference declarator can be used to declare references outside functions:

```
int i = 0;
int &ir = i; // ir is an alias for i
ir = 2;      // same effect as i = 2
```

This creates the lvalue *ir* as an alias for *i*, provided that the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, `ir = 2` assigns 2 to *i*, and `&ir` returns the address of *i*.

Reference arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir); // ir is type "reference to int"
...
int sum=3;
func1(sum);           // sum passed by value
func2(sum);           // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by **func2()**. On the other hand, **func1()** gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by **func1()**.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can return a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The C method for changing *x* uses the actual argument `&x`, the address of `&x`, rather than `&x` itself. Although `&x` is passed by value, the function can access *x* through the copy of `&x` it receives. Even if the function does not need to change *x*, it is still useful (though subject to possibly dangerous side effects) to pass `&x`, especially if *x* is a large data structure. Passing *x* directly by value involves the wasteful copying of the data structure.

Compare the three implementations of the function **treble()**:

```
Implementation 1      int treble_1(n)
                     {
                       return 3*n;
                     }
                     ...
                     int x, i = 4;
                     x = treble_1(i);      // x now = 12, i = 4
                     ...

Implementation 2      void treble_2(int* np)
                     {
                       *np = (*np)*3;
                     }
                     ...
                     treble_2(int &i);    // i now = 12

Implementation 3      void treble_3(int& n)  // n is a reference type
                     {
                       n = 3*n;
                     }
                     ...
                     treble_3(i);        // i now = 36
```

The formal argument declaration `type& t` (or equivalently, `type &t`) establishes `t` as type "reference to **type**." So, when **treble_3()** is called with the real argument `i`, `i` is used to initialize the formal reference argument `n`. `n` therefore acts as an alias for `i`, so that `n = 3*n` also assigns `3 * i` to `i`.

If the initializer is a constant or an object of a different type than the reference type, Turbo C++ creates a temporary object for which the reference acts as an alias:

```
int& ir = 6;    /* temporary int object created, aliased by ir, gets
                value 6 */

float f;
int& ir2 = f;  /* creates temporary int object aliased by ir2; f
                converted before assignment */

ir2 = 2.0      // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

Scope access operator

The scope access (or resolution) operator `::` (two semicolons) lets you access a global (or file duration) name even if it is hidden by a local redeclaration of that name (see page 371 for more on scope):

This code also works if the global `i` is a file-level static.

```
int i;                // global i
...
void func(void);
{
    int i=0;          // local i hides global i
    i = 3;            // this i is the local i
    ::i = 4;          // this i is the global i
    printf ("%d",i); // prints out 3
}
```

The `::` operator has other uses with class types, as discussed throughout this chapter.

The `new` and `delete` operators

The **new** and **delete** operators offer dynamic storage allocation and deallocation, similar but superior to the standard library functions, **malloc()** and **free()**. See the online Help.

A simplified syntax is

```
pointer-to-name = new name <name-initializer>;
delete pointer-to-name;
```

name can be of any type except "function returning..." (however, pointers to functions are allowed).

new tries to create an object of type **name** by allocating (if possible) **sizeof(name)** bytes in *free store* (also called the heap). The storage duration of the new object is from the point of creation until the operator **delete** kills it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the new object. A null pointer indicates a failure (such as insufficient or fragmented heap memory). As with **malloc()**, you need to test for null before trying to access the new object (unless you use a new-handler; see the following section for details). However, unlike **malloc()**, **new** calculates the size of **name** without the need for an explicit **sizeof**

operator. Further, the pointer returned is of the correct type, "pointer to *name*," without the need for explicit casting.

new and delete, being keywords, don't need prototypes.

```
name *nameptr;        // name is any nonfunction type
...
if (!(nameptr = new name)) {
    errmsg("Insufficient memory for name");
    exit (1);
}
// use *nameptr to initialize new name object
...
delete nameptr;      // destroy name and deallocate sizeof(name) bytes
```

Handling errors

You can define a function that will be called if the **new** operator fails (returns 0). To tell the **new** operator about the new-handler function, call **set_new_handler()** and supply a pointer to the new-handler. The prototype for **set_new_handler()** is as follows (from `new.h`):

```
void (*set_new_handler( void (*)() ))();
```

set_new_handler() returns the old new-handler, and changes the function **_new_handler()** so that it, in turn, points to the new-handler that you define. See the online Help for discussions of **set_new_handler()** and **_new_handler()**.

The operator **new** with arrays

If *name* is an array, the pointer returned by **new** points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the left-most dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];    // OK
mat_ptr = new int[n][10][12];   // OK
mat_ptr = new int[3][][12];     // illegal
mat_ptr = new int[][10][12];    // illegal
```

Although the first array dimension may be a variable, all following dimensions must be constants.

The operator **delete** with arrays

You must use the syntax `delete [] expr` when deleting an array. In C++ 2.1, the array dimension should not be specified within the brackets:

```
char * p;

void func()
{
    p = new char[10];    // allocate 10 chars
    delete[] p;        // delete 10 chars
}
```

C++ 2.0 code required the array size. In order to allow 2.0 code to compile, Turbo C++ issues a warning and simply ignores any size that is specified. For example, if the preceding example reads `delete[10] p` and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

With Turbo C++, the `[]` is actually only required when the array element is a class with a destructor. But it is good programming practice to always tell the compiler that an array is being deleted.

The `::operator new`

When used with non-class objects, **new** works by calling a standard library routine, the global `::operator new`. With class objects of type *name*, a specific operator called `name::operator new` can be defined. **new** applied to class *name* objects invokes the appropriate `name::operator new` if present; otherwise, the standard `::operator new` is used.

Initializers with the **new** operator

The optional initializer is another advantage **new** has over **malloc()** (although **calloc()** does clear its allocations to zero). In the absence of explicit initializers, the object created by **new** contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the *default constructor* (see page 470). The user-defined **new** operator with

customized initialization plays a key role in C++ constructors for class-type objects.

Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

```
class-key class-name <: base-list> { <member-list> }
```

class-key is one of **class**, **struct**, or **union**.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class (see page 464, “Base and derived class access”). The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes (see page 462, “Member access control”).

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that may affect which functions can access which members.

Class names

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted (see “Untagged structures and **typedefs**,” page 410.)

Class types

The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```

class X { ... };
X x, &xr, *xpтр, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of
X*/

struct Y { ... };
Y y, &yr, *ypтр, yarray[10];
// C would have
// struct Y y, *ypтр, yarray[10];

union Z { ... };
Z z, &zr, *zpтр, zarray[10];
// C would have
// union Z z, *zpтр, zarray[10];

```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++ they are needed only when the class names, **Y** and **Z**, are hidden (see the following section).

Class name scope

The scope of a class name is local, with some tricks peculiar to classes. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can only be referred to using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union** must be used with the class name. For example,

```

struct S { ... };

int S(struct S *Spтр);

void func(void)
{
    S t;           // ILLEGAL declaration: no class key
                  // and function S in scope
    struct S s;   // OK: elaborated with class key
    S(&s);        // OK: this is a function call
}

```

C++ also allows an incomplete class declaration:

```

class X; // no members, yet!

```

Incomplete declarations permit certain references to class name **X** (usually references to pointers to class objects) before the class has

been fully defined (see “Structure member declarations,” page 410). Of course, you must make a complete class declaration with members before you can define and use class objects.

Class objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including member and friend functions, and the redefinition of standard functions and operators when used with objects of a certain class. Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers a mechanism whereby the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class member list

The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes) and function declarations and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions

A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

The same name can be used to denote more than one function, provided that they differ in argument type or number of arguments.

The keyword **this**

Nonstatic member functions operate on the class type object with which they are called. For example, if *x* is an object of class **X** and **f()** is a member function of **X**, the function call *x.f()* operates on *x*.

Similarly, if *xptr* is a pointer to an **X** object, the function call `xptr->f()` operates on **xptr*. But how does **f()** know which instance of **X** it is operating on? C++ provides **f** with a pointer to *x* called **this**. **this** is passed as a hidden argument in all calls to nonstatic member functions.

The keyword **this** is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If **x.f(y)** is called, for example, where *y* is a member of **X**, **this** is set to *&x* and *y* is set to **this->y**, which is equivalent to *x.y*.

Inline functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline* function.

Turbo C++ can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The *inline* specifier is a request (or hint) to the compiler that you would welcome an inline expansion. As with the **register** storage class specifier, the compiler may or may not take the hint!

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the *operator functions* that implement overloaded operators. For example, the following class declaration:

```
int i;                               // global int

class X {
public:
    char* func(void) { return i; } // inline by default
    char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func() is defined outside the class with an explicit inline specifier. The *i* returned by **func()** is the **char*** *i* of class **X** (see “Member scope,” starting on page 460).

Static members

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each object in its class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If *x* is a static member of class **X**, it can be referenced as **X::x** (even if objects of class **X** haven’t been created yet). It is still possible to access *x* using the normal member access operators. For example, *y.x* and *yptr->x*, where *y* is an object of class **X** and *yptr* is a pointer to an object of class **X**, although the expressions *y* and *yptr* are *not* evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
    int member_int;
public:
    static void func(int i, X* ptr);
};

void g(void);
{
    X obj;
    func(1, &obj);    // error unless there is a global func()
                    // defined elsewhere

    X::func(1, &obj); // calls the static func() in X
                    // OK for static functions only
    obj.func(1, &obj); // so does this (OK for static and
                    // nonstatic functions)
}
```

Since a static member function can be called with no particular object in mind, it has no **this** pointer. A consequence of this is that a static member function cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, **func()** might be defined as follows:

```
void X::func(int i, X* ptr)
{
```

```

member_int = i;           // which object does member_int
                          // refer to? Error
ptr->member_int = i;     // OK: now we know!
}

```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members obey the usual class member access rules, except they can be initialized.

```

class X {
    ...
    static int x;
    ...
};

int X::x = 1;

```

The main use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- reduce the number of visible global names
- make obvious which static objects logically belong to which class
- permit access control to their names

Member scope

The expression **X::func()** in the example on page 458 uses the class name **X** with the scope access modifier to signify that **func()**, although defined “outside” the class, is indeed a member function of **X**, and it exists within the scope of **X**. The influence of **X::** extends into the body of the definition. This explains why the *i* returned by **func()** refers to **X::i**, the **char*** *i* of **X**, rather than the global **int** *i*. Without the **X::** modifier, the function **func()** would

represent an ordinary non-class function, returning the global `int i`.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class **X** can be referenced using the selection operators `.` and `->` (as with C structures). Member functions can also be called using the selection operators (see also “The keyword **this**,” page 457). For example,

```
class X {
public:
    int i;
    char name[20];
    X *ptr1;
    X *ptr2;
    void Xfunc(char*data, X* left, X* right); // define elsewhere
};
void f(void);
{
    X x1, x2, *xptr=&x1;
    x1.i = 0;
    x2.i = x1.i;
    xptr->i = 1;
    x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class **X**, the expression **X::m** is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. A key point is that even if the class name **X** is hidden by a non-type name, the qualified name **X::m** will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class **X** cannot contain objects of class **X**, but can contain pointers or references to objects of class **X** (note the similarity with C’s structure and union types).

Nested types

In C++ 2.1, even tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can in general be accessed only using the **xxx::yyy** notation, except when in the scope of the appropriate class.

A class declared within another class is called a *nested class*. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is purely lexical nesting. The nested

class has no additional privileges in accessing members of the enclosing class (and vice versa).



Classes can be nested in this way to an arbitrary level. For example:

```
struct outer
{
    typedef int t; // 'outer::t' is a typedef name
    struct inner // 'outer::inner' is a class
    {
        static int x;
    };
    static int x;
    int f();
};

int outer::x; //define static data member
int outer::f()
{
    t x; // 't' visible directly here
    return x;
}

int outer::inner::x; //define static data member
outer::t x; // have to use 'outer::t' here
```

With C++ 2.0, any tags or **typedef** names declared inside a class actually belong to the global (file) scope. For example:

```
struct foo
{
    enum bar { x }; // 2.0 rules: 'bar' belongs to file scope
                  // 2.1 rules: 'bar' belongs to 'foo' scope
};

bar x;
```

The preceding fragment compiles without errors. But, because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member access control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

Friend function declarations are not affected by access specifiers (see "friends of classes," page 466).

- public** The member can be used by any function.
- private** The member can be used only by member functions and friends of the class in which it is declared.
- protected** Same as for **private**, but additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in the next section.)

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
    int i;    // X::i is private by default
    char ch; // so is X::ch
public:
    int j;    // next two are public
    int k;
protected:
    int l;    // X::l is protected
};

struct Y {
    int i;    // Y::i is public by default
private:
    int j;    // Y::j is private
public:
    int k;    // Y::k is public
};

union Z {
    int i;    // public by default; no other choice
    double d;
};
```

The access specifiers can be listed and grouped in any convenient sequence. You can save a little typing effort by declaring all the private members together, and so on.

Base and derived class access

When you declare a derived **class D**, you list the base classes **B1**, **B2**, ... in a comma-delimited *base-list*:

```
class-key D : base-list { <member-list> }
```

Since a base class can itself be a derived class, the access attribute question is recursive: You backtrack until you reach the basest of the base classes, those that do not inherit.

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) **D** can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by **D**? **D** may want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

When declaring **D**, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the *base-list*:

Unions cannot have base classes, and unions cannot be used as base classes.

```
class D : public B1, private B2, ... {  
    ...  
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they *can* alter the access attributes of base members as viewed by the derived class.

The default is **private** if **D** is a **class** declaration, and **public** if **D** is a **struct** declaration.

The derived class inherits access attributes from a base class as follows:

public base class: **public** members of the base class are **public** members of the derived class. **Protected** members of the base class are **protected** members of the derived class. **Private** members of the base class remain **private** to the base class.

protected base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **Private** members of the base class remain **private** to the base class.

private base class: Both **public** and **protected** members of the base class are **private** members of the

derived class. **Private** members of the base class remain **private** to the base class.

In both cases, note carefully that **private** members of a base class are, and remain, inaccessible to member functions of the derived class *unless friend* declarations are explicitly declared in the base class granting access. For example,

```
class X : A {           // default for class is private A
...
}
/* class X is derived from class A */

class Y : B, public C { // override default for C
...
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */

struct S : D {         // default for struct is public D
...                   /* struct S is derived from D */
}

struct T : private D, E { // override default for D
                        // E is public by default
...
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations in the derived class. For example,

```
class B {
    int a;           // private by default
public:
    int b, c;
    int Bfunc(void);
};

class X : private B { // a, b, c, Bfunc are now private in X
    int d;           // private by default, NOTE: a is not
                    // accessible in X

public:
    B::c;           // c was private, now is public
    int e;
    int Xfunc(void);
};

int Efunc(X& x);    // external to B and X
```

The function **Efunc()** can use only the public names *c*, *e*, and **Xfunc()**.

The function **Xfunc()** is in **X**, which is derived from **private B**, so it has access to

- The “adjusted-to-public” *c*
- The “private-to-**X**” members from **B**: *b* and **Bfunc()**
- **X**'s own private and public members: *d*, *e*, and **Xfunc()**

However, **Xfunc()** cannot access the “private-to-**B**” member, *a*.

virtual base classes

With multiple inheritance, a base class can't be specified more than once in a derived class:

```
class B { ... };
class D : B, B { ... }; // Illegal
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... } // OK
```

In this case, each object of class **Z** will have two sub-objects of class **B**. If this causes problems, the keyword **virtual** can be added to a base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class **Z** has only one sub-object of class **B**.

friends of classes

A **friend F** of a class **X** is a function or class that, although not a member function of **X**, has full access rights to the private and protected members of **X**. In all other respects, **F** is a normal function with respect to scope, declarations, and definitions.

Since **F** is not a member of **X**, it is not in the scope of **X** and it cannot be called with the $x.F$ and $xptr->F$ selector operators (where x is an **X** object, and $xptr$ is a pointer to an **X** object).

If the specifier **friend** is used with a function declaration or definition within the class **X**, it becomes a friend of **X**.

Friend functions defined within a class obey the same inline rules as member functions (see "Inline functions," page 458). Friend functions are not affected by their position within the class or by any access specifiers. For example,

```
class X {
    int i; // private to X
    friend void friend_func(X*, int);
    /* friend_func is not private, even though it's declared in the
       private section */
public:
    void member_func(int);
};

/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;

/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class **Y** into friends of class **X** with a single declaration:

```
class Y; // incomplete declaration
class X {
    friend Y;
    int i;
    void member_funcX();
};

class Y; { // complete the declaration
    void friend_X1(X&);
    void friend_X2(X*);
    ...
};
```

The functions declared in **Y** are friends of **X**, although they have no **friend** specifiers. They can access the private members of **X**, such as i and **member_funcX**.

It is also possible for an individual member function of class **X** to be a friend of class **Y**:

```
class X {
    ...
    void member_funcX();
}

class Y {
    int i;
    friend void X::member_funcX();
    ...
};
```

Class friendship is not transitive: **X** friend of **Y** and **Y** friend of **Z** does not imply **X** friend of **Z**. However, friendship *is* inherited.

Constructors and destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features.

1. They do not have return value declarations (not even **void**).
2. They cannot be inherited, though a derived class can call the base class's constructors and destructors.
3. Constructors, like most C++ functions, can have default arguments or use member initialization lists.
4. Destructors can be **virtual**, but constructors cannot. (See “**virtual** destructors” on page 478.)
5. You can't take their addresses.

```
int main(void)
{
    ...
    void *ptr = base::base;    // illegal
    ...
}
```

6. Constructors and destructors can be generated by Turbo C++ if they haven't been explicitly defined; they are also invoked

on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.

7. You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
...
    X *p;
...
    p->X::~~X();           // legal call of destructor
    X::X();                // illegal call of constructor
...
}
```

8. The compiler automatically calls constructors and destructors when defining and destroying objects.
9. Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
10. An object with a constructor or destructor cannot be used as a member of a union.

If a **class X** has one or more constructors, one of them is invoked each time you define an object *x* of **class X**. The constructor creates *x* and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the **main()** function is called. When the **#pragma startup** directive is used to install a function prior to the **main()** function, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X
{
public:
    X(); // class X constructor
};
```

A **class X** constructor cannot take **X** as an argument:

```
class X
{
public:
    X(X); // illegal
}
```

The parameters to the constructor can be of any type except that of the class of which it is a member. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the *copy constructor*. A constructor that accepts no parameters is called the *default constructor*. We discuss the default constructor next; the description of the copy constructor starts on page 471.

Constructor defaults

The default constructor for **class X** is one that takes no arguments; it usually has the form `X::X()`. If no user-defined constructors exist for a class, Turbo C++ generates a default constructor. On a declaration such as `X x`, the default constructor creates the object `x`.

Important! Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes *no* arguments and must not be confused with, say, `X::X(int = 0)`, which can be called

with no arguments as a default constructor, or can take an argument.

Take care to avoid ambiguity in calling constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
    X();
    X(int i = 0);
};

int main()
{
    X one(10); // OK; uses X::X(int)
    X two;     // illegal; ambiguous whether to call X::X() or
              // X::X(int = 0)
    return 0;
}
```

The copy constructor

A copy constructor for **class X** is one that can be called with a single argument of type X as follows:

```
X::X(const X&)
or
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x;
X x2 = x1;
X x3(x1);
```

Turbo C++ generates a copy constructor for **class X** if one is needed and no other constructor is defined in **class X**.

See also the discussion of member-by-member class assignment in Section "The assignment **operator=()**," on 484. The assignment operator requires you to define the copy constructor.

Overloading constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X
{
    int    integer_part;
    double double_part;
public:
    X(int i)    { integer_part = i; }
    X(double d) { double_part = d; }
};

main()
{
    X one(10); // invokes X::X(int) and sets integer_part to 10
    X one(3.14); // invokes X::X(double) setting double_part
    ...
    return 0;
}
```

Order of calling constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y(); // base class constructor
X(); // derived class constructor
```

For the case of multiple base classes:

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```

Y(); // base class constructors come first
Z();
X();

```

Constructors for virtual base classes are invoked before any non-virtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first so that the virtual base class may be properly constructed. The code

```

class X : public Y, virtual public Z
X one;

```

produces this order:

```

Z(); // virtual base class initialization
Y(); // non-virtual base class
X(); // derived class

```

Or for a more complicated example:

```

class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;

```

The construction order of view would be as follows:

```

base(); // virtual base class highest in hierarchy
// base is only constructed once
base2(); // non-virtual base of virtual base level2
// must be called to construct level2
level2(); // virtual base class
base2(); // non-virtual base of level1
level1(); // other non-virtual base
toplevel();

```

In the event that a class hierarchy contains multiple instances of a virtual base class, that base class is only constructed once. If, however, there exist both virtual and non-virtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each non-virtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be of the type of the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
    int i;
public:
    X();           // function bodies omitted for clarity
    X(int x);
    X(const X&);
};

main()
{
    X one;        // default constructor invoked
    X two(1);     // constructor X::X(int) is used
    X three = 1; // calls X::X(int)
    X four = one; // invokes X::X(const X&) for copy
    X five(two); // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
    int a, b;
public:
```

```
X(int i, int j) { a = i; b = j }
};
```

An initializer list can be used prior to the function body:

```
class X
{
    int a, b, &c; // Note the reference variable.
public:
    X(int i, int j) : a(i), b(j), c(a) {}
};
```



Important! The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to `x::a` and 2 to `x::b`. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

*Base class constructors must be declared as either **public** or **protected** to be called from a derived class.*

```
class base1
{
    int x;
public:
    base1(int i) { x = i; }
};

class base2
{
    int x;
public:
    base2(int i) : x(i) {}
};

class top : public base1, public base2
{
    int a, b;
public:
    top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j; }
};
```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of **base1** with the value 5 and **base2** with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
    int a, b;
```

```

public:
    X(int i, j) : a(i), b(a+j) {}
};

```

With this class, a declaration of `x x(1,1)` results in an assignment of 1 to `x::a` and 2 to `x::b`.

Base class constructors are called prior to the construction of any of the derived classes members. The values of the derived class can't be changed and then have an affect on the base class's creation.

```

class base
{
    int x;
public:
    base(int i) : x(i) {}
};

class derived : base
{
    int a;
public:
    derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                        // passed an uninitialized a
};

```

With this class setup, a call of derived `d(1)` will *not* result in a value of 10 for the base class member `x`. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```

derived::derived(int i) : a(i)
{
    ...
}

```

Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
class X
{
public:
    ~X(); // destructor for class X
};
```

If a destructor is not explicitly defined for a class, the compiler will generate one.

When destructors are invoked

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see page 472).

atexit(), #pragma exit, and destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the **main()** function, are destroyed as they go out of scope. The order of execution at the end of a Turbo C++ program in these regards is as follows:

- **atexit()** functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

exit() and destructors

When you call **exit()** from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

abort() and destructors

If you call **abort()** anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are only necessary for objects allocated a specific address through calls to **new**.

```
class X {
...
    ~X();
...
};

void* operator new(size_t size, void *ptr)
{
    return ptr;
}

char buffer[sizeof(X)];

main()
{
    X* pointer = new X;
    X* exact_pointer;

    exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer

    ...

    delete pointer;                // delete used to destroy pointer
    exact_pointer->X::~X();         // direct call used to deallocate
}
```

virtual destructors

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
class color
{
public:
    virtual ~color();    // virtual destructor for color
}
```

```

};

class red : public color
{
public:
    ~red();          // destructor for red is also virtual
};

class brightred: public red
{
public:
    ~brightred();   // brightred's destructor also virtual
};

```

The previously listed classes and the following declarations

```

color *palette[3];

palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;

```

will produce these results

```

delete palette[0];
// The destructor for red is called followed by the
// destructor for color.

delete palette[1];
// The destructor for brightred is called, followed by ~red
// and ~color.

delete palette[2];
// The destructor for color is invoked.

```

However, in the event that no destructors were declared as virtual, **delete palette[0]**, **delete palette[1]**, and **delete palette[2]** would all call only the destructor for class **color**. This would incorrectly destruct the first two elements, which were actually of type **red** and **brightred**.

Operator overloading

C++ lets you redefine the action of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators on page 423 can be overloaded except for

. .* :: ?:

The preprocessing symbols # and ## also cannot be overloaded.

The keyword **operator** followed by the operator symbol is called the *operator function name*; it is used like a normal function name when defining the new (overloaded) action of the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function can't alter the number of arguments or the precedence and associativity rules (Table 12.10 on page 420) applying to normal operator use. Consider the class *complex*:

*This class was invented for illustrative purposes only. It isn't the same as the class **complex** in the run-time library.*

```
class complex {
    double real, imag;           // private by default
public:
    ...
    complex() { real = imag = 0; } // inline constructor
    complex(double r, double i = 0) { // another one
        real = r; imag = i;
    }
    ...
}
```

We could easily devise a function for adding complex numbers, say,

```
complex AddComplex(complex c1, complex c2);
```

but it would be more natural to be able to write:

```
complex c1(0,1), c2(1,0), c3;
c3 = c1 + c2;
```

than

```
c3 = AddComplex(c1, c2);
```

The operator **+** is easily overloaded by including the following declaration in the class *complex*:

```
friend complex operator +(complex c1, complex c2);
```

and defining it (possibly inline) as:

```
complex operator +(complex c1, complex c2)
{
    return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2); // same as c3 = c1 + c2
```

Apart from **new** and **delete**, which have their own rules (see the next sections), an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions **=**, **()**, **[]** and **->** must be nonstatic member functions.

Overloaded operators and inheritance

With the exception of the assignment function **operator=()** (see “The assignment **operator=()**” on page 484), all overloaded operator functions for **class X** are inherited by classes derived from **X**, with the standard resolution rules for overloaded functions. If **X** is a base class for **Y**, an overloaded operator function for **X** may possibly be further overloaded for **Y**.

Operators **new** and **delete**

The type **size_t** is defined in *stdlib.h*

The operators **new** and **delete** can be overloaded to provide alternative free storage (heap) memory-management routines. A user-defined operator **new** must return a **void*** and must have a **size_t** as its first argument. A user-defined operator **delete** must have a **void** return type and **void*** as its first argument; a second argument of type **size_t** is optional. For example,

```
#include <stdlib.h>

class X {
    ...
public:
    void* operator new(size_t size) { return newalloc(size); }
    void operator delete(void* p) { newfree(p); }
    X() { /* initialize here */ }
    X(char ch) { /* and here */ }

    ~X() { /* clean up here */ }
    ...
};
```

The *size* argument gives the size of the object being created, and **newalloc()** and **newfree()** are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of **class X** (or objects of classes derived from **X** that do not have their own overloaded operators **new** and **delete**) will invoke

the matching user-defined **X::operator new()** and **X::operator delete()**, respectively.

The **X::operator new** and **X::operator delete** operator functions are static members of **X** whether explicitly declared as **static** or not, so they cannot be virtual functions.

The standard, predefined (global) **new** and **delete** operators can still be used within the scope of **X**, either explicitly with the global scope operator (**::operator new** and **::operator delete**), or implicitly when creating and destroying non-**X** or non-**X**-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
    void* ptr = new char[s]; // standard new called
    ...
    return ptr;
}

void X::operator delete(void* ptr)
{
    ...
    delete (void*) ptr; // standard delete called
}
```

The reason for the *size* argument is that classes derived from **X** inherit the **X::operator new**. The size of a derived class object may well differ from that of the base class.

Unary operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a non-member function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either **x.operator@()** or **operator@(x)**, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.



Under C++ 2.0, an overloaded operator**++** or **--** is used for both prefix and postfix uses of the operator. For example:

```
struct foo
{
    operator++();
    operator--();
}
```

```

    x;

void func()
{
    x++; // calls x.operator++()
    ++x; // calls x.operator++()

    x--; // calls x.operator--()
    --x; // calls x.operator--()
}

```

With C++ 2.1, when an **operator++** or **operator--** is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix **operator++** or **operator--**. You can only overload a postfix **operator++** or **operator--** by defining it as a member function taking an **int** parameter or as a nonmember function taking one class and one **int** parameter. For example add the following lines to the previous code:

```

operator++(int);
operator--(int);

```

When only the prefix version of an **operator++** or **operator--** is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```

Warning: Overloaded prefix 'operator ++' used as a postfix operator
in function func()

Warning: Overloaded prefix 'operator --' used as a postfix operator
in function func()

```

Binary operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually **friend**) taking two arguments. If **@** represents a binary operator, $x@y$ can be interpreted as either $x.\mathbf{operator}@ (y)$ or $\mathbf{operator}@ (x,y)$, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

The assignment **operator=()**

The assignment **operator=()** can be overloaded by declaring a nonstatic member function. For example,

```
class String {
    ...
    String& operator = (String& str);
    ...
    String (String&);
    ~String();
}
```

This code, with suitable definitions of **String::operator=()**, allows string assignments *str1 = str2*, just like other languages. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any **class X**, there is no user-defined operator **=**, the operator **=** is defined by default as a member-by-member assignment of the members of **class X**:

```
X& X::operator = (const X& source)
{
    // memberwise assignment
}
```

The function call **operator()**

The function call

primary-expression (<*expression-list*>)

is considered a binary operator with operands *primary-expression* and *expression-list* (possibly empty). The corresponding operator function is **operator()**. This function can be user-defined for a **class X** (and any derived classes) only by means of a nonstatic member function. A call **x(arg1, arg2)**, where **x** is an object of **class X**, is interpreted as **x.operator()(arg1, arg2)**.

The subscript operator

Similarly, the subscripting operation

primary-expression [*expression*]

is considered a binary operator with operands *primary-expression* and *expression*. The corresponding operator function is **operator[]**; this can be user-defined for a **class X** (and any derived classes)

only by means of a nonstatic member function. The expression `x[y]`, where `x` is an object of **class X**, is interpreted as `x.operator[](y)`.

The class member access operator

Class member access using

primary-expression -> expression

is considered a unary operator. The function `operator->` must be a nonstatic member function. The expression `x->m`, where `x` is a **class X** object, is interpreted as `(x.operator->())->m`, so that the function `operator->()` must either return a pointer to a class object or return an object of a class for which `operator->` is defined.

virtual functions

virtual functions can only be member functions.

Virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. The redefined function is said to *override* the base class function. You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not **virtual**. The base class version is available to derived class objects via scope override. If they are **virtual**, only the function associated with the actual type of the object is available.

With **virtual** functions, you cannot change just the function type. It is illegal, therefore, to redefine a virtual function so that it differs only in the return type. If two functions with the same name have different arguments, C++ considers them different, and the **virtual** function mechanism is ignored.

If a base **class B** contains a **virtual** function `vf()`, and **class D**, derived from **B**, contains a function `vf()` of the same type, then if `vf()` is called for an object `d` or **D**, the call made is `D::vf()`, even if the access is via a pointer or reference to **B**. For example,

```
struct B {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();
};
```



```

};
class D : public B {
    virtual void vf1(); // virtual specifier is legal but redundant
    void vf2(int);     // not virtual, since it's using a different
                        // arg list
    char vf3();       // Illegal: return-type-only change!
    void f();
};

void extf()
{
    D d;           // declare a D object
    B* bp = &d;   // standard conversion from D* to B*
    bp->vf1();     // calls D::vf1
    bp->vf2();     // call B::vf2 since D's vf2 has different args
    bp->f();       // calls B::f (not virtual)
};

```

The overriding function **vf1()** in **D** is automatically **virtual**. The **virtual** specifier *can* be used with an overriding function declaration in the derived class, but its use is redundant.

The interpretation of a **virtual** function call depends on the type of the object for which it is called; with **non-virtual** function calls, the interpretation depends only on the type of the pointer or reference denoting the object for which it is called.



virtual functions must be members of some class, but they cannot be static members. A **virtual** function can be a **friend** of another class.

A **virtual** function in a base class, like all member functions of a base class, must be defined or, if not defined, declared *pure*:

```

class B {
    virtual void vf(int) = 0;    // = 0 means 'pure'
};

```

In a class derived from such a base class, each pure function must be defined or redeclared as pure (see the section, "Abstract classes").

If a **virtual** function is defined in the base it need not necessarily be redefined in the derived class. Calls will simply call the base function.

virtual functions exact a price for their versatility: Each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

Abstract classes

An *abstract class* is a class with at least one pure virtual function. A virtual function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {           // abstract class
    point center;
    ...
public:
    where() { return center; }
    move(point p) { center = p; draw(); }
    virtual void rotate(int) = 0; // pure virtual function
    virtual void draw() = 0;      // pure virtual function
    virtual void hilite() = 0;    // pure virtual function
    ...
}

shape x;                // ERROR: attempted creation of an object of
                        // an abstract class

    shape* sptr;        // pointer to abstract class is OK
    shape f();          // ERROR: abstract class cannot be a return
                        // type
int g(shape s);         // ERROR: abstract class cannot be a
                        //function argument type
shape& h(shape&);       // reference to abstract class as return
                        // value or function argument is OK
```

Suppose that **D** is a derived class with the abstract **class B** as its immediate base class. Then for each pure virtual function **pvf()** in **B**, if **D** doesn't provide a definition for **pvf()**, **pvf()** becomes a pure member function of **D**, and **D** will also be an abstract class.

For example, using the **class shape** previously outlined,

```
class circle : public shape { // circle derived from
    // abstract class
    int radius;                // private
public:
    void rotate(int) { }        // virtual function defined:
                                // no action to rotate a
                                // circle
```

```
void draw();           // circle::draw must be
                      // defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

C++ scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement may appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

Class scope

The name *M* of a member of a **class X** has class scope “local to X;” it can only be used in the following situations:

- In member functions of **X**
- In expressions such as **x.M**, where **x** is an object of **X**
- In expressions such as *xptr->M*, where *xptr* is a pointer to an object of **X**
- In expressions such as **X::M** or **D::M**, where **D** is a derived class of **X**
- In forward references within the class of which it is a member.

Names of functions declared as friends of **X** are not members of **X**; their names simply have enclosing scope.

Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: **X::M**. A hidden file scope (global) name can be referenced with the unary operator **::**; for example, **::g**. A class name **X** can be hidden by the name of an object, function, or enumerator declared within the scope of **X**, regardless of the order in which the names are declared. However, the hidden class name **X** can still be accessed

by prefixing **X** with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name *x* is immediately after its complete declaration but before its initializer, if one exists.

C++ scoping rules summary

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

1. The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
2. If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
3. If the name is used outside any function and class, or is prefixed by the unary scope access operator `::`, and if the name is not qualified by the binary `::` operator or the member selection operators `.` and `->`, then the name must be a global object, function, or enumerator.
4. If the name *n* appears in any of the forms **X**::*n*, *x*.*n* (where *x* is an object of **X** or a reference to **X**), or *ptr*->*n* (where *ptr* is a pointer to **X**), then *n* is the name of a member of **X** or the member of a class from which **X** is derived.
5. Any name not covered so far that is used in a static member function must be declared in the block in which it occurs or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.
6. Any name not covered so far that is used in a nonstatic member function of class **X** must be declared in the block in which it occurs or in an enclosing block, be a member of class **X** or a base class of **X**, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
7. The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a non-defining function declaration has no scope at all. The scope of a default argument is determined

by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.

8. A constructor initializer (see *ctor-initializer* in the class declarator syntax, Table 12.3 on page 381) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Templates

For a discussion of templates in the **container** class library see the online documentation file `CONTAIN.DOC`.

Templates, also called *generics* or *parameterized types*, allow you to construct a family of related functions or classes. In this section, we'll introduce the basic concept then some specific points.

Syntax:

Template-declaration:

template < *template-argument-list* > *declaration*

template-argument-list:

template-argument

template-argument-list, *template argument*

template-argument:

type-argument

argument-declaration

type-argument:

class identifier

template-class-name:

template-name < *template-arg-list* >

template-arg-list:

template-arg

template-arg-list, *template-arg*

template-arg:

expression

type-name

Function templates

Consider a function **max(x,y)** that returns the larger of its two arguments. *x* and *y* can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects

the types of the parameters *x* and *y* to be declared at compile time. Without using templates, many overloaded versions of **max()** are required, one for each data type to be supported, even though the code for each version is essentially identical. Each version compares the arguments and returns the larger. For example,

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}

long max(long x, long y)
{
    return (x > y) ? x : y;
}
...
```

followed by other versions of **max()**.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the **#define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of **max(x,y)** to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be:

```
class Foo
{
public:
    int max(int, int); // Results in syntax error;
                       // this gets expanded!!!
    // ...
};
```

By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

Function template definition

```
template <class T> T max(T x, T y)
{
    return (x > y) ? x : y;
};
```

The data type is represented by the template argument: **<class T>**. When used in an application, the compiler generates the appropriate function according to the data type actually used in the call:

```
int i;
Myclass a, b;

int j = max(i,0);    // arguments are integers
Myclass m = max(a,b); // arguments are type Myclass
```



Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator()**, so you can use **max()** with arguments of any type for which **operator()** is defined.

Overriding a template function

The previous example is called a *function template* (or *generic function*, if you like). A specific instantiation of a function template is called a *template function*. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>

char *max(char *x, char *y)
{
    return(strcmp(x,y)>0) ?x:y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit and explicit template functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b)
{
```

```

        return (a > b) ? a : b;
    }
    void f(int i, char c)
    {
        max(i, i);           // calls max(int ,int )
        max(c, c);          // calls max(char,char)
        max(i, c);          // no match for max(int,char)
        max(c, i);          // no match for max(char,int)
    }

```

This code results in the following error messages.

Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)

If the user explicitly declares a template function, this function, on the other hand, will participate fully in overload resolution. For example:

```

template<class T> T max(T a, T b)
{
    return (a > b) ? a : b;
}
Explicit template function. int max(int,int);           // declare max(int,int) explicitly
void f(int i, char c)
{
    max(i, i);           // calls max(int ,int )
    max(c, c);          // calls max(char,char)
    max(i, c);          // calls max(int,int)
    max(c, i);          // calls max(int,int)
}

```

Class templates

A class template (also called a *generic class* or *class generator*) allows you to define a pattern for class definitions. Generic container classes are good examples. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a *type* parameter to the class, the system will generate type-safe class definitions on the fly:

```

Class template definition
#include <iostream.h>
template <class T> class Vector
{
    T *data;

```



```

    int size;

public:
    Vector(int);
    ~Vector() {delete[] data;}
    T& operator[](int i) {return data[i];}
};

    // Note the syntax for out-of-line definitions:
template <class T> Vector<T>::Vector(int n)
{
    data = new T[n];
    size = n;
};

main()
{
    Vector<int> x(5); // Generate a vector of ints

    for (int i = 0; i < 5; ++i)
        x[i] = i;
    for (i = 0; i < 5; ++i)
        cout << x[i] << ' ';
    cout << '\n';
    return 0;
}

// Output will be: 0 1 2 3 4

```

As with function templates, an explicit *template class* definition may be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol **Vector** must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

For a more complete implementation of a vector class, see the file vectimp.h in the container class library source code, found in the \TC\CLASSLIB\INCLUDE subdirectory. Also see online documentation, "CONTAIN.DOC."

Arguments

Although these examples use only one template argument, multiple arguments are allowed. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Non-type template arguments such as *size* can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;

Buffer<int, 2*N> b1; // OK
Buffer<float, i> b2; // Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Angle brackets Take care when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the `>` between `x` and `100` would prematurely close the template argument list.

Type-safe generic lists In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
public:
    void insert( void * );
    void *peek();
    // ...
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList
{
public:
    void insert( Foo *f ) { GList::insert( f ); }
    Foo *peek() { return (Foo *)GList::peek(); }
    // ...
};
```

This is type-safe. **insert()** will only take arguments of type pointer-to-**Foo** or object-derived-from-**Foo**, so the underlying container will only hold pointers that in fact point to something of

type **Foo**. This means that the cast in **FooList::peek()** is always safe, and you've created a true **FooList**. Now to do the same thing for a **BarList**, a **BazList**, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and be type-safe, once again, use templates:

Type-safe generic list class definition

```
template <class T> class List : public GList
{
public:
    void insert( T *t ) { GList::insert( t ); }
    T *peek() { return (T *)GList::peek(); }
    // ...
};

List<Foo> fList; // create a FooList class and an instance
                // named fList.
List<Bar> bList; // create a BarList class and an instance
                // named bList.
List<Baz> zList; // create a BazList class and an instance
                // named zList.
```

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no run-time overhead imposed by this type safety.

Eliminating pointers

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is a big benefit if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

Template definition that eliminates pointers

```
template <class T> aBase
{
    // ...
private:
    T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf>
{
    // ...
};
```

All the functions in **aBase** can call functions defined in **aFilebuf** directly, without having to go through a pointer. And if any of the

functions in **aFilebuf** can be inlined, you'll get a speed improvement, since templates allow them to be inlined.

Template compiler switches



The **-Jg** family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler. For template functions the switch applies to the function instances; for template classes, it will apply to all member functions and static data members of the template class. In all cases this switch applies only to compiler-generated template instances, and never to user-defined instances, although it can be used to tell the compiler which instances will be user-defined so that they are not generated from the template.

- Jg** Default value of the switch. All template instances first encountered when this switch value is in effect will be generated, such that if several compilation units generate the same template instance, the linker will merge them to produce a single copy of the instance. This is the most convenient approach to generating template instances, because it's almost entirely automatic. Note, though, that in order to be able to generate the template instances, the compiler must have the function body (in case of a template function) or bodies of member functions and definitions for static data members (in case of a template class).
- Jgd** Instructs the compiler to generate public definitions for template instances. This is similar to **-Jg**, but if more than one compilation unit generates a definition for the same template instance, the linker will report public symbol re-definition errors.
- Jgx** Instructs the compiler to generate external references to template instances. Some other compilation unit must generate a public definition for that template instance (using the **-Jgd** switch) so that the external references can be satisfied.

Using template switches

When using the **-Jg** family of switches, there are two basic approaches for generating template instances:

1. Include the function body (for a function template) or member function and static data member definitions (for a template class) in the header file that defines the particular template, and use the default setting of the template switch (**-Jg**). If some instances of the template are user-defined, the declarations (prototypes, for example) for them should be included in the same header, but preceded by **#pragma option -Jgx**, thus letting the compiler know that it should not generate those particular instances.

Here's an example of a template function header file:

```
// Declare a template function along with its body
template<class T> void sort(T* array, int size)
{
    ... body of template function goes here ...
}

// Sorting of 'int' elements done by user-defined instance
#pragma option -Jgx
extern void sort(int* array, int size);
// Restore the template switch to its original state
#pragma option -Jg.
```

If the preceding header file is included in a C++ source file, the **sort()** template can be used without worrying about how the various instances are generated (with the exception of **sort()** for **int** arrays, which is declared as a user-defined instance, and whose definition must be defined by the user).

2. Compile all of the source files comprising the program with the **-Jgx** switch (causing external references to templates to be generated); this way, template bodies don't need to appear in header files. In order to provide the definitions for all of the template instances, add a file (or files) to the program that includes the template bodies (including any user-defined instance definitions), and list all the template instances needed in the rest of the program, to provide the necessary public symbol definitions. Compile the file (or files) with the **-Jgd** switch.

Here's an example:

```

// vector.h
template <class elem, int size> class vector
{
    elem * value;
public:
    vector();
    elem & operator[](int index) { return value[index]; }
};

// MAIN.CPP
#include "vector.h"
// Tell the compiler that the template instances that follow
// will be defined elsewhere.
#pragma option -Jgx
// Use two instances of the 'vector' template class.
vector<int,100> int_100;
vector<char,10> char_10;
main()
{
    return int_100[0] + char_10[0];
}

// TEMPLATE.CPP
#include <string.h>
#include "vector.h"
// Define any template bodies
template <class elem, int size> vector<elem, size>::vector()
{
    value = new elem[size];
    memset(value, 0, size * sizeof(elem));
}

// Generate the necessary instances
#pragma option -Jgd
typedef vector<int,100> fake_int_100;
typedef vector<char,10> fake_char_10;

```


The preprocessor

Although Turbo C++ uses an integrated single-pass compiler for its IDE and command-line versions, it is useful to retain the terminology associated with earlier multipass compilers.

With a multipass compiler, a first pass of the source text would pull in any include files, test for any conditional-compilation directives, expand any macros, and produce an intermediate file for further compiler passes. Since the IDE and command-line versions of the Turbo C++ compiler perform this first pass with no intermediate output, Turbo C++ provides an independent preprocessor, CPP.EXE, that does produce such an output file. The independent preprocessor is useful as a debugging aid, letting you see the net result of include directives, conditional compilation directives, and complex macro expansions.

The following discussion on preprocessor directives, their syntax and semantics, therefore, applies both to the CPP preprocessor and to the preprocessor functionality built into the Turbo C++ compiler.

The Turbo C++ preprocessor includes a sophisticated macro processor that scans your source code before the compiler itself gets to work. The preprocessor gives you great power and flexibility in the following areas:

- Defining macros that reduce programming effort and improve your source code legibility. Some macros can also eliminate the overhead of function calls.

The independent preprocessor is documented online.

The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them.

- Including text from other files, such as header files containing standard library and user-supplied function prototypes and manifest constants.
- Setting up conditional compilations for improved portability and for debugging sessions.

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program.

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

The full syntax for Turbo C++'s preprocessor directives is given in the next table.

Table 14.1: Turbo C++ preprocessing directives syntax

<i>preprocessing-file:</i> group	#pragma warn <i>action abbreviation newline</i> #pragma inline <i>newline</i> # <i>newline</i>
<i>group:</i> group-part group group-part	<i>action:</i> one of + -
<i>group-part:</i> <pp-tokens> <i>newline</i> <i>if-section</i> <i>control-line</i>	<i>abbreviation:</i> nondigit nondigit nondigit
<i>if-section:</i> if-group <elif-groups> <else-group> <i>endif-line</i>	<i>lparen:</i> the left parenthesis character without preceding whitespace
<i>if-group:</i> #if <i>constant-expression</i> <i>newline</i> <group> #ifdef <i>identifier</i> <i>newline</i> <group> #ifndef <i>identifier</i> <i>newline</i> <group>	<i>replacement-list:</i> <pp-tokens>
<i>elif-groups:</i> elif-group elif-groups <i>elif-group</i>	<i>pp-tokens:</i> preprocessing-token pp-tokens preprocessing-token
<i>elif-group:</i> #elif <i>constant-expression</i> <i>newline</i> <group>	<i>preprocessing-token:</i> header-name (only within an #include directive) identifier (no keyword distinction) constant string-literal operator punctuator each non-whitespace character that cannot be one of the preceding
<i>else-group:</i> #else <i>newline</i> <group>	<i>header-name:</i> <h-char-sequence>
<i>endif-line:</i> #endif <i>newline</i>	<i>h-char-sequence:</i> h-char h-char-sequence h-char
<i>control-line:</i> #include <i>pp-tokens</i> <i>newline</i> #define <i>identifier</i> <i>replacement-list</i> <i>newline</i> #define <i>identifier</i> <i>lparen</i> < <i>identifier-list</i> > <i>replacement-list</i> <i>newline</i> #undef <i>identifier</i> <i>newline</i> #line <i>pp-tokens</i> <i>newline</i> #error < <i>pp-tokens</i> > <i>newline</i> #pragma < <i>pp-tokens</i> > <i>newline</i>	<i>h-char:</i> any character in the source character set except the newline (\n) or greater than (>) character
	<i>newline:</i> the <i>newline</i> character

Null directive

The null directive consists of a line containing the single character `#`. This directive is always ignored.

The `#define` and `#undef` directives

The `#define` directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Simple `#define` macros

In the simple case with no parameters, the syntax is as follows:

```
#define macro_identifier <token_sequence>
```

Each occurrence of *macro_identifier* in your source code following this control line will be replaced *in situ* with the possibly empty *token_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.

An empty token sequence results in the effective removal of each affected macro identifier from the source code:

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
...
puts(HI);           /* expands to puts("Have a nice day!"); */
puts(NIL);          /* expands to puts(""); */
puts("empty");      /* NO expansion of empty! */
/* NOR any expansion of the empty within comments! */
```

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: The expanded text may contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor:

```
#define GETSTD #include <stdio.h>
...
GETSTD /* compiler error */
```

GETSTD will expand to #include <stdio.h>. However, the preprocessor itself will not obey this apparently legal directive, but will pass it verbatim to the compiler. The compiler will reject #include <stdio.h> as illegal input. A macro won't be expanded during its own expansion. So #define A A won't expand indefinitely.

The #undef directive

You can undefine a macro using the **#undef** directive:

```
#undef macro_identifier
```

This line detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined.

No macro expansion occurs within **#undef** lines.

The state of being *defined* or *undefined* turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

```
#define BLOCK_SIZE 512
...
buff = BLOCK_SIZE*blks; /* expands as 512*blks */
...
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
...
#define BLOCK_SIZE 128 /* redefinition */
...
buff = BLOCK_SIZE*blks; /* expands as 128*blks */
...
...
```

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is *exactly* the same, token-by-token definition as the existing one. The preferred strategy where definitions may exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single space character.

Assembly language programmers must resist the temptation to write:

```
#define BLOCK_SIZE = 512 /* ?? token sequence includes the = */
```

The `-D` and `-U` options

Identifiers can be defined and undefined using the command-line compiler options `-D` and `-U` (see Chapter 8, “The command-line compiler,”). Identifiers can be defined, but not explicitly undefined, from the IDE Options | Compiler | Code Generation dialog box (see Chapter 2, “IDE basics,”).

The command line

```
TCC -Ddebug=1; paradox=0; X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
#define X
#undef mysym
```

in the program.

The Define option

Identifiers can be defined, but not explicitly undefined, from the Defines input box in the Code Generation | Options dialog box (under O | C | Code Generation) (see Chapter 2, “IDE basics,”).

Keywords and protected words

It is legal but ill-advised to use Turbo C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long   /* legal and possibly useful */
```

The following predefined global identifiers may *not* appear immediately following a **#define** or **#undef** directive:

Note the double underscores, leading and trailing.

```
__STDC__          __DATE__
__FILE__          __TIME__
__LINE__
```

Macros with parameters

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Note that there can be no whitespace between the macro identifier and the **(**. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *place holder*.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences and potential pitfalls (see page 508).

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the **#define** line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*. For example,

```

#define CUBE(x) ((x)*(x)*(x))
...
int n,y;
n = CUBE(y);

```

results in the following replacement:

$$n = ((y) * (y) * (y));$$

Similarly, the last line of

```

#define SUM (a,b) ((a) + (b))
...
int i,j,sum;
sum = SUM(i,j);

```

expands to $sum = ((i) + (j))$. The reason for the apparent glut of parentheses will be clear if you consider the call

```
n = CUBE(y+1);
```

Without the inner parentheses in the definition, this would expand as $n = y+1*y+1 *y+1$, which is parsed as

```
n = y + (1*y) + (1*y) + 1; // != (y+1) cubed unless y=0 or y = -3!
```

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Note the following points when using macros with argument lists:

1. **Nested parentheses and commas:** The *actual_arg_list* may contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters:

```

#define ERRMSG(x, str) showerr("Error",x,str)
#define SUM(x,y) ((x) + (y))
...
ERRMSG(2, "Press Enter, then Esc");
/* expands to showerr("Error",2,"Press Enter, then Esc");
return SUM(f(i,j), g(k,l));
/* expands to return ((f(i,j)) + (g(k,l))); */

```

2. **Token pasting with ##:** You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers; for example, given the definition

```
#define VAR(i,j) (i##j)
```

then the call `VAR(x,6)` would expand to `(x6)`. This replaces the older (nonportable) method of using `(i/**/j)`.

3. **Converting to strings with #:** The `#` symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement. So, given the following macro definition:

```
#define TRACE(flag) printf(#flag "=%d\n",flag)
```

the code fragment

```
int highval = 1024;
TRACE(highval);
```

becomes

```
int highval = 1024;
printf("highval" " = %d\n", highval);
```

which, in turn, is treated as

```
int highval = 1024;
printf("highval=%d\n", highval);
```

4. **The backslash for line continuation:** A long token sequence can straddle a line by using a backslash (`\`). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions:

```
#define WARN "This is really a single-\
line warning"
...
puts(WARN);
/* screen will show: This is really a single-line warning */
```

5. **Side effects and other dangers:** The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once. Compare **CUBE** and **cube** in the following example:

```
int cube(int x) {
    return x*x*x;
}
#define CUBE(x) ((x)*(x)*(x))
...
int b = 0, a = 3;
b = cube(a++);
/* cube() is passed actual arg = 3; so b = 27; a now = 4 */
```

Final value of *b* depends on what your compiler does to the expanded expression.

```
a = 3;  
b = CUBE(a++);  
/* expands as ((a++)*(a++)*(a++)); a now = 6 */
```

File inclusion with **#include**

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three forms:

The angle brackets are real tokens, not metasympols that imply that *header_name* is optional.

```
#include <header_name>  
#include "header_name"  
#include macro_identifier
```

The third variant assumes that neither **<** nor **"** appears as the first non-whitespace character following **#include**; further, it assumes that a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the *<header_name>* or *"header_name"* formats.

The first and second variant imply that no macro expansion will be attempted; in other words, *header_name* is never scanned for macro identifiers. *header_name* must be a valid DOS file name with an extension (traditionally .h for header) and optional path name and path delimiters.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** may therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header_name*, only that directory will be searched.

The difference between the *<header_name>* and *"header_name"* formats lies in the searching algorithm employed in trying to locate the include file; these algorithms are described in the following two sections.

Header file
search with
<header_name>

The <header_name> variant specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header file
search with
"header_name"

The "header_name" variant specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the include directories as in the <header_name> situation.

The following example clarifies these differences:

```
#include <stdio.h>
/* header in standard include directory */

#define myinclud "C:\TC\INCLUDE\MYSTUFF.H"
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\TC\INCLUDE\MYSTUFF.H" */

#include myinclud
/* macro expansion */

#include "myinclud.h"
/* no macro expansion */
```

After expansion, the second **#include** statement causes the preprocessor to look in C:\TC\INCLUDE\MYSTUFF.H and nowhere else. The third **#include** causes it to look for MYINCLUD.H in the current directory, then in the default directories.

Conditional compilation

Turbo C++ supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The **#if**, **#elif**, **#else**, and **#endif** conditional directives

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. They are used as follows:

```
#if constant-expression-1  
  <section-1>  
  <#elif constant-expression-2 newline section-2>  
  ...  
  <#elif constant-expression-n newline section-n>  
  <#else <newline> final-section>  
#endif  
  ...
```

If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the Turbo C++ compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional interlude) and continues with *next-section*. In the *false* case, control passes to the next **#elif** line (if any) where *constant-expression-2* is evaluated. If true, *section-2* is processed, after which control moves on to the matching **#endif**. Otherwise, if *constant-expression-2* is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be carefully balanced with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

The operator **defined** The **defined** operator offers an alternative, more flexible way of testing whether combinations of identifiers are defined or not. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) or **defined** *identifier* (parentheses are optional) evaluates to 1 (true) if the symbol has been previously defined (using **#define**) and has not been subsequently undefined (using **#undef**); otherwise, it evaluates to 0 (false). So the directive

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use **defined** repeatedly in a complex expression following the **#if** directive, such as

```
#if defined(mysym) && !defined(yoursym)
```

The **#ifdef** and **#ifndef** conditional directives

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not, that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

#ifndef tests true for the “not-defined” condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif** given in the previous section.

An identifier defined as NULL is considered to be defined.

The **#line** line control directive

You can use the **#line** command to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program. The syntax is

```
#line integer_constant <"filename">
```

indicating that the following source line originally came from line number *integer_constant* of *filename*. Once the *filename* has been registered, subsequent **#line** commands relating to that file can omit the explicit *filename* argument.

The inclusion of `stdio.h` means that the preprocessor output will be somewhat large.

```
/* TEMP.C: An example of the #line directive */
#include <stdio.h>
#line 4 "junk.c"
void main()
{
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 12 "temp.c"
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 8
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
}
```

If you run TEMP.C through CPP (`cpp temp.c`), you'll get an output file TEMP.I; it should look something like this:

```
temp.c 1:
C:\TC\INCLUDE\STDIO.H 1:
C:\TC\INCLUDE\STDIO.H 2:
C:\TC\INCLUDE\STDIO.H 3:
...
C:\TC\INCLUDE\STDIO.H 212:
C:\TC\INCLUDE\STDIO.H 213:
```

We've eliminated most of the `stdio.h` portion.

```

temp.c 2:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6: printf(" in line %d of %s",6,"junk.c");
junk.c 7:
temp.c 12: printf("\n");
temp.c 13: printf(" in line %d of %s",13,"temp.c");
temp.c 14:
temp.c 8: printf("\n");
temp.c 9: printf(" in line %d of %s",9,"temp.c");
temp.c 10: }
temp.c 11:

```

If you then compile and run TEMP.C, you'll get the output shown here:

```

in line 6 of junk.c
in line 13 of temp.c
in line 9 of temp.c

```

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

The **#error** directive

The **#error** directive has the following syntax:

```
#error errmsg
```

This generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional that is true for the undesired case.

For example, suppose you **#define** *MYVAL*, which must be either 0 or 1. You could then include the following conditional in your source code to test for an incorrect value of *MYVAL*:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

The #pragma directive

The **#pragma** directive permits implementation-specific directives of the form:

#pragma *directive-name*

With **#pragma**, Turbo C++ can define whatever directives it desires without interfering with other compilers that support **#pragma**. If the compiler doesn't recognize *directive-name*, it ignores the **#pragma** directive without any error or warning message.

Turbo C++ supports the following **#pragma** directives:

- **#pragma argsused**
- **#pragma exit**
- **#pragma hdrfile**
- **#pragma hdrstop**
- **#pragma inline**
- **#pragma option**
- **#pragma saveregs**
- **#pragma startup**
- **#pragma warn**

Turbo C++ only

#pragma argsused

The **argsused** pragma is only allowed between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

#pragma exit and #pragma startup

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the main function is called), or program exit (just before the program terminates through **_exit**).

The syntax is as follows:

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

The specified *function-name* must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as

```
void func(void);
```

Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100. For example,

```
#include <stdio.h>

void startFunc(void)
{
    printf("Startup function.\n");
}

#pragma startup startFunc 64
/* priority 64 --> called first at startup */

void exitFunc(void)
{
    printf("Wrapping up execution.\n");
}

#pragma exit exitFunc
/* default priority is 100 */

void main(void)
{
    printf("This is main.\n");
}
```

*Note that the function name used in **pragma startup** or **exit** must be defined (or declared) before the pragma line is reached.*

#pragma hdrfile

This directive sets the name of the file in which to store precompiled headers. The default file name is TCDEF.SYM. The syntax is

```
#pragma hdrfile "filename.SYM"
```

See Appendix B, "Precompiled headers".

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option **-H=filename** or Precompiled Header (O|C|Code Generation) to change the name of the file used to store precompiled headers.

#pragma hdrstop

This directive terminates the list of header files that are eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers. (See Appendix B for more on precompiled headers.)

#pragma inline

This directive is equivalent to the **-B** command-line compiler option or the IDE inline option. It tells the compiler that there is inline assembly language code in your program (see BASM.DOC). The syntax is

#pragma inline

This is best placed at the top of the file, since the compiler restarts itself with the **-B** option when it encounters **#pragma inline**.

#pragma option

Use **#pragma option** to include command-line options within your program code. The syntax is

#pragma option [*options...*]

The command-line compiler options are defined in Chapter 8.

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, then return it to its default, without you having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include

-B	-H	-Q
-c	-Ifilename	-S
-dname	-Lfilename	-T
-Dname = string	-lxset	-Uname
-efilename	-M	-V
-E	-o	-X
-Fx	-P	-Y

You can use **#pragmas**, **#includes**, **#define**, and some **#ifs** before

1. The use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive.
2. The occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can *only* appear in a **#pragma option** command before these events. These options are

-Efilename	-m*	-u
-f*	-npath	-z*
-i#	-ofilename	

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

-1	-h	-r
-2	-k	-rd
-a	-N	-v
-ff	-O	-y
-G	-p	-Z

The following options can be changed at any time and take effect immediately:

See page 584 for more on using **#pragma option** with far objects.

-A	-gn	-zE
-b	-jn	-zF
-C	-K	-zH
-d	-wxxx	

They can additionally appear followed by a dot (.) to reset the option to its command-line state.

#pragma saveregs

The **saveregs** pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

#pragma warn

The **warn** directive lets you override specific **-wxxx** command-line options or check **Display Warnings** settings in the **Options | Compiler | Messages** dialog boxes.

For example, if your source code contains the directives

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned on (even if on the **Options | Compiler | Messages** menu it was toggled to *Off*), the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began.

A complete list of the three-letter abbreviations and the warnings to which they apply is given in Chapter 8, “The command-line compiler”.

Predefined macros

Turbo C++ predefines certain global identifiers, each of which is discussed in this section. Except for **__cplusplus** each of the global identifiers starts and ends with two underscore characters “__”. These macros are also known as *manifest constants*.

__CDECL__

This macro is specific to Borland’s C and C++ family of compilers. It signals that the **-p** flag was not used (the **C** radio button in the **Entry/Exit Code Generation** dialog box). Set to the integer constant 1 if calling was not used; otherwise, undefined.

The following six symbols are defined based on the memory model chosen at compile time.

```
__COMPACT__      __MEDIUM__
__HUGE__         __SMALL__
__LARGE__        __TINY__
```

Only one is defined for any given compilation; the others, by definition, are undefined. For example, if you compile with the small

model, the `__SMALL__` macro is defined and the rest are not, so that the directive

```
#if defined(__SMALL__)
```

will be true, while

```
#if defined(__LARGE__)
```

(or any of the others) will be false. The actual value for any of these defined macros is 1.

`__cplusplus`

This macro is defined as 1 if in C++ mode; it's undefined otherwise. This allows you to write a module that will be compiled sometimes as C and sometimes as C++. Using conditional compilation, you can control which C and C++ parts are included.

`__DATE__`

This macro provides the date the preprocessor began processing the current source file (as a string literal). Each inclusion of `__DATE__` in a given file contains the same value, regardless of how long the processing takes. The date appears in the format *mmm dd yyyy*, where *mmm* equals the month (Jan, Feb, and so forth), *dd* equals the day (1 to 31, with the first character of *dd* a blank if the value is less than 10), and *yyyy* equals the year (1990, 1991, and so forth).

`__FILE__`

This macro provides the name of the current source file being processed (as a string literal). This macro changes whenever the compiler processes an `#include` directive or a `#line` directive, or when the include file is complete.

`__LINE__`

This macro provides the number of the current source-file line being processed (as a decimal constant). Normally, the first line of a source file is defined to be 1, through the `#line` directive can affect this. See page 513 for information on the `#line` directive.

__MSDOS__

This macro is specific to Borland's C/C++ family of compilers. It provides the integer constant 1 for all compilations.

__OVERLAY__

This macro is specific to Borland's C and C++ family of compilers. It is predefined to be 1 if you compile a module with the **-Y** option (enable overlay support). If you don't enable overlay support, this macro is undefined.

__PASCAL__

This macro is specific to Borland's C and C++ family of compilers. It signals that the **-p** flag or the Pascal calling convention (O|C|C|Exit/Entry) has been used. The macro is set to the integer constant 1 if used; otherwise, it remains undefined.

__STDC__

This macro is defined as the constant 1 if you compile with the ANSI compatibility flag (**-A**) or ANSI radio button (Source Options); otherwise, the macro is undefined.

__TCPLUSPLUS__

This macro is specific to Borland's C and C++ family of compilers. It is only defined for C++ compilation. If you've selected C++ compilation, it is defined as 0x300, a hexadecimal constant. This numeric value will increase in later releases.

__TEMPLATES__

This macro is specific to Borland's C and C++ family of compilers. It is defined as 1 for C++ files (meaning that Turbo C++ supports templates); it's undefined otherwise.

__TIME__

This macro keeps track of the time the preprocessor began processing the current source file (as a string literal).

As with `__DATE__`, each inclusion of `__TIME__` contains the same value, regardless of how long the processing takes. It takes the format *hh:mm:ss*, where *hh* equals the hour (00 to 23), *mm* equals minutes (00 to 59), and *ss* equals seconds (00 to 59).

`__TURBOC__`

This macro is specific to Borland's C and C++ family of compilers. It is defined as `0x400`, a hexadecimal constant. This numeric value will increase in later releases.

The main function

Every C and C++ program must have a **main** function; where you place it is a matter of preference. Some programmers place **main** at the beginning of the file, others at the end. Regardless of its location, the following points about **main** always apply.

Arguments to main

Three parameters (arguments) are passed to **main** by the Turbo C++ startup routine: *argc*, *argv*, and *env*.

- *argc*, an integer, is the number of command-line arguments passed to **main**.
- *argv* is an array of pointers to strings (**char *[]**).
 - Under 3.0 and higher versions of DOS, *argv*[0] is the full path name of the program being run.
 - Under versions of DOS before 3.0, *argv*[0] points to the null string ("").
 - *argv*[1] points to the first string typed on the DOS command line after the program name.
 - *argv*[2] points to the second string typed after the program name.
 - *argv*[*argc*-1] points to the last argument passed to **main**.
 - *argv*[*argc*] contains null.

■ *env* is also an array of pointers to strings. Each element of *env*[] holds a string of the form *ENVVAR=value*.

- *ENVVAR* is the name of an environment variable, such as *PATH* or *87*.
- *value* is the value to which *ENVVAR* is set, such as *C:\DOS;C:\TOOLS;* (for *PATH*) or *YES* (for *87*).

If you declare any of these parameters, you *must* declare them exactly in the order given: *argc*, *argv*, *env*. For example, the following are all valid declarations of **main**'s arguments:

```
main()
main(int argc)           /* legal but very unlikely */
main(int argc, char * argv[])
main(int argc, char * argv[], char * env[])]
```

The declaration `main(int argc)` is legal, but it's very unlikely that you would use *argc* in your program without also using the elements of *argv*.

The argument *env* is also available through the global variable *environ*. You can read about the *environ* global variable in the online help file. Also, read about the **putenv** and **getenv** functions in the online help for more information.

argc and *argv* are also available via the global variables *_argc* and *_argv*.

An example program

Here is an example program, *ARGS.EXE*, that demonstrates a simple way of using these arguments passed to **main**.

```
/* Program ARGS.C */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
{
    int i;

    printf("The value of argc is %d \n\n",argc);
    printf("These are the %d command-line arguments passed to
           main:\n\n",argc);

    for (i = 0; i < argc; i++)
        printf("  argv[%d]: %s\n", i, argv[i]);

    printf("\nThe environment string(s) on this system are:\n\n");
```

```

    for (i = 0; env[i] != NULL; i++)
        printf("  env[%d]: %s\n", i, env[i]);
    return 0;
}

```

Suppose you run ARG5.EXE at the DOS prompt with the following command line:

```
C:> args first_arg "arg with blanks" 3 4 "last but one" stop!
```

Note that you can pass arguments with embedded blanks by surrounding them with double quotes, as shown by "argument with blanks" and "last but one" in this example command line.

The output of ARG5.EXE (assuming that the environment variables are set as shown here) would then be like this:

The value of argc is 7

These are the 7 command-line arguments passed to main:

```

argv[0]: C:\TC\TESTARG5.EXE
argv[1]: first_arg
argv[2]: arg with blanks
argv[3]: 3
argv[4]: 4
argv[5]: last but one
argv[6]: stop!

```

The environment string(s) on this system are:

```

env[0]: COMSPEC=C:\COMMAND.COM
env[1]: PROMPT=$p $g
env[2]: PATH=C:\SPRINT;C:\DOS;C:\TC

```

The maximum combined length of the command-line arguments passed to **main** (including the space between adjacent arguments and the name of the program itself) is 128 characters; this is a DOS limit.

Wildcard arguments

Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARG5.OBJ object file, which is included with Turbo C++.

Once WILDARG5.OBJ is linked into your program code, you can send wildcard arguments of the type *.* to your **main** function.

The argument will be expanded (in the *argv* array) to all files matching the wildcard mask. The maximum size of the *argv* array varies, depending on the amount of memory available in your heap.

If no matching files are found, the argument is passed unchanged. (That is, a string consisting of the wildcard mask is passed to **main**.)

Arguments enclosed in quotes ("...") are not expanded.

An example program

The following commands will compile the file ARG.S.C and link it with the wildcard expansion module WILDARGS.OBJ, then run the resulting executable file ARG.S.EXE:

```
TCC ARG.S WILDARGS.OBJ
ARG.S C:\TC\INCLUDE\*.H "*.C"
```

When you run ARG.S.EXE, the first argument is expanded to the names of all the *.H files in your Turbo C++ INCLUDE directory. Note that the expanded argument strings include the entire path. The argument *.C is not expanded as it is enclosed in quotes.

In the IDE, simply specify a project file (from the project menu) that contains the following lines:

```
ARG.S
WILDARGS.OBJ
```

Then use the **Run | Arguments** option to set the command-line parameters.



If you prefer the wildcard expansion to be the default, modify your standard C?.LIB library files to have WILDARGS.OBJ linked automatically. In order to accomplish that, remove SETARGV from the libraries and add WILDARGS. The following commands invoke the Turbo librarian (TLIB) to modify all the standard library files (assuming the current directory contains the standard C and C++ libraries and WILDARGS.OBJ):

For more on TLIB, see the online document called UTIL.DOC.

```
tlib cs -setargv +wildargs
tlib cc -setargv +wildargs
tlib cm -setargv +wildargs
tlib cl -setargv +wildargs
tlib ch -setargv +wildargs
```

Using `-p` (Pascal calling conventions)

If you compile your program using Pascal calling conventions, you *must* remember to explicitly declare **main** as a C type. Do this with the **cdecl** keyword, like this:

```
int cdecl main(int argc, char * argv[], char * envp[])
```

The value main returns

The value returned by **main** is the status code of the program: an **int**. If, however, your program uses the routine **exit** (or **_exit**) to terminate, the value returned by **main** is the argument passed to the call to **exit** (or to **_exit**).

For example, if your program contains the call

```
exit(1)
```

the status is 1.

If you are using the IDE (as opposed to the command-line compiler) to run your program, you can display the return value from **main** by selecting **File | Get Info**.

Using C++ streams

This chapter is divided into two sections: a brief, practical overview of using C++ stream I/O, and a reference section to the C++ stream class library.

Stream input/output in C++ (commonly referred to as *iostreams*, or merely *streams*) provide all the functionality of the **stdio** library in C. Iostreams are used to convert typed objects into readable text, and vice versa. Streams may also read and write binary data. The C++ language allows you to define or overload I/O functions and operators that are then called automatically for corresponding user-defined types.

What is a stream?

A stream is an abstraction referring to any flow of data from a source (or *producer*) to a *sink* (or *consumer*). We also use the synonyms *extracting*, *getting*, and *fetching* when speaking of inputting characters from a source; and *inserting*, *putting*, or *storing* when speaking of outputting characters to a sink. Classes are provided that support console output (`constrea.h`), memory buffers (`iostream.h`), files (`fstream.h`), and strings (`strstrea.h`) as sources or sinks (or both).

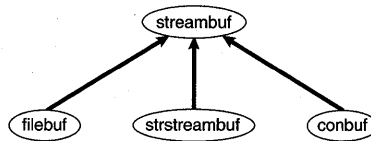
The iostream library

The **iostream** library has two parallel families of classes: those derived from **streambuf**, and those derived from **ios**. Both are low-level classes, each doing a different set of jobs. All stream classes have at least one of these two classes as a base class. Access from **ios**-based classes to **streambuf**-based classes is through a pointer.

The **streambuf** class

The **streambuf** class provides an interface to physical devices. **streambuf** provides underlying methods for buffering and handling streams when little or no formatting is required. The member functions of the **streambuf** family of classes are used by the **ios**-based classes. You can also derive classes from **streambuf** for your own functions and libraries. The classes **conbuf**, **filebuf** and **strstreambuf** are derived from **streambuf**.

Figure 16.1
Class **streambuf** and its
derived classes



The **ios** class

The **class ios** (and hence any of its derived classes) contains a pointer to a **streambuf**. It performs formatted I/O with error-checking using a **streambuf**.

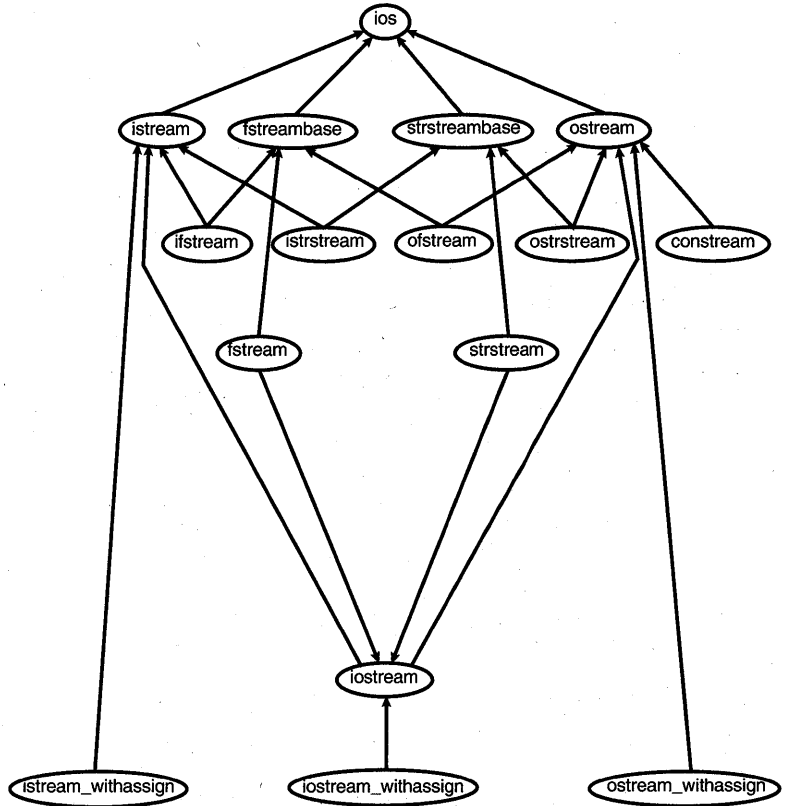
An inheritance diagram for all the **ios** family of classes is found in Figure 16.2. For example, the **ifstream** class is derived from the **istream** and **fstreambase** classes, and **istrstream** is derived from **istream** and **strstreambase**. This diagram is not a simple hierarchy because of the generous use of *multiple inheritance*. With multiple inheritance, a single class can inherit from more than one base class. (The C++ language provides for *virtual inheritance* to avoid multiple declarations.) This means, for example, that all the members (data and functions) of **iostream**, **istream**, **ostream**, **fstreambase**, and **ios** are part of objects of the **fstream** class. All classes in the **ios**-based tree use a **streambuf** (or a **filebuf** or **strstreambuf**, which are special cases of a **streambuf**) as its source and/or sink.

C++ programs start with four predefined open streams, declared as objects of **withassign** classes as follows:

```
extern istream_withassign cin; // Corresponds to stdin
extern ostream_withassign cout; // Corresponds to stdout
extern ostream_withassign cerr; // Corresponds to stderr
extern ostream_withassign clog; // A buffered cerr
```

Figure 16.2
Class **ios** and its derived
classes

By accepted practice, the
arrows point **from** the derived
class **to** the base class.



Output

Stream output is accomplished with the *insertion* (or *put to*) operator, `<<`. The standard left shift operator, `<<`, is overloaded for output operations. Its left operand is an object of type **ostream**. Its right operand is any type for which stream output has been defined (that is, fundamental types or any types you have overloaded it for). For example,

```
cout << "Hello!\n";
```

writes the string "Hello!" to **cout** (the standard output stream, normally your screen) followed by a new line.

The **<<** operator associates from left to right and returns a reference to the **ostream** object for which it is invoked. This allows several insertions to be cascaded as follows:

```
int i = 8;
double d = 2.34;
cout << "i = " << i << ", d = " << d << "\n";
```

This will write the following to standard output:

```
i = 8, d = 2.34
```

Fundamental types

The fundamental data types directly supported are **char**, **short**, **int**, **long**, **char*** (treated as a string), **float**, **double**, **long double**, and **void***. Integral types are formatted according to the default rules for **printf()** (unless you've changed these rules by setting various **ios** flags). For example, the following two output statements give the same result:

```
int i;
long l;
cout << i << " " << l;
printf("%d %ld", i, l);
```

The pointer (**void ***) inserter is used to display pointer addresses:

```
int i;
cout << &i;           // display pointer address in hex
```

Read the description of **class ostream** (page 554) for other output functions.

I/O formatting

Formatting for both input and output is determined by various *format state* flags contained in the **class ios**. The flags are read and set with the **flags()**, **setf()**, and **unsetf()** member functions.

Output formatting may also be affected by the use of the **fill()**, **width()**, and **precision()** member functions of **class ios**.

The format flags are detailed in **class ios**, data members, page 548.

Manipulators

A simple way to change some of the format variables is to use a special function-like operator called a *manipulator*. Manipulators take a stream reference as an argument and return a reference to the same stream. You can embed manipulators in a chain of insertions (or extractions) to alter stream states as a side effect without actually performing any insertions (or extractions). For example,

Parameterized manipulators must be called for each stream operation.

```
#include <iostream.h>
#include <iomanip.h> // Required for parameterized manipulators.

int main(void) {
    int i = 6789, j = 1234, k = 10;

    cout << setw(6) << i << j << i << k << j;
    cout << "\n";
    cout << setw(6) << i << setw(6) << j << setw(6) << k;
    return(0);
}
```

Produces this output:

```
678912346789101234
6789 1234 10
```

setw() is a *parameterized manipulator* declared in `iomanip.h`. Other parameterized manipulators, **setbase()**, **setfill()**, **setprecision()**, **setiosflags()** and **resetiosflags()**, work in the same way. To make use of these, your program must include `iomanip.h`. You can write your own manipulators without parameters:

```
#include <iostream.h>

// Tab and prefix the output with a dollar sign.
ostream& money( ostream& output) {
    return output << "\t$";
}

int main(void) {
    float owed = 1.35, earned = 23.1;
    cout << money << owed << money << earned;
    return(0);
}
```

produces the following output:

```
$1.35 $23.1
```


The non-parameterized manipulators **dec**, **hex**, and **oct** (declared in `iostream.h`) take no arguments and simply change the conversion base (and leave it changed):

```
int i = 36;
cout << dec << i << " " << hex << i << " " << oct << i << endl;
cout << dec; // Must reset to use decimal base.
// displays 36 24 44
```

Table 16.1
Stream manipulators

Manipulator	Action
dec	Set decimal conversion base format flag.
hex	Set hexadecimal conversion base format flag.
oct	Set octal conversion base format flag.
ws	Extract whitespace characters.
endl	Insert newline and flush stream.
ends	Insert terminal null in string.
flush	Flush an ostream.
setbase(int n)	Set conversion base format to base <i>n</i> (0, 8, 10, or 16). 0 means the default: decimal on output, ANSI C rules for literal integers on input.
resetiosflags(long f)	Clear the format bits specified by <i>f</i> .
setiosflags(long f)	Set the format bits specified by <i>f</i> .
setfill(int c)	Set the fill character to <i>c</i> .
setprecision(int n)	Set the floating-point precision to <i>n</i> .
setw(int n)	Set field width to <i>n</i> .

The manipulator **endl** inserts a newline character and flushes the stream. You can also flush an **ostream** at any time with

```
ostream << flush;
```

Filling and padding

The fill character and the direction of the padding depend on the setting of the fill character and the left, right, and internal flags.

The default fill character is a space. You can vary this by using the function **fill()**:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i; // display ***123
```

The default direction of padding gives right-justification (pad on the left). You can vary these defaults (and other format flags) with the functions **setf()** and **unsetf()**:

```
int i = 56;
```

```

cout.width(6);
cout.fill('#');
cout.setf(ios::left,ios::adjustfield);
cout << i;           // display 56####

```

The second argument, `ios::adjustfield`, tells `setf()` which bits to set. The first argument, `ios::left`, tells `setf()` what to set those bits to. Alternatively, you can use the manipulators `setfill()`, `setiosflags()`, and `resetiosflags()` to modify the fill character and padding mode. See `ios` data members on page 547 for a list of masks used by `setf()`.

Input

Stream input is similar to output but uses the overloaded right shift operator, `>>`, known as the *extraction (get from) operator*, or *extractor*. The left operand of `>>` is an object of type **class istream**. As with output, the right operand can be of any type for which stream input has been defined.

By default, `>>` skips whitespace (as defined by the `isspace()` function in `cctype.h`), then reads in characters appropriate to the type of the input object. Whitespace skipping is controlled by the `ios::skipws` flag in the format state's enumeration. The `skipws` flag is normally set to give whitespace skipping. Clearing this flag (with `setf()`, for example) turns off whitespace skipping. There is also a special "sink" manipulator, `ws`, that lets you discard whitespace.

Consider the following example:

```

int i;
double d;
cin >> i >> d;

```

When the last line is executed, the program skips any leading whitespace. The integer value (*i*) is then read. Any whitespace following the integer is ignored. Finally, the floating-point value (*d*) is read.

For type **char (signed or unsigned)**, the effect of the `>>` operator is to skip whitespace and store the next (non-whitespace) character. If you need to read the next character, whether it is whitespace or not, you can use one of the `get()` member functions (see the discussion of `istream`, beginning on page 551).

For type **char*** (treated as a string), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) characters until another whitespace character is found. A final null character is then appended. Care is needed to avoid "overflowing" a string. You can alter the default width of zero (meaning no limit) using **width()** as follows:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array;           // Avoids overflow.
```

For all input of fundamental types, if only whitespace is encountered nothing is stored in the target, and the istream state is set to *fail*. The target will retain its previous value; if it was uninitialized, it remains uninitialized.

I/O of user-defined types

To input or output your own defined types, you must overload the extraction and insertion operators. Here is an example:

```
#include <iostream.h>

struct info {
    char *name;
    double val;
    char *units;
};

// You can overload << for output as follows:
ostream& operator << (ostream& s, info& m) {
    s << m.name << " " << m.val << " " << m.units;
    return s;
};

// You can overload >> for input as follows:
istream& operator >> (istream& s, info& m) {
    s >> m.name >> m.val >> m.units;
    return s;
};

int main(void) {
    info x;
    x.name = new char[15];
    x.units = new char[10];

    cout << "\nInput name, value and units:";
    cin >> x;
    cout << "\nMy input:" << x;
```

```
    return(0);  
}
```

Simple file I/O

The class **ofstream** inherits the insertion operations from **ostream**, while **ifstream** inherits the extraction operations from **istream**. The file-stream classes also provide constructors and member functions for creating files and handling file I/O. You must include `fstream.h` in all programs using these classes.

Consider the following example that copies the file `FILE.IN` to the file `FILE.OUT`:

```
#include <fstream.h>  
  
int main(void) {  
    char ch;  
    ifstream f1("FILE.IN");  
    ofstream f2("FILE.OUT");  
  
    if (!f1) cerr << "Cannot open FILE.IN for input";  
    if (!f2) cerr << "Cannot open FILE.OUT for output";  
    while (f2 && f1.get(ch))  
        f2.put(ch);  
    return(0);  
}
```

Note that if the **ifstream** or **ofstream** constructors are unable to open the specified files, the appropriate stream error state is set.

The constructors allow you to declare a file stream without specifying a named file. Later, you can associate the file stream with a particular file:

```
ofstream ofile;           // creates output file stream  
...  
ofile.open("payroll"); // ofile connects to file "payroll"  
// do some payrollling...  
  
ofile.close();           // close the ofile stream  
ofile.open("employee"); // ofile can be reused...
```

By default, files are opened in text mode. This means that on input, carriage-return/linefeed sequences are converted to the `'\n'` character. On output, the `'\n'` character is converted to a carriage-return/linefeed sequence. These translations are not done in binary mode. The file-opening mode is set with an optional

second parameter to the **open()** function or in some constructors. The file opening-mode constants can be used alone or they can be logically ORed together. See **class ios**, data members, on page 548.

String stream processing

The functions defined in `strstream.h` support in-memory formatting, similar to **sscanf()** and **sprintf()**, but much more flexible. All of the **istream** member functions are available for **class istream** (input *string stream*); likewise for output: **ostream** inherits from **ostream**.

Given a text file with the following format:

```
101 191 Cedar Chest
102 1999.99 Livingroom Set
```

Each line can be parsed into three components: an integer ID, a floating-point price, and a description. The output produced is:

```
1: 101 191.00 Cedar Chest
2: 102 1999.99 Livingroom Set
```

Here is the program:

```
#include <fstream.h>
#include <strstream.h>
#include <iomanip.h>
#include <string.h>

int main(int argc, char **argv) {
    int id;
    float amount;
    char description[41];
    ifstream inf(argv[1]);

    if (inf) {
        char inbuf[81];
        int lineno = 0;

        // Want floats to print as fixed point
        cout.setf(ios::fixed, ios::floatfield);

        // Want floats to always have decimal point
        cout.setf(ios::showpoint);

        while (inf.getline(inbuf,81)) {
            // 'ins' is the string stream:
            istream ins(inbuf,strlen(inbuf));
```

```

        ins >> id >> amount >> ws;
        ins.getline(description,41); // Linefeed not copied.
        cout << ++lineno << ": "
            << id << '\t'
            << setprecision(2) << amount << '\t'
            << description << "\n";
    }
}
return(0);
}

```

Note the use of format flags and manipulators in this example. The calls to **setw()** coupled with **setprecision()** allow floating-point numbers to be printed in a money format. The manipulator **ws** skips whitespace before the description string is read.

Screen output streams

The class **constream**, derived from **ostream** and defined in `constrea.h`, provides the functionality of `conio.h` for use with C++ streams. This allows you to create output streams that write to specified areas of the screen, in specified colors, and at specific locations.

Console stream manipulators are provided to facilitate formatting of console streams. These manipulators work in the same way as the corresponding function provided by `conio.h`. For a detailed description of the manipulators' behavior and valid arguments, see online Help.

Table 16.2
Console stream manipulators

Manipulator	conio function	Action
clreol	clreol	Clears to end of line in text window.
delline	delline	Deletes line in the text window.
highvideo	highvideo	Selects high-intensity characters.
insline	insline	Inserts a blank line in the text window.
lowvideo	lowvideo	Selects low-intensity characters.
normvideo	normvideo	Selects normal-intensity characters.
setattr(int)	textattr	Sets screen attributes.
setbk(int)	textcolor	Sets new character color.
setclr(int)	textcolor	Set the color.
setcrstye(int)	_setcursortype	Selects cursor appearance.
setxy(int, int)	gotoxy	Positions the cursor at the specified position.

Typical use of parameterized manipulators.

```
#include <constrea.h>

int main(void) {
    constream win1;

    win1.window(1, 1, 40, 20); // Initialize the desired space.
    win1.clrscr();             // Clear this rectangle.

    // Use the parameterized manipulator to set screen attributes.
    win1 << setattr((BLUE<<4 | WHITE)
        << "This text is white on blue.");

    // Use this parameterized manipulator to specify output area.
    win1 << setxy(10, 10)
        << "This text is in the middle of the window.";
    return(0);
}
```

You can create multiple constreams, each writing to its own portion of the screen. Then, you can output to any of them without having to reset the window each time.

```
#include <constrea.h>

int main(void) {
    constream demo1, demo2;

    demo1.window( 1, 2, 40, 10 );
    demo2.window( 1, 12, 40, 20 );

    demo1.clrscr();
    demo2.clrscr();

    demo1 << "Text in first window" << endl;
    demo2 << "Text in second window" << endl;
}
```

```

demo1 << "Back to the first window" << endl;
demo2 << "And back to the second window" << endl;
return(0);
}

```

Stream class reference

The stream class library in C++ consists of several classes. This reference presents some of the most useful details of these classes, in alphabetical organization. The following cross-reference lists tell which classes belong to which header files.

constrea.h:	conbuf, constream
iostream.h:	ios, iostream, iostream_withassign, istream, istream_withassign, ostream, ostream_withassign, streambuf.
fstream.h:	filebuf, fstream, fstreambase, ifstream, ofstream.
strstream.h:	istrstream, ostrstream, strstream, strstreambase, strstreambuf.

conbuf

<constrea.h>

Specializes **streambuf** to handle console output.

constructor conbuf()

Makes an unattached **conbuf**.

Member functions

cleol void cleol()

Clears to end of line in text window.

clrscr void clrscr()

Clears the defined screen.

delline void delline()

Deletes a line in the window.

conbuf

gotoxy	void gotoxy(int x, int y) Positions the cursor in the window at the specified location.
highvideo	void highvideo() Selects high-intensity characters.
incline	void incline() Inserts a blank line.
lowvideo	void lowvideo() Selects low-intensity characters.
normvideo	void normvideo() Selects normal-intensity characters.
overflow	virtual int overflow(int = EOF) Flushes the conbuf to its destination.
setcursortype	void setcursortype(int cur_type) Selects the cursor appearance.
textattr	void textattr(int newattribute) Selects cursor appearance.
textbackground	void textbackground(int newcolor) Selects the text background color.
textcolor	void textcolor(int newcolor) Selects character color in text mode.
textmode	static void textmode(int newmode) Puts the screen in text mode.
wherex	int wherex() Gets the horizontal cursor position.
wherey	int wherey() Gets the vertical cursor position.
window	void window(int left, int top, int right, int bottom) Defines the active window.

constream

<constrea.h>

Provides console output streams. This class is derived from **ostream**.

constructor `constream()`

Provides an unattached output stream to the console.

 Member
functions

clrscr `void clrscr()`

Clears the screen.

rdbuf `conbuf *rdbuf()`

Returns a pointer to this constream's assigned conbuf.

textmode `void textmode(int newmode)`

Puts the screen in text mode.

window `void window(int left, int top, int right, int bottom)`

Defines the active window.

filebuf

<fstream.h>

Specializes **streambuf** to handle files.

constructor `filebuf();`

Makes a **filebuf** that isn't attached to a file.

constructor `filebuf(int fd);`

Makes a **filebuf** attached to a file as specified by file descriptor *fd*.

constructor `filebuf(int fd, char *, int n);`

Makes a **filebuf** attached to a file and uses a specified *n*-character buffer.

 Data
members

openprot `static const int openprot`

filebuf

The default file protection. The exact value of *openprot* should not be of interest to the user. Its purpose is to set the file permissions to read and write.

Member functions

- attach** filebuf* attach(int)
Attaches this closed **filebuf** to opened file descriptor.
- close** filebuf* close()
Flushes and closes the file. Returns 0 on error.
- fd** int fd()
Returns the file descriptor or EOF.
- is_open** int is_open();
Returns nonzero if the file is open.
- open** filebuf* open(const char *name, int mode, int prot = filebuf::openprot);
Opens the file specified by *name* and connects to it. The file-opening mode is specified by *mode*.
- overflow** virtual int overflow(int = EOF);
Flushes a buffer to its destination. Every derived class should define the actions to be taken.
- seekoff** virtual streampos seekoff(streamoff, ios::seek_dir, int);
Moves the file pointer relative to the current position.
- setbuf** virtual streambuf* setbuf(char*, int);
Specifies a buffer for this **filebuf**.
- sync** virtual int sync();
Establishes consistency between internal data structures and the external stream representation.
- underflow** virtual int underflow();
Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

fstream

<fstream.h>

This stream class, derived from **fstreambase** and **iostream**, provides for simultaneous input and output on a **filebuf**.

- constructor** `fstream();`
 Makes an **fstream** that isn't attached to a file.
- constructor** `fstream(const char *name, int mode, int prot = filebuf::openprot);`
 Makes an **fstream**, opens a file, and connects to it.
- constructor** `fstream(int fd);`
 Makes an **fstream**, connects to an open file descriptor specified by *fd*.
- constructor** `fstream(int fd, char *buf, int buf_len);`
 Makes an **fstream** connected to an open file handle specified by *fd*.

 Member functions

- open** `void open(const char *name, int mode = ios::in, int prot = filebuf::openprot);`
 Opens a file specified by *name* for an **fstream**. The file-opening mode is specified by the variable *mode*.
- rdbuf** `filebuf* rdbuf();`
 Returns the filebuf used.

fstreambase

<fstream.h>

This stream class, derived from **ios**, provides operations common to file streams. It serves as a base for **fstream**, **ifstream**, and **ofstream**.

- constructor** `fstreambase();`
 Makes an **fstreambase** that isn't attached to a file.
- constructor** `fstreambase(const char *, int mode, int = filebuf::openprot);`
 Makes an **fstreambase**, opens a file in mode specified by *mode*, and connects to it.
- constructor** `fstreambase(int fd);`

fstreambase

Makes an **fstreambase**, connects to an open file descriptor specified by *fd*.

constructor `fstreambase(int fd, char *buf, int len);`

Makes an **fstreambase** connected to an open file descriptor specified by *fd*. The buffer is specified by *buf* and the buffer size is *len*.

Member functions

attach `void attach(int);`

Connects to an open file descriptor.

close `void close();`

Closes the associated **filebuf** and file.

open `void open(const char *name, int mode, int prot = filebuf::openprot);`

Opens a file for an **fstreambase**. The file-opening mode is specified by *mode*.

rdbuf `filebuf* rdbuf();`

Returns the **filebuf** used.

setbuf `void setbuf(char*, int);`

Uses a specified buffer.

ifstream

<fstream.h>

This stream class, derived from **fstreambase** and **istream**, provides input operations on a **filebuf**.

constructor `ifstream();`

Makes an **ifstream** that isn't attached to a file.

constructor `ifstream(const char *name, int mode = ios::in, int = filebuf::openprot);`

Makes an **ifstream**, opens a file for input in protected mode, and connects to it. The existing file contents are preserved; new writes are appended.

constructor `ifstream(int fd);`

Makes an **ifstream**, connects to an open file descriptor *fd*.

constructor `ifstream(int fd, char *buf, int buf_len);`

Makes an **ifstream** connected to an open file. The file is specified by its descriptor, *fd*. The uses the buffer specified by *buf* of length *buf_len*.

Member functions

open void open(const char*, int, int = filebuf::openprot);

Opens a file for an **ifstream**.

rdbuf filebuf* rdbuf();

Returns the filebuf used.

ios <iostream.h>

Provides operations common to both, input and output. Its derived classes (**istream**, **ostream**, **iostream**) specialize I/O with high-level formatting operations. The **ios** class is a base for **istream**, **ostream**, **fstreambase**, and **stringstreambase**.

constructor ios(); **protected**

Constructs an **ios** object that has no corresponding **streambuf**.

constructor ios(streambuf *);

Associates a given **streambuf** with the stream.

Data members

The three following constants are used as the second parameter of the **setf()** function.

```
static const long  adjustfield; // left | right | internal
static const long  basefield;  // dec | oct | hex
static const long  floatfield; // scientific | fixed

streambuf         *bp;          // the associated streambuf   protected
int               x_fill;      // padding character on // output protected
long              x_flags;     // formatting flag bits protected
int               x_precision; // floating-point precision on // output protected
```

```

int          state;          // current state of the
                                // streambuf
ostream     *x_tie;         // the tied ostream, if any
int         x_width;        // field width on output

```

protected
protected
protected

Stream seek direction.

```
enum seek_dir { beg=0, cur=1, end=2 };
```

Stream operation mode. These may be logically ORed.

```
enum open_mode {
    app,          Append data—always write at end of file.
    ate,          Seek to end of file upon original open.
    in,           Open for input (default for ifstream).
    out,          Open for output (default for ofstream).
    binary,       Open file in binary mode.
    trunc,        Discard contents if file exists (default if out is specified
                    and neither ate nor app is specified).
    nocreate,     If file does not exist, open() fails.
    noreplace,    If file exists, open() for output fails unless ate or app is
                    set.
};
```

Format flags used with **flags(), **setf()**, and **unsetf()** member functions.**

```
enum {
    skipws,       Skip whitespace on input.
    left,         Left-adjust output.
    right,        Right-adjust output.
    internal,     Pad after sign or base indicator.
    dec,          Decimal conversion.
    oct,          Octal conversion.
    hex,          Hexadecimal conversion.
    showbase,     Show base indicator on output.
    showpoint,    Show decimal point for floating-point output.
    uppercase,    Uppercase hex output.
    showpos,      Show '+' with positive integers.
    scientific,   Suffix floating-point numbers with exponential (E)
                    notation on output.
    fixed,        Use fixed decimal point for floating-point numbers.
    unitbuf,      Flush all streams after insertion.
    stdio,        Flush stdout, stderr after insertion.
};
```

Member functions

- bad** `int bad();`
 Nonzero if error occurred.
- bitalloc** `static long bitalloc();`
 Acquires a new flag bit set. The return value may be used to set, clear, and test the flag. This is for user-defined formatting flags.
- clear** `void clear(int = 0);`
 Sets the stream state to the given value.
- eof** `int eof();`
 Nonzero on end of file.
- fail** `int fail();`
 Nonzero if an operation failed.
- fill** `char fill();`
 Returns the current fill character.
- fill** `char fill(char);`
 Resets the fill character; returns the previous one.
- flags** `long flags();`
 Returns the current format flags.
- flags** `long flags(long);`
 Sets the format flags to be identical to the given **long**; returns previous flags. Use **flags(0)** to set the default format.
- good** `int good();`
 Nonzero if no state bits set (that is, no errors appeared).
- init** `void init(streambuf *);` **protected**
 Provides the actual initialization.
- precision** `int precision();`
 Returns the current floating-point precision.
- precision** `int precision(int);`

Sets the floating-point precision; returns previous setting.

rdbuf streambuf* rdbuf();

Returns a pointer to this stream's assigned streambuf.

rdstate int rdstate();

Returns the stream state.

self long setf(long);

Sets the flags corresponding to those marked in the given **long**; returns previous settings.

self long setf(long *_setbits*, long *_field*);

The bits corresponding to those marked in *_field* are cleared, and then reset to be those marked in *_setbits*.

setstate protected: void setstate(int);

Sets all status bits.

sync_with_stdio static void sync_with_stdio();

Mixes `stdio` files and `iostreams`. This should not be used for new code.

tie ostream* tie();

Returns the *tied stream*, or zero if none. Tied streams are those that are connected such that when one is used, the other is affected. For example, **cin** and **cout** are tied; when **cin** is used, it flushes **cout** first.

tie ostream* tie(ostream*);

Ties another stream to this one and returns the previously tied stream, if any. When an input stream has characters to be consumed, or if an output stream needs more characters, the tied stream is first flushed automatically. By default, **cin**, **cerr** and **clog** are tied to **cout**.

unsetf long unsetf(long);

Clears the bits corresponding to those marked in the given **long**; returns previous settings.

width int width();

Returns the current width setting.

width int width(int);

Sets the width as given; returns the previous width.

xalloc static int xalloc();

Returns an array index of previously unused words that can be used as user-defined formatting flags.

iostream

<iostream.h>

This class, derived from **istream** and **ostream**, is simply a mixture of its base classes, allowing both input and output on a stream. It is a base for **fstream** and **strstream**.

constructor `iostream(streambuf *);`

Associates a given **streambuf** with the stream.

iostream_withassign

<iostream.h>

This class is an **istream** with an added assignment operator.

constructor `iostream_withassign();`

Default constructor (calls **istream**'s constructor).

Member
functions

None (although the = operator is overloaded).

istream

<iostream.h>

Provides formatted and unformatted input from a **streambuf**. The >> operator is overloaded for all fundamental types, as explained in the narrative at the beginning of the chapter. This **ios** class is a base for **ifstream**, **iostream**, **istrstream**, and **istream_withassign**.

constructor `istream(streambuf *);`

Associates a given **streambuf** with the stream.

Member
functions

eatwhite `void eatwhite();`

protected

Extract consecutive whitespace.

istream

- gcount** `int gcount();`
Returns the number of characters last extracted.
- get** `int get();`
Extracts the next character or EOF.
- get** `istream& get(signed char*, int len, char = '\n');`
`istream& get(unsigned char*, int len, char = '\n');`
Extracts characters into the given **char *** until the delimiter (third parameter) or end-of-file is encountered, or until (*len* - 1) bytes have been read. A terminating null is always placed in the output string; the delimiter never is. The delimiter remains in the stream. Fails only if no characters were extracted.
- get** `istream& get(signed char&);`
`istream& get(unsigned char&);`
Extracts a single character into the given character reference.
- get** `istream& get(streambuf&, char = '\n');`
Extracts characters into the given **streambuf** until the delimiter is encountered.
- getline** `istream& getline(signed char *buffer, int, char = '\n');`
`istream& getline(unsigned char *buffer, int, char = '\n');`
Same as **get**, except the delimiter is also extracted. The delimiter is not copied to *buffer*.
- ignore** `istream& ignore(int n = 1, int delim = EOF);`
Causes up to *n* characters in the input stream to be skipped; stops if *delim* is encountered.
- peek** `int peek();`
Returns next char without extraction.
- putback** `istream& putback(char);`
Pushes back a character into the stream.
- read** `istream& read(signed char*, int);`
`istream& read(unsigned char*, int);`
Extracts a given number of characters into an array. Use **gcount()** for the number of characters actually extracted if an error occurred.

- seekg** `istream& seekg(streampos);`
 Moves to an absolute position (as returned from **tellg**).
- seekg** `istream& seekg(streamoff, seek_dir);`
 Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**
- tellg** `streampos tellg();`
 Returns the current stream position.

istream_withassign

<iostream.h>

This class is an **istream** with an added assignment operator.

- constructor** `istream_withassign();`
 Default constructor (calls **istream**'s constructor).

Member
functions

None (although the = operator is overloaded).

istream

<strstream.h>

Provides input operations on a **strstreambuf**. This class is derived from **strstreambase** and **istream**.

- constructor** `istream(char *);`
 Makes an **istream** with a specified string (a null character is never extracted).
- constructor** `istream(char *, int n);`
 Makes an **istream** using up to *n* bytes of a specified string.

ofstream

<fstream.h>

Provides input operations on a **filebuf**. This class is derived from **fstreambase** and **ostream**.

ofstream

- constructor** `ofstream();`
Makes an **ofstream** that isn't attached to a file.
- constructor** `ofstream(const char *name, int mode = ios::out, int prot = filebuf::openprot);`
Makes an **ofstream**, opens a file, and connects to it.
- constructor** `ofstream(int fd);`
Makes an **ofstream**, connects to an open file descriptor specified by *fd*.
- constructor** `ofstream(int fd, char *buf, int len);`
Makes an **ofstream** connected to an open file descriptor specified by *fd*. The buffer specified by *buf* of *len* is used by the **ofstream**.

Member functions

- open** `void open(const char*, int = ios::out, int = filebuf::openprot);`
Opens a file for an **ofstream**.
- rdbuf** `filebuf* rdbuf();`
Returns the **filebuf** used.

ostream

<iostream.h>

Provides formatted and unformatted output to a **streambuf**. The `<<` operator is overloaded for all fundamental types, as explained on page 531. This **ios**-based class is a base for **constream**, **iostream**, **ofstream**, **ostrstream**, and **ostream_withassign**.

- constructor** `ostream(streambuf *);`
Associates a given **streambuf** with the stream.

Member functions

- flush** `ostream& flush();`
Flushes the stream.
- put** `ostream& put(char);`

Inserts the character.

seekp ostream& seekp(streampos);

Moves to an absolute position (as returned from **tellp**).

seekp ostream& seekp(streamoff, seek_dir);

Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**

tellp streampos tellp();

Returns the current stream position.

write ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);

Inserts *n* characters (nulls included).

ostream_withassign

<iostream.h>

This class is an **ostream** with an added assignment operator.

constructor ostream_withassign();

Default constructor (calls **ostream**'s constructor).

Member functions

None (although the = operator is overloaded).

ostrstream

<strstrea.h>

Provides output operations on a **strstreambuf**. This class is derived from **strstreambase** and **ostream**.

constructor ostrstream();

Makes a dynamic **ostrstream**.

constructor ostrstream(char *buf, int len, int mode = ios::out);

Makes a **ostrstream** with a specified *len*-byte buffer. If the file-opening mode is **ios::app** or **ios::ate**, the get/put pointer is positioned at the null character of the string.

ostream

Member functions

-
- pcount** int pcount();
Returns the number of bytes currently stored in the buffer.
- str** char *str();
Returns and freezes the buffer. You must deallocate it if it was dynamic.

streambuf

<iostream.h>

This is a buffer-handling class. Your applications gain access to buffers and buffering functions through a pointer to **streambuf** that is set by **ios**. **streambuf** is a base for **filebuf** and **strstreambuf**.

- constructor** streambuf();
Creates an empty buffer object.
- constructor** streambuf(char *, int);
Uses the given array and size as the buffer.

Member functions

-
- allocate** int allocate(); **protected**
Sets up a buffer area.
- base** char *base(); **protected**
Returns the start of the buffer area.
- blen** int blen(); **protected**
Returns the length of buffer area.
- eback** char *eback(); **protected**
Returns the base of putback section of get area.
- ebuf** char *ebuf(); **protected**
Returns the end+1 of the buffer area.
- egptr** char *egptr(); **protected**

	Returns the end+1 of the get area.	
epptr	char *epptr();	protected
	Returns the end+1 of the put area.	
gbump	void gbump(int);	protected
	Advances the get pointer.	
gptr	char *gptr();	protected
	Returns the next location in get area.	
in_avail	int in_avail();	
	Returns the number of characters remaining in the input buffer.	
out_waiting	int out_waiting();	
	Returns the number of characters remaining in the output buffer.	
pbase	char *pbase();	protected
	Returns the start of put area.	
pbump	void pbump(int);	protected
	Advances the put pointer.	
pptr	char *pptr();	protected
	Returns the next location in put area.	
sbumpc	int sbumpc();	
	Returns the current character from the input buffer, then advances.	
seekoff	virtual streampos seekoff(streamoff, ios::seek_dir, int = (ios::in ios::out));	
	Moves the get or put pointer (the third argument determines which one or both) relative to the current position.	
seekpos	virtual streampos seekpos(streampos, int = (ios::in ios::out));	
	Moves the get or put pointer to an absolute position.	
setb	void setb(char *, char *, int = 0);	protected
	Sets the buffer area.	
setbuf	virtual streambuf* setbuf(signed char *, int); streambuf* setbuf(unsigned char *, int);	
	Connects to a given buffer.	

streambuf

setg	void setg(char *, char *, char *); Initializes the get pointers.	protected
setp	void setp(char *, char *); Initializes the put pointers.	protected
sgetc	int sgetc(); Peeks at the next character in the input buffer.	
sgetn	int sgetn(char*, int n); Gets the next <i>n</i> characters from the input buffer.	
snextc	int snextc(); Advances to and returns the next character from the input buffer.	
sputbackc	int sputbackc(char); Returns a character to input.	
sputc	int sputc(int); Puts one character into the output buffer.	
sputn	int sputn(const char*, int n); Puts <i>n</i> characters into the output buffer.	
stoss	void stoss(); Advances to the next character in the input buffer.	
unbuffered	void unbuffered(int); Sets the buffering state.	protected
unbuffered	int unbuffered(); Returns non-zero if not buffered.	protected

strstreambase

<strstrea.h>

Specializes **ios** to string streams. This class is entirely protected except for the member function **strstreambase::rdbuf()**. This class is a base for **strstream**, **istrstream**, and **ostrstream**.

constructor	strstreambase(); Makes an empty strstreambase .	protected
--------------------	---	------------------

constructor `strstreambase(char *, int, char *start);` **protected**
 Makes an **strstreambase** with a specified buffer and starting position.

Member functions

rdbuf `strstreambuf * rdbuf();`
 Returns a pointer to the `strstreambuf` associated with this object.

strstreambuf

<strstrea.h>

Specializes **streambuf** for in-memory formatting.

constructor `strstreambuf();`
 Makes a dynamic **strstreambuf**. Memory will be dynamically allocated as needed.

constructor `strstreambuf(void * (*)(long), void (*)(void *));`
 Makes a dynamic buffer with specified allocation and free functions.

constructor `strstreambuf(int n);`
 Makes a dynamic **strstreambuf**, initially allocating a buffer of at least *n* bytes.

constructor `strstreambuf(signed char *, int, signed char *strt = 0);`
`strstreambuf(unsigned char *, int, unsigned char *strt = 0);`
 Makes a static **strstreambuf** with a specified buffer. If *strt* is not null, it delimits the buffer.

Member functions

doallocate `virtual int doallocate();`
 Performs low-level buffer allocation.

freeze `void freeze(int = 1);`
 If the input parameter is nonzero, disallows storing any characters in the buffer. Unfreeze by passing a zero.

overflow `virtual int overflow(int);`

strstreambuf

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

seekoff virtual streampos seekoff(streamoff, ios::seek_dir, int);

Moves the pointer relative to the current position.

setbuf virtual streambuf* setbuf(char*, int);

Specifies the buffer to use.

str char *str();

Returns a pointer to the buffer and freezes it.

sync virtual int sync();

Establishes consistency between internal data structures and the external stream representation.

underflow virtual int underflow();

Makes input available. This is called when a character is requested and the strstreambuf is empty. Every derived class should define the actions to be taken.

strstream

<strstrea.h>

Provides for simultaneous input and output on a **strstreambuf**. This class is derived from **strstreambase** and **iostream**.

constructor strstream();

Makes a dynamic **strstream**.

constructor strstream(char*, int sz, int mode);

Makes a **strstream** with a specified *sz*-byte buffer. If *mode* is **ios::app** or **ios::ate**, the get/put pointer is positioned at the null character of the string.

Member function

str char *str();

Returns and freezes the buffer. The user must deallocate it if it was dynamic.

Converting from Microsoft C

If you're an experienced C or C++ programmer, but the Turbo C++ programming environment is new to you, then you should read this chapter before you do anything else. We appreciate that you want to be up and running fast with a new piece of software, and we know that you want to spend as little time as possible reading the manual. However, the time you spend reading this chapter will probably save you a lot of time later. Please read on.

Environment and tools

The Turbo C++ IDE (integrated development environment) is roughly the equivalent of the Programmer's Workbench, although naturally we think you'll find the IDE much easier to use.

You can find out more about configuration and project files in Chapters 2 and 7

The IDE loads its settings from two files: TCCONFIG.TC, the default configuration file, and a project file (.PRJ). TCCONFIG.TC contains general environmental information. The current project file contains information more specific to the application you're building.

A project is the IDE's equivalent of a makefile. It includes the list of files to be built, as well as settings for the IDE options that control the compilation and linkage of that program. If you don't specify a project file when you start the IDE, a nameless project is opened and set with default compiler and linker options, but no file name list.

Unlike Microsoft C, however, Turbo C++ does not automatically create and run a makefile based on settings and file names that you give it in the project. If you want to use the IDE to set up a project, but use MAKE to do the actual build, then you can use the PRJ2MAK utility to convert a project file to a makefile.

The following sections describe the significant differences between Turbo C++'s MAKE, Project Manager, linker (TLINK), and command-line compiler (TCC) and Microsoft C's NMAKE, LINK, and CL.

Paths for .h and .LIB files

Microsoft C works with two environment variables, LIB and INCLUDE. The Microsoft linker uses the LIB variable to discover the location of the run-time libraries; similarly, INCLUDE is used to find standard header files. Turbo C++ does not use environment variables to store the path for the library or include files. Instead, you can easily set these paths in the IDE using the environment options. If you are working with the command-line compiler or the linker, you can use command-line options or configuration files.

When you install Turbo C++, you are asked to set paths for include files and library files. Those paths are then the default paths in the IDE. The include and library files paths are also written to the default command-line compiler configuration file TURBOC.CFG. The library path is written to the default stand-alone linker configuration file TLINK.CFG.

Remember that even if you haven't opened a project, Turbo C++ will store the paths in its default project file.

In the IDE, reset default search paths for libraries and header files with the Options | Directories command. The settings in the Directories dialog box become a part of the current project.

For the command-line compiler, you can reset the search path for include and library files with the `-I` and `-L` options, respectively. These options can also be reset in the configuration file for the command-line compiler, TURBOC.CFG.

The linker can use the `/L` option to change search paths for libraries and initialization code (like C0S.OBJ, the startup code for the small memory model). For instance, the option

```
/LC:\TC\LIB;C:\APPS\LIB
```

tells the linker to look in the two paths named for library and initialization files.

You can also create a TLINK.CFG file. TLINK.CFG is a regular text file that contains a list of valid TLINK options.

MAKE

New!

The version of MAKE supplied with Turbo C++ 3.0 contains many new features, some of which are designed to increase compatibility with Microsoft's NMAKE. The new command-line switch **-N** turns on full NMAKE compatibility. See Chapter 9 for more information on MAKE's options. The following list summarizes the differences between MAKE and NMAKE.

- NMAKE supports response files but MAKE doesn't.
- In NMAKE, you must surround strings to be compared with quotes. MAKE doesn't have this requirement; as long as the string to be compared doesn't contain spaces, you can compare them without quotes.
- NMAKE predefines several implicit rules; MAKE doesn't. However, the BUILTINS.MAK file contains several implicit rules that you can use without specifying them in the makefile.

Command-line compiler

The following table lists comparable TCC and CL command-line compiler options. If an option is in one product, but not the other, it is omitted. Some of the CPP (standalone preprocessor) options are listed. In many multi-pass compilers, a separate pass performs the work of the preprocessor, and the results of the pass can be examined. Since Turbo C++ uses an integrated single-pass compiler, we provide the standalone utility CPP to supply the first-pass functionality found in other compilers.

Note that most CL options that take arguments allow for a space between the option and the argument. TCC options that take arguments are usually immediately followed by the argument or list.

Table 17.1: CL and TCC options compared

Microsoft C CL option	Turbo C++ TCC option	What it does
(See /Zpn)	-a	Align word.
(See /Zpn)	-a-	Align byte (default).
/Ax	-mx	Use memory model <i>x</i> . For TCC, following <i>t</i> , <i>s</i> , or <i>m</i> with ! tells compiler to assume DS != SS.
/C	-C	Nested comments on.
/c	-c	Compile to .OBJ but do not link.
/Did	-Dname	Define <i>name</i> to the string consisting of the null character.
/Did=value	-Dname=string	Defines <i>name</i> to <i>string</i> .
/E	CPP -P	Preprocess source to standard output, include line numbers.
/EP	CPP -P-	Preprocess source to standard output, without line numbers.
/F hexnum	(See note)	Sets stack size to <i>hexnum</i> bytes (<i>hexnum</i> must be hexadecimal). In Turbo C++ code, initialize the global variable <i>_stklen</i> .
(By default)	-Fc	Generates COMDEFS.
(By default)	-Fs	Make DS == SS for all memory models.
/Fa [listfile]	-S	Create assembly listing. Name for list file defaults to <i>Source.ASM</i> .
/Fe exefile	-eexefile	<i>exefile</i> names executable file.
/Fm [mapfile]	-M	Creates map file. Name defaults to <i>Source.MAP</i> , where <i>source</i> is the first source file specified.
/Fo objfile	-oobjfile	<i>objfile</i> names object file.
/FPc	(default)	Emulate floating point (default for Turbo C++); coprocessor used if present at run time.
/FPI	-f	Inlines 80x87 instructions; selects emulator library (coprocessor used if present at run time).
/FPI87	-f87 or -f287	Inlines 80x87 instructions; chooses coprocessor library (coprocessor must be present at run time).
/G0	-1	Generate 80186 instructions.
/G1	-1-	Generate 8088/8086 instructions (default).
/G2	-2	Generate 80286 protected-mode compatible instructions.
/Gc	-p	Use Pascal calling convention. For CL, this is Pascal or FORTRAN, but currently same calling convention.
/Gd	-p-	Standard C calling conventions (default).
/Ge	-N	Check for stack overflow. (Default for CL, but not for TCC).
/Gs	-N-	Turn off checking for stack overflow. (Off by default for TCC.)
/Gt [number]	-Ff[=size]	Creates far variables automatically; <i>size</i> or <i>number</i> is threshold.
By default	-h	Use fast huge pointer arithmetic.
/H number	-inumber	Restricts length of external names to <i>number</i> .
/HELP	TCC	Calls QuickHelp. For Help on TCC, simply invoke without options or load THELP.
/I directory	-lpath	Directories for include files. For CL, adds <i>directory</i> to the beginning of include file search directory list. See page 562.
/J	-K	Changes default for char from signed to unsigned. For Turbo C++, -K- returns to signed.
/link options	-loptions	Pass <i>options</i> to linker when invoked.
/MAoption	-Toption	Pass to assembler when invoked.
/NDdaseg	-zRname	Sets the data segment name. For TCC, this option changes the name of the uninitialized data segment class to <i>name</i> . By default, the uninitialized data segments are assigned to class BSS.

Table 17.1: CL and TCC options compared (continued)

/NTsegname	-zCname	Sets code segment name. This option changes the name of the code segment to <i>name</i> . By default, the code segment is named <code>_TEXT</code> , except for the medium, large and huge models, where the name is <code>filename_TEXT</code> . (<i>filename</i> here is the source file name.)
/Op	-ff -	Strict ANSI floating point.
/Os	-G -	Optimize for size (default).
/Ot	-G	Optimize for speed.
/P	CPP -ofilename	Preprocesses source file and sends output to <i>filename</i> (CPP), or to <i>Source.I</i> (CL).
/U Ident	-UIdent	Undefine any previous definitions of <i>Ident</i> .
/w	-w-	Display warnings off.
/W n	(See note)	Set warning level 0, 1, 2, 3, or 4. In Turbo C++ you can selectively enable or disable any warning.
/WX	-g1	Makes all warnings fatal. No object files are generated if warning occurs. (The -g option takes the form -gn , where <i>n</i> is the limit to number of warnings.)
/Za	-A	Enforces ANSI compatibility. Use only ANSI keywords. No vendor-specific extension allowed.
/Zd	/y	Generates line numbers for symbolic debugger.
/Ze	-A-, -AT	Enable vendor-specific extensions.
/Zi	/v	For Microsoft, generates debugger information for CodeView. For Turbo C++, generates information for IDE debugger and Turbo Debugger.
/Zpn	(See -a, -a-)	Packs structure members on the <i>n</i> byte boundary. <i>n</i> can be 1, 2, or 4.

Command-line options and libraries

The C0Fx.OBJ modules are provided for compatibility with source files intended for compilers from other vendors. The C0Fx.OBJ modules substitute for the C0x.OBJ modules. These initialization modules are written to alter the memory model such that the stack segment is inside the data segment. The appropriate C0Fx.OBJ module will be used automatically if you use either the **-Fs** or the **-Fm** command-line compiler option.

The **-Fc** (generate COMDEFs), **-Ff** (create far variables), **-Fs** (assume DS == SS in all models), and **-Fm** (enable all **-Fx** options) command-line compiler options are provided for compatibility. These options are fully documented in Chapter 8.

Linker

The Turbo C++ linker, TLINK, is invoked automatically from the command-line compiler unless the **-c** compiler option is used. Options such as memory model are passed from the compiler to

TLINK; TLINK links the appropriate libraries based on the compile options.

The following table compares options common to both TLINK and LINK. If an option is in one product, but not the other, it is omitted. Note that Turbo C++ TLINK options are case-sensitive, while Microsoft TLINK options are not.

Table 17.2: LINK and TLINK options compared

Microsoft C 6.0 Link option	Turbo C++ TLINK option	What it does
/CO	/v	Include full symbolic debug information.
/DOSSEG	(See comment)	For assembly programs, forces a certain ordering of segments in executable. To enable DOSSEG for an assembly program, include DOSSEG in the source code.
/F	By default	For LINK, tells linker to optimize far calls to procedures in same segment as caller. (Used with MS /PACKCODE option.) TLINK optimizes far calls automatically.
/HE	/?	Provides help on command-line options.
/LI	/l	Include source line numbers and associated addresses in map file.
/M	/m	Create map file with public global symbols.
/NOD[:filename]	/n	Don't use default libraries.
/NOE	/e	Ignore Extended Dictionary.
/NOI	/c	Treat case as significant in symbols.
/NOP	/P-	Turn off code packing.
/PACKC[:number]	/P=n	Pack code segments. <i>number</i> or <i>n</i> specifies maximum size of groups formed by /PACKC or /P .
/T	/t	Produce .COM files.

Source-level compatibility

The following sections tell you how to make sure that your code is compatible with Turbo C++'s compiler and linker.

`__MSC` macro

The Turbo C++ libraries contain many functions to increase compatibility with applications originally written in Microsoft C. If you define the macro `__MSC` before you include the `dos.h` header file, the `DOSERROR` structure will be defined to match Microsoft's format.

Header files

Some nonstandard header files can be included by one of two names, as follows.

Original name	Alias
alloc.h	malloc.h
dir.h	direct.h
mem.h	memory.h

If you are defining data in header files in your program, you should use the **-Fc** command-line compiler option or Options | Compiler | Advanced code generation | Generate COMDEFs IDE option to generate COMDEFs. Otherwise you will get linker errors. Chapter 8 provides a complete reference to the command-line compiler options.

Memory models

Although the same names are used for the standard memory models, there are fairly significant differences for the large data models in the standard configuration.

In Microsoft C, all large data models have a default NEAR data segment to which DS is maintained. Data is allocated in this data segment if the data size falls below a certain threshold, or in a far data segment otherwise. You can set the threshold value with the **/Gtn** option, where *n* is a byte value. The default threshold is 32,767. If **/Gt** is given but *n* is not specified, the default is 256.

In all other memory models under Microsoft C, both a near and a far heap are maintained.

In Turbo C++, the large and compact models (but not huge) have a default NEAR data segment to which DS is maintained. All static data is allocated to this segment by default, limiting the total static data in the program to 64K, but making all external data references near. In the huge model all data is far.

In Microsoft's version of the huge memory model, a default data segment for the entire program is maintained which limits total near data to 64K. No limit is imposed on array sizes since all extern arrays are treated as huge (**_huge**).

In Turbo C++'s huge memory model, each module has its own data segment. The data segment is loaded on function entry. All data defined in a module is referenced as near data and all external data references are far. The huge model is limited to 64K of near data in each module.

Keywords

Turbo C++ supports the same set of keywords as Microsoft C 5.1 with the exception of **fortran**.

Turbo C++ supports the same set of keywords as Microsoft C 6.0 with the exception of:

- **_based**, **_self**, and **_segname**, because Turbo C++ does not support based pointers
- **_segment**; Turbo C++'s keyword **_seg** is the equivalent of **_segment**
- **_emit**; Turbo C++ uses the pseudofunction **__emit__**, because this style allows addresses of variables to be given as arguments, and allows multiple bytes to be output; **_emit**, by contrast, works like an assembly DB, allowing one immediate byte to be output
- **_fortran**; use the **_pascal** calling convention instead

Turbo C++ provides **_cs**, **_ds**, **_es**, and **_ss** pointer types. See the section "Mixed model programming: Addressing modifiers" in Chapter "18" for more information.

Floating-point return values

In Microsoft C, **_cdecl** causes float and double values to be returned in the **__fac** (floating point accumulator) global variable. Long doubles are returned on the NDP stack. **_fastcall** causes floating point types to be returned on the NDP stack. **_pascal** causes the calling program to allocate space on the stack and pass address to function. The function stores the return value and returns the address.






In Turbo C++, floating point values are returned on the NDP stack.

Structures returned by value

In a Microsoft C-compiled function declared with `_cdecl`, the function returns a pointer to a static location. This static location is created on a per-function basis. For a function declared with `_pascal`, the calling program allocates space on the stack for the return value. The calling program passes the address for the return value in a hidden argument to the function.

Turbo C++ returns 1-byte structures in AL, 2-byte structures in AX and 4-byte structures in AX and DX. For 3-byte structures and structures larger than 4 bytes, the compiler passes a hidden argument (a far pointer) to the function that tells the function where to return the structure.

Conversion hints

-  Write *portable* code. Portable code is compatible with many different compilers and machines. Whenever possible, use only functions from the ANSI standard library (for example, use **time** instead of **gettime**). The portability sections in online Help will tell you if a function is ANSI standard.
-  If you must use a function that's not in the ANSI standard library, use a Unix-compatible function, if possible (for example, use **chmod** instead of `_chmod`, or **signal** instead of `ctrlbrk`). Again, the portability sections in online Help will tell you if a function is available on Unix machines.
-  Avoid the use of bit fields and code that depends on word size, structure alignment, or memory model. For example, Turbo C++ defines **ints** to be 16 bits wide, but a 32-bit C++ compiler would define 32-bit wide **ints**.
-  Insert the preprocessor statement `"#define _MSC"` in each module before `dos.h` is included.
-  If you were using the link option `/STACK:n` in your Microsoft application, initialize the global variable `_stklen` with the appropriate stack size.

Memory management

This chapter covers the following topics:

- What to do when you receive **Out of memory errors**.
- What **Memory models** are, how to choose one, and why you would (or would not) want to use a particular memory model.
- How **Overlays** work, and how to use them.

Running out of memory

Turbo C++ does not generate any intermediate data structures to disk when it is compiling (Turbo C++ writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message "Out of memory..." if there is not enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions. Deeply nested header files might cause this problems, also.

Memory models

Turbo C++ gives you six memory models, each suited for different program and code sizes. Each memory model uses

See page 578 for a summary of each memory model.

memory differently. What do you need to know to use memory models? To answer that question, we have to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; an 80286, 80386, or 80486. For now, we'll just refer to it as an iAPx86.

The iAPx86 registers

These are some of the registers found in the iAPx86 processor. There are other registers—but they can't be accessed directly, so they're not shown here.

Figure 18.1
iAPx86 registers

General-purpose registers	
AX	accumulator (math operations)
	AH AL
BX	base (indexing)
	BH BL
CX	count (indexing)
	CH CL
DX	data (holding data)
	DH DL

Segment address registers	
CS	code segment pointer
DS	data segment pointer
SS	stack segment pointer
ES	extra segment pointer

Special-purpose registers	
SP	stack pointer
BP	base pointer
SI	source index
DI	destination index

General-purpose registers

The general-purpose registers are the ones used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.
- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers

The iAPx86 also has some special-purpose registers:

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Turbo C++ for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

Turbo C++ functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP points to the saved previous BP value if there is a stack frame. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off*.

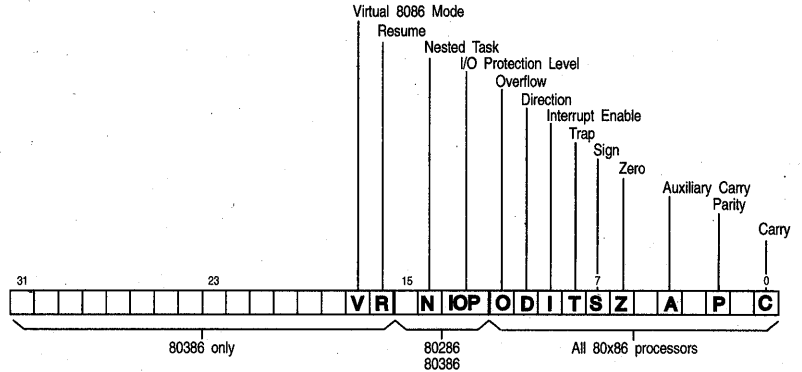
Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to variables.

The flags register

The 16-bit flags register contains all pertinent information about the state of the iAPx86 and the results of recent instructions.

For example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the *carry* and *overflow flags*, similarly report the results of arithmetic and logical operations.

Figure 18.2
Flags register of the iAPx86



Other flags control modes of operation of the iAPx86. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD**, **STI**, and **CMC**) and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ**, **RCR**, and **MOVS**. The flags register is not really used as a storage location, but rather holds the status and control data for the iAPx86.

Memory segmentation

The Intel iAPx86 microprocessor has a *segmented memory architecture*. It has a total address space of 1 MB, but it is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase, "segmented memory architecture."

- The iAPx86 keeps track of four different segments: *code*, *data*, *stack*, and *extra*. The code segment is where the machine instructions are; the data segment, where information is, such as global and static data; the stack is where function addresses and local variables are pushed; and the extra segment is also used for extra data.

- The iAPx86 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.
- A segment can be located anywhere in memory—at least, almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that's evenly divisible by 16 (in base 10).

Address calculation

A complete address on the iAPx86 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment; how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

DS register (shifted):	0010 1111 1000 0100 0000	=	2F840
Offset:	0000 0101 0011 0010	=	00532
Address:	0010 1111 1101 0111 0010	=	2FD72

A chunk of 16 bytes is known as a paragraph, so you could say that a segment always starts on a paragraph boundary.

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means—as we said—that segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
```

0010:0023

0012:0003

Segments can overlap (but don't have to). For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that's all the space you'd have for your code, your data, and your stack.

Pointers

Although you can declare a pointer or function to be a specific type regardless of the model used, by default the type of memory model you choose determines the default type of pointers used for code and data. Pointers come in four flavors: *near* (16 bits), *far* (32 bits), *huge* (also 32 bits), and *segment* (16 bits).

Near pointers A near pointer (16-bits) relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains only an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far pointers A far pointer (32-bits) contains not only the offset within the segment, but also the segment address (as another 16-bit value), which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; that, in turn, allow you to have programs larger than 64K. You can also address more than 64K of data. The use of far pointers allows access to code or data residing outside of the current segment.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) . . .
if (b == c) . . .
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the `>`, `>=`, `<`, and `<=` operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) . . .
if (b > c) . . .
if (a > c) . . .
```

The equals (`==`) and not-equal (`!=`) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (`<=`, `>=`, `<`, and `>`) use just the offset.

The `==` and `!=` operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

Important!

If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge pointers

Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, they are *normalized* to avoid the problems associated with far pointers.

What is a normalized pointer? It is a 32-bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

There are three reasons why it is important to always keep huge pointers normalized.

1. For any given memory address there is only one possible huge address—segment:offset pair. That means that the `==` and `!=` operators return correct answers for any huge pointers.
2. In addition, the `>`, `>=`, `<`, and `<=` operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results of these comparisons will be correct also.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **structs** that's larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

The six memory models

Use this model when memory is at an absolute premium.

This is a good size for average applications.

Turbo C++ gives you six memory models: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

Tiny. As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the `/t` option.

Small. The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

Best for large programs without much data in memory.

Best if code is small but needs to address a lot of data.

Large and huge are needed only for very lengthy applications.

Medium. Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MB.

Compact. The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1 MB range.

Large. Far pointers are used for both code and data, giving both a 1 MB range.

Huge. Far pointers are used for both code and data. Turbo C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

Figures 18.3 through 18.8 show how memory in the iAPx86 is apportioned for the Turbo C++ memory models. To select these memory models, you can either use menu selections from the IDE, or you can type options invoking the command-line compiler version of Turbo C++.

Figure 18.3
Tiny model memory segmentation

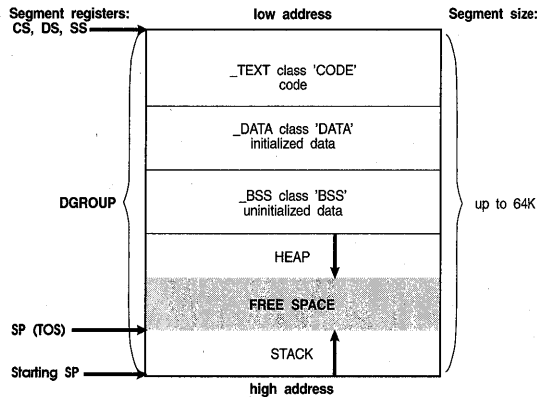


Figure 18.4
Small model memory segmentation

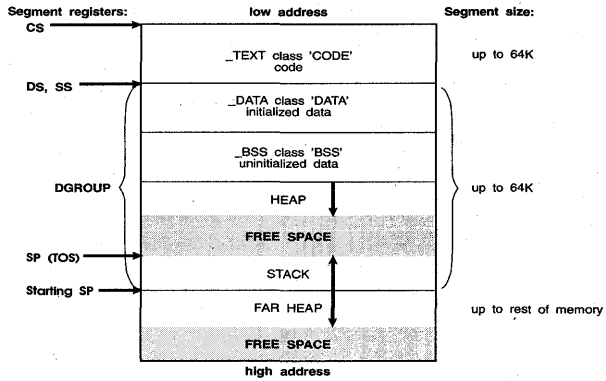


Figure 18.5
Medium model memory segmentation

CS points to only one sfile at a time

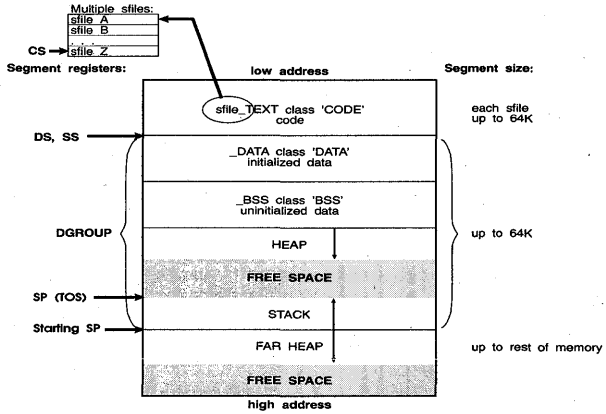


Figure 18.6
Compact model memory segmentation

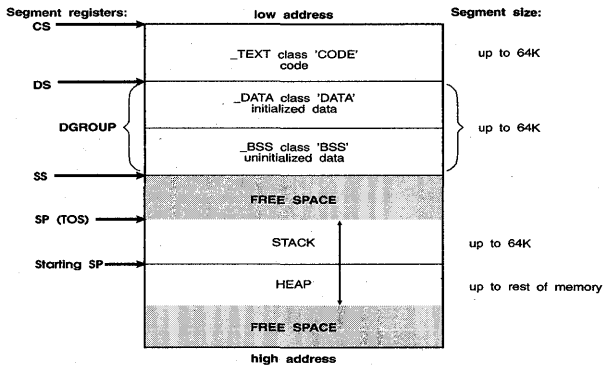


Figure 18.7
Large model memory
segmentation

CS points to only one sfile at a time

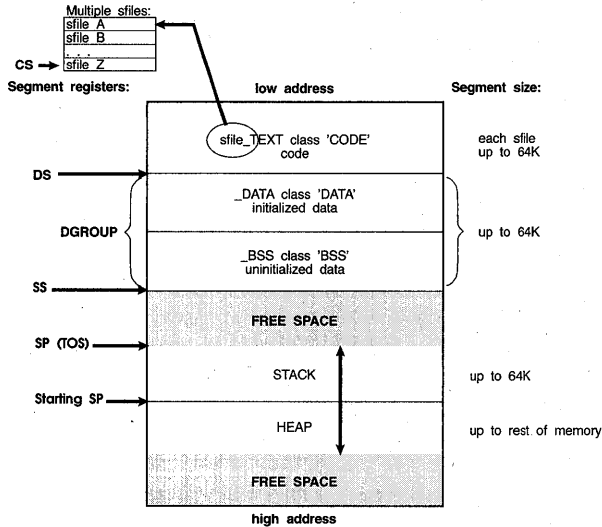
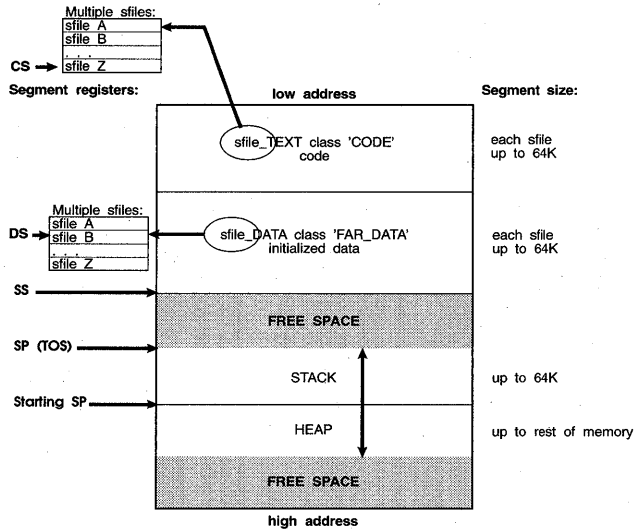


Figure 18.8
Huge model memory
segmentation

CS and DS point to only one sfile at a time



The following table summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (1 MB); these groups correspond to the rows and columns in Table 8.

Table 18.1
Memory models

The models tiny, small, and compact are small code models because, by default, code pointers are near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

Data size	Code size	
	64K	1 MB
64K	Tiny (data, code overlap; total size = 64K)	
	Small (no overlap; total size = 128K)	Medium (small data, large code)
1 MB	Compact (large data, small code)	Large (large data, code)
		Huge (same as large but static data > 64K)

Important!

When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code models (medium, large, or huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in *each* module.

Mixed-model programming: Addressing modifiers

Turbo C++ introduces eight new keywords not found in standard ANSI C (**near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_ss**, and **_seg**) that can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Turbo C++, you can modify the declarations of pointers, objects, and functions with the keywords **near**, **far**, or **huge**. We explained **near**, **far**, and **huge** data pointers earlier in this chapter. You can declare far objects using the **far** keyword. **near** functions are invoked with near calls and exit with near returns. Similarly, **far** functions are called **far** and do far returns. **huge** functions are like **far** functions, except that **huge** functions set DS to a new value, while **far** functions do not.

There are also four special **near** data pointers: **_cs**, **_ds**, **_es**, and **_ss**. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

then *p* would contain a 16-bit offset into the stack segment.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The next table shows just how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 MB memory space (far, has its own segment address).

Table 18.2
Pointer results

Memory model	Function pointers	Data pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Segment pointers

Use **_seg** in segment pointer type declarators. The resulting pointers are 16-bit segment pointers. The syntax for **_seg** is:

```
datatype _seg *identifier;
```

For example,

```
int _seg *name;
```

Any indirection through *identifier* has an assumed offset of 0. In arithmetic involving segment pointers the following rules hold true:

1. You can't use the **++**, **--**, **+=**, or **-=** operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. When adding a near pointer to a segment pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer.

Therefore, the two pointers must either point to the same type, or one must be a pointer to void. There is no multiplication of the offset regardless of the type pointed to.

4. When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
5. When adding or subtracting an integer operand to or from a segment pointer, the result is a far pointer, with the segment taken from the segment pointer and the offset found by multiplying the size of the object pointed to by the integer operand. The arithmetic is performed as if the integer were added to or subtracted from the far pointer.
6. Segment pointers can be assigned, initialized, passed into and out of functions, compared and so forth. (Segment pointers are compared as if their values were **unsigned** integers.) In other words, other than the above restrictions, they are treated exactly like any other pointer.

Declaring far objects

You can declare far objects in Turbo C++. For example,

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

The command-line compiler options **-zE**, **-zF**, and **-zH** (which can also be set using **#pragma option**) affect the far segment name, class, and group, respectively. When you change them with **#pragma option**, you can change them at any time and they apply to any ensuing far object declarations. Thus you could use the following sequence to create a far object in a specific segment:

```
#pragma option -zEsegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH* -zF*
```

This will put *x* in segment MYSEGMENT 'MYCLASS' in the group 'MYGROUP', then reset all of the far object items to the default values. Note that by using these options, several far objects can be forced into a single segment:

```
#pragma option -zEcombined -zFmyclass
int far x;
double far y;
#pragma option -zE* -zF*
```

Both x and y will appear in the segment COMBINED 'MYCLASS' with no group.

Declaring functions to be near or far

On occasion, you'll want (or need) to override the default function type of your memory model shown in Table 8 (page 273).

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp)
{
    if (exp <= 0)
        return(1);
    else
        return(x * power(x, exp-1));
}
```

Every time **power** calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring **power** as **near**, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double near power(double x,int exp)
```

This guarantees that **power** is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you are using a large code model (medium, large, or huge), you can only call **power** from within the module where it is defined. Other modules have their own code segment and thus cannot call **near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the **power** example. It is wise to also declare **power** as static, since it should only be called from within the current module. That way, being a static, its name will not be available to any functions outside the module.

Declaring pointers to be near, far, or huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. Why might you want to do the same thing for pointers? For the same reasons given in the preceding section: either to avoid unnecessary overhead (declaring **near** when the default would be **far**) or to reference something outside of the default segment (declaring **far** or **huge** when the default would be **near**).

Pointing to a given segment:offset address

How do you make a far pointer point to a given memory location (a specific segment:offset address)? You can use the macro **MK_FP**, which takes a segment and an offset and returns a far pointer. For example,

```
MK_FP(segment_value, offset_value)
```

Given a **far** pointer, *fp*, you can get the segment component with **FP_SEG(fp)** and the offset component with **FP_OFF(fp)**. For more information about these three Turbo C++ library routines, refer to the online Help.

Using library files

Turbo C++ offers a version of the standard library routines for each of the six memory models. Turbo C++ is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Turbo C++ linker, **TLINK**, directly (as a standalone linker), you need to specify which libraries to use. See Chapter 10, "TLINK: The Turbo linker" for details on how to do so.

Linking mixed modules

What if you compiled one module using the small memory model, and another module using the large model, then wanted to link them together? What would happen?

The files would link together fine, but the problems you would encounter would be similar to those described in the earlier section, "Declaring functions to be near or far." If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as de-

scribed in the earlier section, “Declaring pointers to be near, far, or huge,” since a function in the small module would expect to pass and receive **near** pointers, while a function in the large module would expect **far** pointers.

The solution is to properly declare the function to be near or far in its prototype.

What if you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **far**. To do this, make a copy of each header file you would normally include (such as `stdio.h`), and rename the copy to something appropriate (such as `fstdio.h`).

Then edit each function prototype in the copy so that it is explicitly **far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **far** calls be made to the routines, but the pointers passed will be **far** pointers as well. Modify your program so that it includes the new header file:

```
#include <fstdio.h>

main()
{
    char near *mystr;
    mystr = "Hello, world\n";
    printf(mystr);
}
```

Compile your program with the command-line compiler TCC then link it with TLINK, specifying a large model library, such as CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Overlays (VROOMM)

Overlays are parts of a program’s code that share a common memory area. Only the parts of the program that are required for a given function reside in memory at the same time.

Overlays can significantly reduce a program’s total run-time memory requirements. With overlays, you can execute programs

that are much larger than the total available memory, since only parts of the program reside in memory at any given time.

How overlays

work

Turbo C++'s overlay manager (called VROOMM for Virtual Runtime Object-Oriented Memory Manager) is highly sophisticated; it does much of the work for you. In a conventional overlay system, modules are grouped together into a base and a set of overlay units. Routines in a given overlay unit may call other routines in the same unit and routines in the base, but not routines in other units. The overlay units are overlaid against each other; that is, only one overlay unit may be in memory at a time, and they each occupy the same physical memory. The total amount of memory needed to run the program is the size of the base plus the size of the largest overlay.

This conventional scheme is quite inflexible. It requires complete understanding of the possible calling dependencies in the program, and requires you to have the overlays grouped accordingly. It may be impossible to break your program into overlays if you can't split it into separable calling dependencies.

VROOMM's scheme is quite different. It provides *dynamic segment swapping*. The basic swapping unit is the segment. A segment can be one or more modules. More importantly, any segment can call *any other segment*.

Memory is divided into an area for the base plus a swap area. Whenever a function is called in a segment that is neither in the base nor in the swap area, the segment containing the called function is brought into the swap area, possibly displacing other segments. This is a powerful approach—it is like software virtual memory. You no longer have to break your code into static, distinct, overlay units. You just let it run!

What happens when a segment needs to be brought into the swap area? If there is room for the segment, execution just continues. If there is not, then one or more segments in the swap area must be thrown out to make room. How to decide which segment to throw out? The actual algorithm is quite sophisticated. A simplified version: If there is an inactive segment, choose it for removal. Inactive segments are those without executing functions. Otherwise, pick an active segment and toss it out. Keep tossing out segments until there is enough room available. This technique is called *dynamic swapping*.

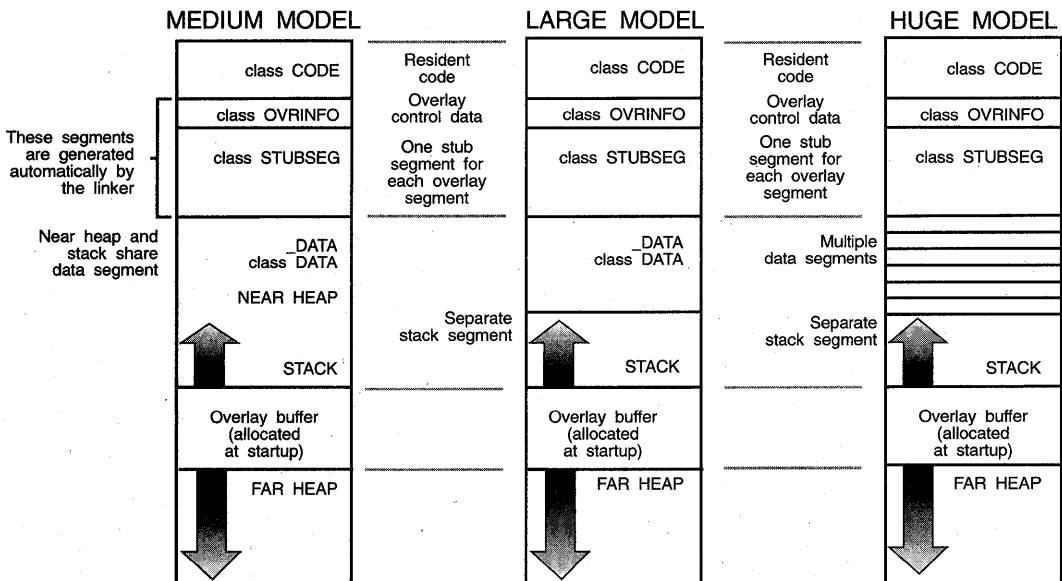
The more memory you provide for the swap area, the better the program performs. The swap area acts like a cache; the bigger the cache, the faster the program runs. The best setting for the size of the swap area is the size of the program's *working set*.

Once an overlay is loaded into memory, it is placed in the overlay buffer, which resides in memory between the stack segment and the far heap. By default, the size of the overlay buffer is estimated and set at startup, but you can change it using the global variable `_ovrbuffer` (see page 592). If enough memory isn't available, an error message will be displayed by DOS ("Program too big to fit in memory") or by the C startup code ("Not enough memory to run program").

One very important option of the overlay manager is the ability to swap the modules to expanded or extended memory when they are discarded from the overlay buffer. Next time the module is needed, the overlay manager can copy it from where the module was swapped to instead of reading from the file. This makes it much faster.

When using overlays, memory is used as shown in the next figure.

Figure 18.9: Memory maps for overlays



Getting the best out of Turbo C++ overlays

See page 592 for more information on setting the size of the overlay buffer.

To get the best out of Turbo C++ overlays,

- Minimize resident code (resident run-time library, interrupt handlers, and device drivers is a good starting point).
- Set overlay buffer size to be a comfortable working set (start with 128K and adjust up and down to see the speed/size tradeoff).
- Think versatility and variety: Take advantage of the overlay system to provide support for special cases, interactive help, and other end-user benefits you could not consider before.

Requirements

In order to create overlays, you'll need to remember a few simple rules,

- The smallest part of a program that can be made into an overlay is a segment.
- Overlaid applications must use the medium, large, or huge programming models; the tiny, small, and compact models are not supported.
- Normal segment merging rules govern overlaid segments. That is, several .OBJ modules can contribute to the same overlaid segment.

The link-time generation of overlays is completely separated from the run-time overlay management; the linker does *not* automatically include code to manage the overlays. In fact, from the linker's point of view, the overlay manager is just another piece of code that gets linked in. The only assumption the linker makes is that the overlay manager takes over an interrupt vector (typically INT 3FH) through which all dynamic loading is controlled. This level of transparency makes it very easy to implement custom-built overlay managers that suit the particular needs of each application.

Using overlays

To overlay a program, all of its modules must be compiled with the **-Y** compiler option enabled. To make a particular module into an overlay, it needs to be compiled with the **-Yo** option. (**-Yo** automatically enables **-Y**.)

The **-Yo** option applies to all modules and libraries that follow it on the command line; you can disable it with **-Yo-**. These are the only command line options that are allowed to follow file names. For example, to overlay the module OVL.C but not the library GRAPHICS.LIB, either of the following command lines could be used:

```
TCC -ml -Yo ovl.c -Yo- graphics.lib
```

or

```
TCC -ml graphics.lib -Yo ovl.c
```

If TLINK is invoked explicitly to link the .EXE file, the **/b** linker option must be specified on the linker command line or response file. See Chapter 10, "TLINK: The Turbo linker for details on how to use the **/b** option.

Overlay example

Suppose that you want to overlay a program consisting of three modules: MAIN.C, O1.C, and O2.C. Only the modules O1.C and O2.C should be made into overlays. (MAIN.C contains time-critical routines and interrupt handlers, so it should stay resident.) Let's assume that the program uses the large memory model.

The following command accomplishes the task:

```
TCC -ml -Y main.c -Yo o1.c o2.c
```

The result will be an executable file MAIN.EXE, containing two overlays.

Overlaying in the IDE

To overlay modules in the IDE, you must take the following steps:

1. Select Options | Application | Overlay
2. Select Project | Local Options to specify each module that needs to go into an overlay.

Selecting Options | Application | Overlay will also select the following options automatically for you:

- Options | Compiler | Entry/Exit Code | Overlay
- Options | Linker | Settings | Output | Overlaid EXE
- Project | Local options | Overlay this module

- Options | Compiler | Code generation | Model | Medium
- Options | Compiler | Code generation | Assume SS Equals DS | Default for memory model
- Options | Linker | Libraries | Graphics library

➡ If you are building an .EXE file containing overlays, compile all modules after selecting DOS Overlay from the Options | Application dialog box.

➡ No module going into an overlay should ever change the default Code Class name. The IDE lets you change the set of modules residing in overlays without having to worry about recompiling. This can only be accomplished (with current .OBJ information) if overlays keep default code class names.

Overlaid programs

This section discusses issues vital to well-behaved overlaid applications.

The far call requirement

Use a large code model (medium, large, or huge) when you want to compile an overlay module. At any call to an overlaid function in another module, you *must* guarantee that all currently active functions are far.

You *must* compile all overlaid modules with the **-Y** option, which makes the compiler generate code that can be overlaid.

Important!

Failing to observe the far call requirement in an overlaid program will cause unpredictable and possibly catastrophic results when the program is executed.

Buffer size

The default overlay buffer size is twice the size of the largest overlay. This is adequate for some applications. But imagine that a particular function of a program is implemented through many modules, each of which is overlaid. If the total size of those modules is larger than the overlay buffer, a substantial amount of swapping will occur if the modules make frequent calls to each other.

The solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. You can do this by setting the `_ovrbuffer` global variable to the required size in para-

graphs. For example, to set the overlay buffer to 128K, include the following statement in your code:

```
unsigned _ovrbuffer = 0x2000;
```

There is no general formula for determining the ideal overlay buffer size. Borland's Turbo Profiler, available separately, can help provide a suitable value.

What not to overlay

Don't overlay modules that contain interrupt handlers, or small and time-critical routines. Due to the non-reentrant nature of the DOS operating system, modules that may be called by interrupt functions should not be overlaid.

Turbo C++'s overlay manager fully supports passing overlaid functions as arguments, assigning and initializing function pointer variables with addresses of overlaid functions, and calling overlaid routines via function pointers.

Debugging overlays

Most debuggers have very limited overlay debugging capabilities, if any at all. Not so with Turbo C++'s integrated debugger and Turbo Debugger, the standalone debugger, which is available separately. Both debuggers fully support single-stepping and breakpoints in overlays in a manner completely transparent to you. By using overlays, you can easily engineer and debug huge applications—all from inside the IDE or by using Turbo Debugger.

External routines in overlays

Like normal C functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid functions, the assembly language routine *must* be declared FAR, and it *must* set up a stack frame using the BP register. For example, assuming that *OtherFunc* is an overlaid function in another module, and that the assembly language routine *ExternFunc* calls it, then *ExternFunc* must be FAR and set up a stack frame, as shown:

```
ExternFunc    PROC    FAR
              push    bp                ;Save BP
              mov     bp,sp             ;Set up stack frame
              sub     sp,LocalSize      ;Allocate local variables
              :
              call   OtherFunc         ;Call another overlaid module
              :
```

```

        mov     sp, bp           ;Dispose local variables
        pop     bp             ;Restore BP
        RET                    ;Return
ExternFunc      ENDP

```

where *LocalSize* is the size of the local variables. If *LocalSize* is zero, you can omit the two lines to allocate and dispose local variables, but you must not omit setting up the BP stack frame even if you have no arguments or variables on the stack.

These requirements are the same if *ExternFunc* makes *indirect* references to overlaid functions. For example, if *OtherFunc* makes calls to overlaid functions, but is not itself overlaid, *ExternFunc* must be FAR and still has to set up a stack frame.

In the case where an assembly language routine doesn't make any direct or indirect references to overlaid functions, there are no special requirements; the assembly language routine can be declared NEAR. It does not have to set up a stack frame.

Overlaid assembly language routines should *not* create variables in the code segment, since any modifications made to an overlaid code segment are lost when the overlay is disposed. Likewise, pointers to objects based in an overlaid code segment cannot be expected to remain valid across calls to other overlays, since the overlay manager freely moves around and disposes overlaid code segments.

Swapping

If you have expanded or extended memory available, you can tell the overlay manager to use it for swapping. If you do so, when the overlay manager has to discard a module from the overlay buffer (because it should load a new module and the buffer is full), it can store the discarded module in this memory. Any later loading of this module is reduced to in-memory transfer, which is significantly faster than reading from a disk file.

In both cases there are two possibilities: The overlay manager can either detect the presence of expanded or extended memory and can take it over by itself, or it can use an already detected and allocated portion of memory. For extended memory, the detection of the memory use is not always successful because of the many different cache and RAM disk programs that can take over extended memory without any mark. To avoid this problem, you can tell the overlay manager the starting address of the extended memory and how much of it is safe to use.

Expanded memory The **_OvrInitEms** function initializes expanded memory swapping. Here's its prototype:

_OvrInitEms and **_OvrInitExt**
are defined in *dos.h*.

```
int cdecl far _OvrInitEms
(
    unsigned __emsHandle,
    unsigned __emsFirst,
    unsigned __emsPages
);
```

If the *emsHandle* parameter is zero, the overlay manager checks for the presence of expanded memory and allocates the amount (if it can) that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *emsHandle* should be a legal EMS handle, *emsFirst* is the first usable EMS page, and *emsPages* is the number of pages usable by the overlay manager. This function returns 0 if expanded memory is available.

Extended memory The **_OvrInitExt** function initializes extended memory swapping. Here's its prototype:

```
int cdecl far _OvrInitExt
(
    unsigned long __extStart,
    unsigned long __extLength
);
```

If the *extStart* parameter is zero, the overlay manager checks for extended memory. If it can, the overlay manager uses the amount of free memory that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *extStart* is the start of the usable extended memory, with *extLength* bytes usable by the overlay manager. If *extlength* is zero, the overlay manager will use all available extended memory above *extStart*. This function returns 0 if extended memory is available. **_OvrInitExt** is defined in *dos.h*.

Important!

The use of extended memory is not standardized. Though the overlay manager tries every known method to find out the amount of extended memory which is already used, use this function carefully. For example, if you have a 2 MB hard disk cache program installed (that uses extended memory), you could use the following call to let the overlay manager use the remaining extended memory:

```
if (_OvrInitExt (1024L * (2048 + 1024), 0L))  
    puts ("No extended memory available for overlay swapping");
```

Mathematical operations

This chapter covers the floating-point options and explains how to use complex math.

Floating-point options

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor is set up to easily handle integer values, but it takes more time and effort to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

If you have an 80486 processor, the numeric coprocessor is probably already built in.

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

Emulating the 80x87 chip

This software resides in the last 512 bytes of your stack, so make allowance for it when using the emulation option and set your stack size accordingly.

The default Turbo C++ code generation option is *emulation* (the `-f` command-line compiler option). This option is for programs that may or may not have floating point, and for machines that may or may not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it will use the 80x87 if it is present; if no coprocessor is present at run time, it uses special software that *emulates* the 80x87.

Using 80x87 code

If your program is *only* going to run on machines with an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Simply choose the 80x87 floating-point code generation option (the `-f87` command-line compiler option). Turbo C++ will then link your programs with FP87.LIB instead of EMU.LIB.

No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code generation option (the `-f-` command-line compiler option). Then Turbo C++ will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

Fast floating-point option

Turbo C++ has a fast floating-point option (the `-ff` command-line compiler option). It can be turned off with `-ff-` on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;  
x = (float)(3.5*x);
```

To execute this correctly, *x* is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in *x*. Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend

on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

The 87 environment variable

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

There are some situations in which you might want to override this default autodetection behavior. For example, your own runtime system might have an 80x87, but you need to verify that your program will work as intended on systems without a coprocessor. Or your program may need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

Turbo C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces to either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.

Setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. Let the programmer beware!! If you set 87 = Y when, in fact, there is no 80x87 available on that system, your system will hang.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press *Enter* immediately after typing the equal sign.

Registers and the 80x87

There are a couple of points concerning registers that you should be aware of when using floating point.

1. In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
2. If you are mixing floating point with inline assembly, you may need to take special care when using 80x87 registers. You might need to pop and save the 80x87 registers before calling functions that use the coprocessor, unless you are sure that enough free registers exist.

Disabling floating-point exceptions

By default, Turbo C++ programs abort if a floating-point overflow or divide by zero error occurs. You can mask these floating-point exceptions by a call to **_control87** in **main**, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
    _control87(MCW_EM,MCW_EM);
    ...
}
```

You can determine whether a floating-point exception occurred after the fact by calling **_status87()** or **_clear87()**.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN will likely cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of **matherr()** into your program.

```
#include <math.h>
int cdecl matherr(struct exception *e)
{
    return 1;          /* error has been handled */
}
```

Any other use of **matherr()** to intercept math errors is not encouraged, as it is considered obsolete and may not be supported in future versions of Turbo C++.

Using complex math

Complex numbers are numbers of the form $x + yi$, where x and y are real numbers, and i is the square root of -1 . Turbo C++ has always had a type

```
struct complex
{
    double x, y;
};
```

defined in `math.h`. This type is convenient for holding complex numbers, as they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

*See the description of class **complex** in online Help for more information.*

To use complex numbers in C++, all you have to do is to include `complex.h`. In `complex.h`, all the following have been overloaded to handle complex numbers:

- all of the usual arithmetic operators
- the stream operators, `>>` and `<<`
- the usual math functions, such as **sqrt()** and **log()**

The complex library is invoked only if the argument is of type **complex**. Thus, to get the complex square root of -1 , use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

As an example of the use of complex numbers, the following function computes a complex Fourier transform.

```
#include <complex.h>

// calculate the discrete Fourier transform of a[0], ..., a[n-1].
void Fourier(int n, complex a[], complex b[])
{
    int j, k;
    complex i(0,1); // square root of -1
```

```

for (j = 0; j < n; ++j)
{
    b[j] = 0;
    for (k = 0; k < n; ++k)
        b[j] += a[k] * exp(2*M_PI*j*k*i/n);
    b[j] /= sqrt(n);
}
}

```

Using BCD math

Turbo C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This is sometimes confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.

Binary numbers are preferable for most applications, but in some situations the roundoff error involved in converting between base 2 and 10 is undesirable. The most common case is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```

#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n", x);

```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny roundoff error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Turbo C++ offers the C++ type **bcd**, which is declared in `bcd.h`. With **bcd**, the number 0.01 is represented exactly, and the **bcd** variable *x* will give an exact penny count.

```

#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;

```

```
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";
```

Here are some facts to keep in mind about **bcd**.

- **bcd** does not eliminate all roundoff error: A computation like $1.0/3.0$ will still have roundoff error.
- The usual math functions, such as **sqrt** and **log**, have been overloaded for **bcd** arguments.
- BCD numbers have about 17 decimal digits precision, and a range of about 1×10^{-125} to 1×10^{125} .

Converting BCD numbers

bcd is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is only performed when at least one operand is of the type **bcd**.

Important!

The **bcd** member function **real** is available for converting a **bcd** number back to one of the usual base 2 formats (**float**, **double**, or **long double**), though the conversion is not done automatically. **real** does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example,

```
bcd a = 12.1;
```

can be printed using any of the following four lines of code:

```
double x = a; printf("a = %g", x);
printf("a = %Lg", real(a));
printf("a = %g", (double)real(a));
cout << "a = " << a;
```

Note that since **printf** does not do argument checking, the format specifier must have the *L* if the **long double** value **real(a)** is passed.

Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a **bcd**. The number of places is an optional second argument to the constructor **bcd**. For example, to convert $\$1000.00/7$ to a **bcd** variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

1000.00/7	=	142.85714...
bcd(1000.00/7, 2)	=	142.860
bcd(1000.00/7, 1)	=	142.900
bcd(1000.00/7, 0)	=	143.000
bcd(1000.00/7, -1)	=	140.000
bcd(1000.00/7, -2)	=	100.000

This method of rounding is specified by IEEE.

The number is rounded using banker's rounding, which means round to the nearest whole number, with ties being rounded to an even digit. For example,

bcd(12.335, 2)	=	12.34
bcd(12.345, 2)	=	12.34
bcd(12.355, 2)	=	12.36

Video functions

Turbo C++ comes with a complete library of graphics functions, so you can produce onscreen charts and diagrams. This chapter briefly discusses video modes and windows, then explains how to program in text mode and in graphics mode.

Turbo C++'s video functions are similar to corresponding routines in Turbo Pascal. If you are already familiar with controlling your PC's screen modes or creating and managing windows and viewports, you can skip to page 607.

Some words about video modes

Your PC has some type of video adapter. This can be a Monochrome Display Adapter (MDA) for text-only display, or it can be a graphics adapter, such as a Color/Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), a Video Graphics Array adapter (VGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80 or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color or black and white).

The screen's operating mode is defined when your program calls one of the mode-defining functions **textmode**, **initgraph**, or **setgraphmode**.

- In *text mode*, your PC's screen is divided into cells (80- or 40-columns wide by 25, 43, or 50 lines high). Each cell consists of an attribute and a character. The character is the displayed ASCII character; the attribute specifies *how* the character is displayed (its color, intensity, and so on). Turbo C++ provides a full range of routines for manipulating the text screen, for writing text directly to the screen, and for controlling cell attributes.
- In *graphics mode*, your PC's screen is divided into pixels; each pixel displays a single dot onscreen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Turbo C++'s graphics library to create graphic displays onscreen: You can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper left corner of the screen is position (1,1), with x-coordinates increasing from left to right, and y-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper left corner is position (0,0), with the x- and y-coordinate values increasing in the same manner.

Some words about windows and viewports

Turbo C++ provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you are not familiar with windows and viewports, you should read this brief overview. Turbo C++'s window- and viewport-management functions are explained in "Programming in text mode" and "Programming in graphics mode" later in this chapter.

What is a window?

A window is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default full-screen text window to a text window smaller than the full screen (with a call to the **window** function). This function specifies the window's position in terms of screen coordinates.

What is a viewport?

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a viewport. When your graphics program outputs drawings and so on, the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the **setviewport** function.

Coordinates

Except for these window- and viewport-defining functions, all coordinates for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

Programming in text mode

This section briefly summarizes the text mode functions.

In Turbo C++, the direct console I/O package (**cprintf**, **cputs**, and so on) provides high-performance text output, window management, cursor positioning, and attribute control functions. These functions are all part of the standard Turbo C++ libraries; they are prototyped in the header file `conio.h`.

The console I/O functions

Turbo C++'s text-mode functions work in any of the six possible video text modes. The modes available on your system depend on the type of video adapter and monitor you have. You specify the current text mode with a call to **textmode**. We explain how to use this function later.

The text mode functions are divided into five separate groups:

- text output and manipulation
- window and mode control
- attribute control
- state query
- cursor shape

These five text mode function groups are covered in the following sections.

Text output and manipulation

Here's a quick summary of the text output and manipulation functions:

Writing and reading text:

cprintf	Sends formatted output to the screen.
cputs	Sends a string to the screen.
getche	Reads a character and echoes it to the screen.
putch	Sends a single character to the screen.

Manipulating text (and the cursor) onscreen:

clreol	Clears from the cursor to the end of the line.
clrscr	Clears the text window.
delline	Deletes the line where the cursor rests.
gotoxy	Positions the cursor.
insline	Inserts a blank line below the line where the cursor rests.

Moving blocks of text into and out of memory:

movetext	Copies text from one area onscreen to another.
gettext	Copies text from an area onscreen to memory.
puttext	Copies text from memory to an area onscreen.

Your screen-output programs will come up in a full-screen text window by default, so you can immediately write, read, and manipulate text without any preliminary mode-setting. You write text to the screen with the direct console output functions **cprintf**, **cputs**, and **putch**, and echo input with the function **getche**. Text wrapping is controlled by the global variable `_wscroll`. If `_wscroll` is 1, text wraps onto the next line, scrolling as necessary. If `_wscroll` is 0, text wraps onto the same line, and there is no scrolling. `_wscroll` is 1 by default.

Once your text is on the screen, you can erase the active window with **clrscr**, erase part of a line with **clreol**, delete a whole line with **delline**, and insert a blank line with **insline**. The latter three functions operate relative to the cursor position; you move the cursor to a specified location with **gotoxy**. You can also copy a whole block of text from one rectangular location in the window to another with **movetext**.

You can capture a rectangle of onscreen text to memory with **gettext**, and put that text back on the screen (anywhere you want) with **puttext**.

Window and mode control

There are two window- and mode-control functions:

- textmode** Sets the screen to a text mode.
- window** Defines a text-mode window.

You can set your screen to any of several video text modes with **textmode** (depending on your system’s monitor and adapter). This initializes the screen as a full-screen text window, in the particular mode specified, and clears any residual images or text.

When your screen is in a text mode, you can output to the full screen, or you can set aside a *portion* of the screen—a window—to which your program’s output is confined. To create a text window, you call **window**, specifying the onscreen area it will occupy:

Attribute control

Here’s a quick summary of the text-mode attribute control functions:

Setting foreground and background:

- textattr** Sets the foreground and background colors (attributes) at the same time.
- textbackground** Sets the background color (attribute).
- textcolor** Sets the foreground color (attribute).

Modifying intensity:

- highvideo** Sets text to high intensity.
- lowvideo** Sets text to low intensity.
- normvideo** Sets text to original intensity.

The attribute control functions set the current attribute, which is represented by an 8-bit value: The four lowest bits represent the foreground color, the next three bits give the background color, and the high bit is the “blink enable” bit.

Subsequent text is displayed in the current attribute. With the attribute control functions, you can set the background and foreground (character) colors separately (with **textbackground** and **textcolor**) or combine the color specifications in a single call to **textattr**. You can also specify that the character (the foreground) will blink. Most color monitors in color modes will display the true colors. Non-color monitors may convert some or all of the attributes to various monochromatic shades or other visual effects, such as bold, underscore, reverse video, and so on.

You can direct your system to map the high-intensity foreground colors to low-intensity colors with **lowvideo** (which turns off the high-intensity bit for the characters). Or you can map the low-intensity colors to high intensity with **highvideo** (which turns on

the character high-intensity bit). When you're through playing around with the character intensities, you can restore the settings to their original values with **normvideo**.

State query Here's a quick summary of the state-query functions:

- gettextinfo** Fills in a **text_info** structure with information about the current text window.
- wherex** Gives the x-coordinate of the cell containing the cursor.
- wherey** Gives the y-coordinate of the cell containing the cursor.

Turbo C++'s console I/O functions include some designed for *state queries*. With these functions, you can retrieve information about your text-mode window and the current cursor position within the window.

The **gettextinfo** function fills a **text_info** structure (defined in `conio.h`) with several details about the text window, including:

- the current video mode
- the window's position in absolute screen coordinates
- the window's dimensions
- the current foreground and background colors
- the cursor's current position

Sometimes you might need only a few of these details. Instead of retrieving all text window information, **wherex** and **wherey** return just the cursor's (window-relative) position.

Cursor shape The function **_setcursortype** enables you to change the appearance of your cursor. The values are **_NOCURSOR**, which turns off the cursor; **_SOLIDCURSOR**, which gives you a solid block (large) cursor; and **_NORMALCURSOR**, which gives you the normal underscore cursor.

Text windows

The default text window is full screen; you can change this to a smaller text window with a call to the **window** function. Text windows can contain up to 50 lines and up to 40 or 80 columns.

The coordinate origin (point where the numbers start) of a Turbo C++ text window is the upper left corner of the window. The

coordinates of the window's upper left corner are (1,1); the coordinates of the bottom right corner of a full-screen 80-column, 25-line text window are (80,25).

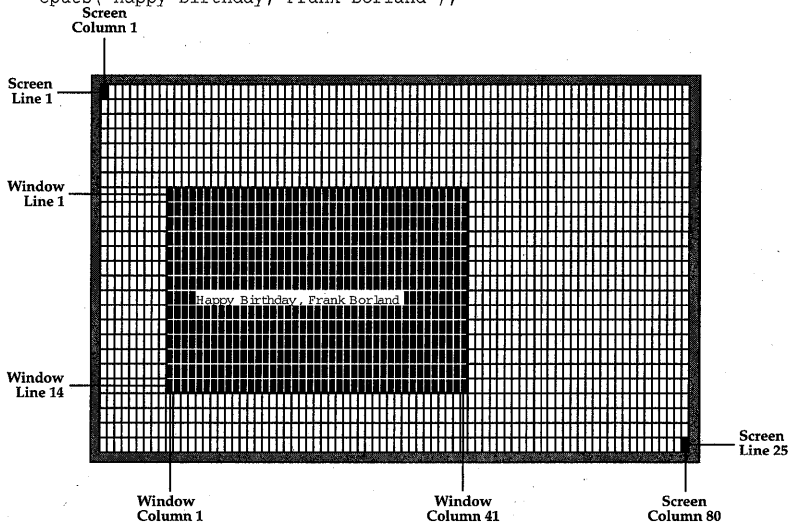
An example Suppose your 100% PC-compatible system is in 80-column text mode, and you want to create a window. The upper left corner of the window will be at screen coordinates (10, 8), and the lower right corner of the window will be at screen coordinates (50, 21). To do this, you call the **window** function, like this:

```
window(10, 8, 50, 21);
```

Now that you've created the text-mode window, you want to move the cursor to the *window* position (5, 8) and write some text in it, so you decide to use **gotoxy** and **cputs**. The following figure illustrates the code.

```
gotoxy(5, 8);  
cputs("Happy Birthday, Frank Borland");
```

Figure 20.1
A window in 80x25 text mode



The *text_modes* type

You can put your monitor into one of seven PC text modes with a call to the **textmode** function. The enumeration type *text_modes*, defined in *conio.h*, enables you to use symbolic names for the *mode* argument to the **textmode** function, instead of "raw" mode numbers. However, with symbolic constants, you must put

```
#include <conio.h>
```

in your source code.

The numeric and symbolic values defined by *text_modes* are as follows:

Symbolic constant	Numeric value	Video text mode
LASTMODE	-1	Previous text mode enabled
BW40	0	Black and white, 40 columns
C40	1	16-color, 40 columns
BW80	2	Black and white, 80 columns
C80	3	16-color, 80 columns
MONO	7	Monochrome, 80 columns
C4350	64	EGA, 80x43; VGA, 80x50 lines

For example, the following calls to **textmode** put your color monitor in the indicated operating mode:

```
textmode(0)    Black and white, 40 column
textmode(BW80) Black and white, 80 column
textmode(C40)  16-color, 40 column
textmode(3)    16-color, 80 column
textmode(7)    Monochrome, 80 columns
textmode(C4350) EGA, 80x43; VGA, 80x50 lines
```

Use **settextinfo** to determine the number of rows in the screen after calling **textmode** in the mode C4350.

Text colors

For a detailed description of how cell attributes are laid out, refer to the **textattr** entry in the online Runtime Library Reference.

When a character occupies a cell, the color of the character is the *foreground*; the color of the cell's remaining area is the *background*. Color monitors with color video adapters can display up to 16 different colors; monochrome monitors substitute different visual attributes (highlighted, underscored, reverse video, and so on) for the colors.

Symbolic constant	Numeric value	Foreground or background?
BLACK	0	Both
BLUE	1	Both
GREEN	2	Both
CYAN	3	Both
RED	4	Both
MAGENTA	5	Both
BROWN	6	Both
LIGHTGRAY	7	Both
DARKGRAY	8	Foreground only
LIGHTBLUE	9	Foreground only
LIGHTGREEN	10	Foreground only
LIGHTCYAN	11	Foreground only
LIGHTRED	12	Foreground only
LIGHTMAGENTA	13	Foreground only
YELLOW	14	Foreground only
WHITE	15	Foreground only
BLINK	128	Foreground only

The include file `conio.h` defines symbolic names for the different colors. If you use the symbolic constants, you must include `conio.h` in your source code.

Table 20 lists these symbolic constants and their corresponding numeric values. Note that only the first eight colors are available for the foreground and background; the last eight (colors 8 through 15) are available for the foreground (the characters themselves) only.

You can add the symbolic constant `BLINK` (numeric value 128) to a foreground argument if you want the character to blink.

High-performance output

Turbo C++'s console I/O package includes a variable called *directvideo*. This variable controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to the video RAM). In general, going directly to video RAM gives very high performance (spelled f-a-s-t-e-r o-u-t-p-u-t), but doing so requires your computer to be 100% IBM PC-compatible: Your video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 will work on any machine that is IBM BIOS-compatible, but the console output will be slower.

Programming in graphics mode

In this section, we give a brief summary of the functions you use in graphics mode. For more detailed information about these functions, refer to online help.

Turbo C++ provides a separate library of over 70 graphics functions, ranging from high-level calls (like **setviewport**, **bar3d**, and **drawpoly**) to bit-oriented functions (like **getimage** and **putimage**). The graphics library supports numerous fill and line styles, and provides several text fonts that you can size, justify, and orient horizontally or vertically.

These functions are in the library file GRAPHICS.LIB, and they are prototyped in the header file graphics.h. In addition to these two files, the graphics package includes graphics device drivers (*.BGI files) and stroked character fonts (*.CHR files); we discuss these additional files in following sections.

In order to use the graphics functions:

- If you're using the IDE, check **Options | Linker | Libraries | Graphics Library**. When you make your program, the linker automatically links in the Turbo C++ graphics library.
- If you're using the command-line compiler TCC.EXE, you have to list GRAPHICS.LIB on the command line. For example, if your program MYPROG.C uses graphics, the TCC command line would be

```
TCC MYPROG GRAPHICS.LIB
```

Important! Because graphics functions use **far** pointers, graphics are not supported in the tiny memory model.

There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, and so on, which are memory-model specific). Each function in GRAPHICS.LIB is a **far** function, and those graphics functions that take pointers take **far** pointers. For these functions to work correctly, it is important that you **#include** graphics.h in every module that uses graphics.

The graphics library functions

Turbo C++'s graphics functions fall into seven categories:

- graphics system control
- drawing and filling
- manipulating screens and viewports
- text output
- color control
- error handling
- state query

Graphics system control

Here's a quick summary of the graphics system control:

closegraph	Shuts down the graphics system.
detectgraph	Checks the hardware and determines which graphics driver to use; recommends a mode.
graphdefaults	Resets all graphics system variables to their default settings.
_graphfreemem	Deallocates graphics memory; hook for defining your own routine.
_graphgetmem	Allocates graphics memory; hook for defining your own routine.
getgraphmode	Returns the current graphics mode.
getmoderange	Returns lowest and highest valid modes for specified driver.
initgraph	Initializes the graphics system and puts the hardware into graphics mode.
installuserdriver	Installs a vendor-added device driver to the BGI device driver table.
installuserfont	Loads a vendor-added stroked font file to the BGI character file table.
registerbgidriver	Registers are linked-in or user-loaded driver file for inclusion at link time.
restorecrtmode	Restores the original (pre- initgraph) screen mode.
setgraphbufsize	Specifies size of the internal graphics buffer.
setgraphmode	Selects the specified graphics mode, clears the screen, and restores all defaults.

Turbo C++'s graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color/Graphics Adapter (CGA)
- Multi-Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode.

You can tell **initgraph** to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to autodetect, it calls **detectgraph** to select a graphics driver and mode. If you tell **initgraph** to use a particular graphics driver and mode, you must be sure that the hardware is present. If you force **initgraph** to use hardware that is not present, the results will be unpredictable.

Once a graphics driver has been loaded, you can find out the name of the driver by using the **getdrivername** function and how many modes a driver supports with **getmaxmode**. **getgraphmode** will tell you which graphics mode you are currently in. Once you have a mode number, you can find out the name of the mode with **getmodename**. You can change graphics modes with **setgraphmode** and return the video mode to its original state (before graphics was initialized) with **restorecrtmode**. **restorecrtmode** returns the screen to text mode, but it does not close the graphics system (the fonts and drivers are still in memory).

graphdefaults resets the graphics state's settings (viewport size, draw color, fill color and pattern, and so on) to their default values.

installuserdriver and **installuserfont** let you add new device drivers and fonts to your BGI.

Finally, when you're through using graphics, call **closegraph** to shut down the graphics system. **closegraph** unloads the driver from memory and restores the original video mode (via **restorecrtmode**).

A more detailed discussion

The previous discussion provided an overview of how **initgraph** operates. In the following paragraphs, we describe the behavior of **initgraph**, **_graphgetmem**, and **_graphfreemem** in some detail.

Normally, the **initgraph** routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGIOBJ utility—see UTIL.DOC, included with your distribution disks), then placing calls to **registerbgidriver** in your source code (before the call to **initgraph**) to register the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (via **detectgraph**), **initgraph** checks to see if the desired driver has been registered. If so, **initgraph** uses the registered driver directly from memory. Otherwise, **initgraph** allocates memory for the driver and loads the .BGI file from disk.

Note Using **registerbgidriver** is an advanced programming technique, not recommended for novice programmers.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls **_graphgetmem** to allocate memory, and calls **_graphfreemem** to free it. By default, these routines simply call **malloc** and **free**, respectively.

*If you provide your own **_graphgetmem** or **_graphfreemem**, you may get a "duplicate symbols" warning message. Just ignore the warning.*

You can override this default behavior by defining your own **_graphgetmem** and **_graphfreemem** functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: They will override the default functions with the same names that are in the standard C libraries.

Drawing and filling

Here's a quick summary of the drawing and filling functions:

<i>Drawing:</i>	arc	Draws a circular arc.
	circle	Draws a circle.
	drawpoly	Draws the outline of a polygon.
	ellipse	Draws an elliptical arc.
	getarccoords	Returns the coordinates of the last call to arc or ellipse .

getaspectratio	Returns the aspect ratio of the current graphics mode.
getlinesettings	Returns the current line style, line pattern, and line thickness.
line	Draws a line from $(x0,y0)$ to $(x1,y1)$.
linerel	Draws a line to a point some relative distance from the current position (CP).
lineto	Draws a line from the current position (CP) to (x,y) .
moveto	Moves the current position (CP) to (x,y) .
moverel	Moves the current position (CP) a relative distance.
rectangle	Draws a rectangle.
setaspectratio	Changes the default aspect ratio-correction factor.
setlinestyle	Sets the current line width and style.

Filling:

bar	Draws and fills a bar.
bar3d	Draws and fills a 3-D bar.
fillellipse	Draws and fills an ellipse.
fillpoly	Draws and fills a polygon.
floodfill	Flood-fills a bounded region.
getfillpattern	Returns the user-defined fill pattern.
getfillsettings	Returns information about the current fill pattern and color.
pieslice	Draws and fills a pie slice.
sector	Draws and fills an elliptical pie slice.
setfillpattern	Selects a user-defined fill pattern.
setfillstyle	Sets the fill pattern and fill color.

With Turbo C++'s drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pie slices, two- and three-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape) with one of eleven pre-defined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the current position (CP).

You draw lines and unfilled shapes with the functions **arc**, **circle**, **drawpoly**, **ellipse**, **line**, **linerel**, **lineto**, and **rectangle**. You can fill these shapes with **floodfill**, or combine drawing/filling into one step with **bar**, **bar3d**, **fillellipse**, **fillpoly**, **pieslice**, and **sector**. You use **setlinestyle** to specify whether the drawing line (and border

line for filled shapes) is thick or thin, and whether its style is solid, dotted, and so forth, or some other line pattern you've defined. You can select a predefined fill pattern with **setfillstyle**, and define your own fill pattern with **setfillpattern**. You move the CP to a specified location with **moveto**, and move it a specified displacement with **moverel**.

To find out the current line style and thickness, you call **getline-settings**. For information about the current fill pattern and fill color, you call **getfillsettings**; you can get the user-defined fill pattern with **getfillpattern**.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with **getaspect-ratio**, and get coordinates of the last drawn arc or ellipse by calling **getarccoords**. If your circles are not perfectly round, use **setaspectratio** to correct them.

Manipulating the screen and viewport

Here's a quick summary of the screen-, viewport-, image-, and pixel-manipulation functions:

Screen manipulation:

cleardevice Clears the screen (active page).
setactivepage Sets the active page for graphics output.
setvisualpage Sets the visual graphics page number.

Viewport manipulation:

clearviewport Clears the current viewport.
getviewsettings Returns information about the current viewport.
setviewport Sets the current output viewport for graphics output.

Image manipulation:

getimage Saves a bit image of the specified region to memory.
imagesize Returns the number of bytes required to store a rectangular region of the screen.
putimage Puts a previously saved bit image onto the screen.

Pixel manipulation:

getpixel Gets the pixel color at (x,y) .
putpixel Plots a pixel at (x,y) .

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one fell swoop with a call to **cleardevice**; this routine erases the entire screen and homes the CP in the viewport, but leaves all other graphics system set-

tings intact (the line, fill, and text styles; the palette; the viewport settings; and so on).

Depending on your graphics adapter, your system has between one and four screen-page buffers, which are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify the active screen page (where graphics functions place their output) with **setactivepage** and the visual page (the one displayed onscreen) with **setvisualpage**.

Once your screen is in a graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to **setviewport**. You define the viewport's position in terms of absolute screen coordinates and specify whether clipping is on (active) or off. You clear the viewport with **clearviewport**. To find out the current viewport's absolute screen coordinates and clipping status, call **getviewsettings**.

You can capture a portion of the onscreen image with **getimage**, call **imagesize** to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with **putimage**.

The coordinates for all output functions (drawing, filling, text, and so on) are viewport-relative.

You can also manipulate the color of individual pixels with the functions **getpixel** (which returns the color of a given pixel) and **putpixel** (which plots a specified pixel in a given color).

Text output in graphics mode

Here's a quick summary of the graphics-mode text output functions:

gettextsettings	Returns the current text font, direction, size, and justification.
outtext	Sends a string to the screen at the current position (CP).
outtextxy	Sends a string to the screen at the specified position.
registerbgifont	Registers a linked-in or user-loaded font.
settextjustify	Sets text justification values used by outtext and outtextxy .
settextstyle	Sets the current text font, style, and character magnification factor.
setusercharsize	Sets width and height ratios for stroked fonts.

textheight Returns the height of a string in pixels.
textwidth Returns the width of a string in pixels.

The graphics library includes an 8×8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either **outtext** or **outtextxy**, and control the justification of the output text (with respect to the CP) with **settextjustify**. You choose the character font, direction (horizontal or vertical), and size (scale) with **settextstyle**. You can find out the current text settings by calling **gettextsettings**, which returns the current text font, justification, magnification, and direction in a **textsettings** structure. **setusercharsize** allows you to modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by **outtext** and **outtextxy** will be clipped at the viewport borders. If clipping is *off*, these functions will throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

To determine the onscreen size of a given text string, call **textheight** (which measures the string's height in pixels) and **textwidth** (which measures its width in pixels).

The default 8×8 bit-mapped font is built into the graphics package, so it is always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGIOBJ utility) and linked into your .EXE file.

Normally, the **settextstyle** routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from

disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR file to an .OBJ file (using the BGIOBJ utility—read about it in UTIL.DOC, the online documentation included with your distribution disks), then placing calls to **registerbgifont** in your source code (before the call to **setttextstyle**) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

Note Using **registerbgifont** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in UTIL.DOC, included with your distribution disks.

Color control Here's a quick summary of the color control functions:

<i>Get color information:</i>	getbkcolor	Returns the current background color.
	getcolor	Returns the current drawing color.
	getdefaultpalette	Returns the palette definition structure.
	getmaxcolor	Returns the maximum color value available in the current graphics mode.
	getpalette	Returns the current palette and its size.
<i>Set one or more colors:</i>	getpalettesize	Returns the size of the palette look-up table.
	setallpalette	Changes all palette colors as specified.
	setbkcolor	Sets the current background color.
	setcolor	Sets the current drawing color.
	setpalette	Changes one palette color as specified by its arguments.

Before summarizing how these color control functions work, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and palettes The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot onscreen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette*. The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The number of colors that can be displayed at any one time is equal to the

number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* is 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* - 1). **getmaxcolor** returns the highest valid pixel value (*size* - 1) for the current graphics driver and mode.

When we discuss the Turbo C++ graphics functions, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is a pixel's value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, and so on) have not changed.

Background and drawing color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are simply set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You choose a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between CGA and EGA, so we'll present them separately. Color control on the AT&T driver, and the lower resolutions of the MCGA driver is similar to CGA.

On the CGA, you can choose to display your graphics in low resolution (320×200), which allows you to use four colors, or high resolution (640×200), in which you can use two colors.

CGA low resolution

In the low resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can only set the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color. This background color can be any one of the 16 available colors (see table of CGA background colors below).

You choose which palette you want by the mode you select (CGAC0, CGAC1, CGAC2, CGAC3); these modes use color palette 0 through color palette 3, as detailed in the following table.

The CGA drawing colors and the equivalent constants are defined in `graphics.h`.

Palette number	Constant assigned to color number (pixel value)		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

To assign one of these colors as the CGA drawing color, call **setcolor** with either the color number or the corresponding constant name as an argument; for example, if you are using palette 3 and you want to use cyan as the drawing color:

```
setcolor(1);
```

or

```
setcolor(CG_A_CYAN);
```

The available CGA background colors, defined in `graphics.h`, are listed in the following table:

Numeric value	Symbolic name	Numeric value	Symbolic name
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

The CGA's foreground colors are the same as those listed in this table.

To assign one of these colors to the CGA background color, use **setbkcolor**(*color*), where *color* is one of the entries in the preceding table. For CGA, this color is not a pixel value (palette index); it directly specifies the *actual* color to be put in the first palette entry.

CGA high resolution

In high resolution mode (640×200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its background color; you set it with the **setbkcolor** routine.

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

CGA palette routines

Because the CGA palette is predetermined, you should not use the **setallpalette** routine on a CGA. Also, you should not use **setpalette**(*index, actual_color*), except for *index* = 0. (This is an alternate way to set the CGA background color to *actual_color*.)

Color control on the EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors, and each entry is user-settable. You can retrieve the current palette with **getpalette**, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with **setpalette**, or all at once with **setallpalette**.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in `graphics.h` that contain the corresponding hardware color values: these are `EGA_BLACK`, `EGA_WHITE`, and so on. You can also get these values with **getpalette**.

The **setbkcolor**(*color*) routine behaves differently on an EGA than on a CGA. On an EGA, **setbkcolor** copies the actual color value that's stored in entry #*color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error handling in graphics mode

Here's a quick summary of the graphics-mode error-handling functions:

grapherrormsg Returns an error message string for the specified error code.

graphresult Returns an error code for the last graphics operation that encountered a problem.

If an error occurs when a graphics library function is called (such as a font requested with **settextstyle** not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling **graphresult**. A call to **grapherrormsg(graphresult())** returns the error strings listed in the following table.

The error return code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when **initgraph** executes successfully, or when you call **graphresult**. Therefore, if you want to know which graphics function returned which error, you should store the value of **graphresult** into a temporary variable and then test it.

Error code	<i>graphics_errors</i> constant	Corresponding error message string
0	grOk	No error
-1	grNoInitGraph	(BGI) graphics not installed (use initgraph)
-2	grNotDetected	Graphics hardware not detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	Not enough memory to load driver
-6	grNoScanMem	Out of memory in scan fill
-7	grNoFloodMem	Out of memory in flood fill
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

Include the full path to your BGI directory (using couple backslashes) in the third parameter to the **initgraph()** function, as shown in the following example:

```
intergraph (&gdriver, &gmode, "c:\\tc\\bgi");
```

State query. The following table summarizes the graphics mode state query functions:

Table 20.1
Graphics mode state query
functions

Function	Returns
getarccoords	Information about the coordinates of the last call to arc or ellipse .
getaspectratio	Aspect ratio of the graphics screen.
getbkcolor	Current background color.
getcolor	Current drawing color.
getdrivername	Name of current graphics driver.
getfillpattern	User-defined fill pattern.
getfillsettings	Information about the current fill pattern and color.
getgraphmode	Current graphics mode.
getlinesettings	Current line style, line pattern, and line thickness.
getmaxcolor	Current highest valid pixel value.
getmaxmode	Maximum mode number for current driver.
getmaxx	Current <i>x</i> resolution.
getmaxy	Current <i>y</i> resolution.
getmodename	Name of a given driver mode.
getmoderange	Mode range for a given driver.
getpalette	Current palette and its size.
getpixel	Color of the pixel at <i>x,y</i> .
gettextsettings	Current text font, direction, size, and justification.
getviewsettings	Information about the current viewport.
getx	<i>x</i> coordinate of the current position (CP).
gety	<i>y</i> coordinate of the current position (CP).

In each of Turbo C++'s graphics functions categories there is at least one state query function. These functions are mentioned under their respective categories and also covered here. Each of the Turbo C++ graphics state query functions is named **getsomething** (except in the error-handling category). Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined in `graphics.h`, fill that structure with the appropriate information, and return no value.

The state query functions for the graphics system control category are **getgraphmode**, **getmaxmode**, and **getmoderange**: The first returns an integer representing the current graphics driver and mode, the second returns the maximum mode number for a given driver, and the third returns the range of modes supported by a given graphics driver. **getmaxx** and **getmaxy** return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state query functions are **getarccoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, and **getlinesettings**. **getarccoords** fills a structure with coordinates from the last call to **arc** or **ellipse**; **getaspectratio** tells the current mode's aspect ratio, which the graphics system uses to make circles come out round.

getfillpattern returns the current user-defined fill pattern. **getfillsettings** fills a structure with the current fill pattern and fill color. **getlinesettings** fills a structure with the current line style (solid, dashed, and so on), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state query functions are **getviewsettings**, **getx**, **gety**, and **getpixel**. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling **getviewsettings**, which fills a structure with the information. **getx** and **gety** return the (viewport-relative) x- and y-coordinates of the CP. **getpixel** returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state query function: **gettextsettings**. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Turbo C++'s color-control function category includes three state query functions. **getbkcolor** returns the current background color, and **getcolor** returns the current drawing color. **getpalette** fills a structure with the size of the current drawing palette and the palette's contents. **getmaxcolor** returns the highest valid pixel value for the current graphics driver and mode (palette *size* - 1).

Finally, **getmodename** and **getdrivername** return the name of a given driver mode and the name of the current graphics driver, respectively.

Editor reference

The editor has two command sets: CUA and Alternate. The tables in this appendix list all the available commands. You can use some commands in both modes, while others are available in only one mode. Choose Options | Environment | Preferences and select the command set you want in the Preferences dialog box.

Most of these commands need no explanation. Those that do are described in the text following Table A.1.

Table A.1
Editing commands

A word is defined as a sequence of characters separated by one of the following: space <> . ; . () ^ ` * + - / \$ # = | ~ ? ! " % & ` ; @ \ , and all control and graphic characters.

Command	Both modes	CUA	Alternate
Cursor movement commands			
Character left	←		Ctrl+S
Character right	→		Ctrl+D
Word left	Ctrl+ ←		Ctrl+A
Word right	Ctrl+ →		Ctrl+F
Line up	↑		Ctrl+E
Line down	↓		Ctrl+X
Scroll up one line	Ctrl+W		
Scroll down one line	Ctrl+Z		
Page up	PgUp		Ctrl+R
Page down	PgDn		Ctrl+C
Beginning of line	Home		
	Ctrl+Q S		
End of line	End		
	Ctrl+Q D		
Top of window	Ctrl+Q E	Ctrl+E	Ctrl+Home
Bottom of window	Ctrl+Q X	Ctrl+X	Ctrl+End
Top of file	Ctrl+Q R	Ctrl+Home	Ctrl+PgUp
Bottom of file	Ctrl+Q C	Ctrl+End	Ctrl+PgDn
Move to previous position	Ctrl+P		

Table A.1: Editing commands (continued)

Command	Both modes	CUA	Alternate
Insert and delete commands			
Delete character	<i>Del</i>		<i>Ctrl+G</i>
Delete character to left	<i>Backspace</i>		<i>Ctrl+H</i>
	<i>Shift+Tab</i>		
Delete line	<i>Ctrl+Y</i>		
Delete to end of line	<i>Ctrl+Q Y</i>	<i>Shift+Ctrl+Y</i>	
Delete word	<i>Ctrl+T</i>		
Insert line	<i>Ctrl+N</i>		
Insert mode on/off	<i>Ins</i>		<i>Ctrl+V</i>
Block commands			
Move to beginning of block	<i>Ctrl+Q B</i>		
Move to end of block	<i>Ctrl+Q K</i>		
Set beginning of block	<i>Ctrl+K B</i>		
Set end of block	<i>Ctrl+K K</i>		
Exit to menu bar	<i>Ctrl+K D</i>		
Hide/Show block	<i>Ctrl+K H</i>		
Mark line	<i>Ctrl+K L</i>		
Print selected block	<i>Ctrl+K P</i>		
Mark word	<i>Ctrl+K T</i>		
Delete block	<i>Ctrl+K Y</i>		
Copy block	<i>Ctrl+K C</i>		
Move block	<i>Ctrl+K V</i>		
Copy to Clipboard	<i>Ctrl+Ins</i>		
Cut to Clipboard	<i>Shift+Del</i>		
Delete block	<i>Ctrl+Del</i>		
Indent block	<i>Ctrl+K I</i>	<i>Shift+Ctrl+I</i>	
Paste from Clipboard	<i>Shift+Ins</i>		
Read block from disk	<i>Ctrl+K R</i>	<i>Shift+Ctrl+R</i>	
Unindent block	<i>Ctrl+K U</i>	<i>Shift+Ctrl+U</i>	
Write block to disk	<i>Ctrl+K W</i>	<i>Shift+Ctrl+W</i>	
Extending selected blocks			
Left one character	<i>Shift+ ←</i>		
Right one character	<i>Shift+ →</i>		
End of line	<i>Shift+End</i>		
Beginning of line	<i>Shift+Home</i>		
Same column on next line	<i>Shift+ ↓</i>		
Same column on previous line	<i>Shift+ ↑</i>		
One page down	<i>Shift+PgDn</i>		
One page up	<i>Shift+PgUp</i>		
Left one word	<i>Shift+Ctrl+ ←</i>		
Right one word	<i>Shift+Ctrl+ →</i>		
End of file	<i>Shift+Ctrl+End</i>		<i>Shift+Ctrl+PgDn</i>
Beginning of file	<i>Shift+Ctrl+Home</i>		<i>Shift+Ctrl+PgUp</i>

Table A.1: Editing commands (continued)

Command	Both modes	CUA	Alternate
Other editing commands			
Autoindent mode on/off	<i>Ctrl+O I</i>		
Cursor through tabs on/off	<i>Ctrl+O R</i>		
Exit the IDE		<i>Alt+F4</i>	<i>Alt+X</i>
Find place marker	<i>Ctrl+Q n *</i>	<i>Ctrl n *</i>	
Help	<i>F1</i>		
Help index	<i>Shift+F1</i>		
Insert control character	<i>Ctrl+P**</i>		
Maximize window			<i>F5</i>
Open file			<i>F3</i>
Optimal fill mode on/off	<i>Ctrl+O F</i>		
Pair matching	<i>Ctrl+Q [,</i> <i>Ctrl+Q]</i>	<i>Alt+[,Alt+]</i>	
Save file	<i>Ctrl+K S</i>		<i>F2</i>
Search	<i>Ctrl+Q F</i>		
Search again		<i>F3</i>	<i>Ctrl+L</i>
Search and replace	<i>Ctrl+Q A</i>		
Set marker	<i>Ctrl+K n *</i>	<i>Shift+Ctrl n *</i>	
Tabs mode on/off	<i>Ctrl+O T</i>		
Topic search help	<i>Ctrl+F1</i>		
Undo	<i>Alt+Bksp</i>		
Redo	<i>Shift+Alt+Bksp</i>		
Unindent mode on/off	<i>Ctrl+O U</i>		

* *n* represents a number from 0 to 9.

** Enter control characters by first pressing *Ctrl+P*, then pressing the desired control character.

Block commands

A block of text is any amount of text, from a single character to hundreds of lines, that is selected on your screen. There can be only one block in a window at a time. Select a block with your mouse or by holding down *Shift* while moving your cursor to the end of the block with the arrow keys. Once selected, the block can be copied, moved, deleted, or written to a file. You can use the Edit menu commands to perform these operations or you can use the keyboard commands listed in the following table.

When you choose Edit | Copy or press *Ctrl+Ins*, the selected block is copied to the Clipboard. When you choose Edit | Paste or *Shift+Ins*, the block held in the Clipboard is pasted at the current cursor position. The selected text remains unchanged and is no longer selected.

If you choose Edit | Cut or press *Shift+Del*, the selected block is moved from its original position and held in the Clipboard. It is pasted at the current cursor position when you choose the Paste command.

The copying, cutting, and pasting commands are the same in both the CUA and Alternate command sets.

Table A.2: Block commands in depth

Command	CUA	Alternate	Function
Copy block	<i>Ctrl+Ins</i> , <i>Shift+Ins</i>	<i>Ctrl+Ins</i> , <i>Shift+Ins</i>	Copies a previously selected block to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The original block is unchanged. If no block is selected, nothing happens.
Copy text	<i>Ctrl+Ins</i>	<i>Ctrl+Ins</i>	Copies selected text to the Clipboard.
Cut text	<i>Shift+Del</i>	<i>Shift+Del</i>	Cuts selected text to the Clipboard.
Delete block	<i>Ctrl+Del</i>	<i>Ctrl+Del</i>	Deletes a selected block. You can "undelete" a block with Undo.
Move block	<i>Shift+Del</i> , <i>Shift+Ins</i>	<i>Shift+Del</i> , <i>Shift+Ins</i>	Moves a previously selected block from its original position to the Clipboard and, after you move your cursor to where you want the text to appear, pastes it to the new cursor position. The block disappears from its original position. If no block is marked, nothing happens.
Paste from Clipboard	<i>Shift+Ins</i>	<i>Shift+Ins</i>	Pastes the contents of the Clipboard.
Read block from disk	<i>Shift+Ctrl+R</i> <i>Ctrl+K R</i>	<i>Ctrl+K R</i>	Reads a disk file into the current text at the cursor position exactly as if it were a block. The text read is then selected as a block. When this command is issued, you are prompted for the name of the file to read. You can use wildcards to select a file to read; a directory is displayed. The file specified can be any legal file name.
Write block to disk	<i>Shift+Ctrl+W</i> <i>Ctrl+K W</i>	<i>Ctrl+K W</i>	Writes a selected block to a file. When you give this command, you are prompted for the name of the file to write to. The file can be given any legal name (the default extension is CPP). If you prefer to use a file name without an extension, append a period to the end of its name.

If you have used Borland editors in the past, you may prefer to use the block commands listed in this table; they work in both command sets.

Table A.3
Borland-style block
commands

Selected text is highlighted only if both the beginning and end have been set and the beginning comes before the end.

Command	Keys	Function
Set beginning of block	<i>Ctrl+K B</i>	Begin selection of text.
Set end of block	<i>Ctrl+K K</i>	End selection of text.
Hides/shows selected text	<i>Ctrl+K H</i>	Alternately displays and hides selected text.
Copy selected text to the cursor.	<i>Ctrl+K C</i>	Copies the selected text to the position of the cursor. Useful only with the Persistent Block option.
Move selected text to the cursor.	<i>Ctrl+K V</i>	Moves the selected text to the position of the cursor. Useful only with the Persistent Block option.

Other editing commands

The next table describes certain editing commands in more detail. The table is arranged alphabetically by command name.

Table A.4: Other editor commands in depth

Command	CUA	Alternate	Function
Autoindent	<i>Ctrl+O I</i>	<i>Ctrl+O I</i>	Toggles the automatic indenting of successive lines. You can also use Options Environment Editor Autoindent in the IDE to turn automatic indenting on and off.
Cursor through tabs	<i>Ctrl+O R</i>	<i>Ctrl+O R</i>	The arrow keys will move the cursor to the middle of tabs when this option is on; otherwise the cursor jumps several columns when cursoring over multiple tabs. <i>Ctrl+O R</i> is a toggle.
Find place marker	<i>Ctrl+n*</i> <i>Ctrl+Q n*</i>	<i>Ctrl+Q n*</i>	Finds up to ten place markers (<i>n</i> can be any number in the range 0 to 9) in text. Move the cursor to any previously set marker by pressing <i>Ctrl+Q</i> and the marker number.
Open file		<i>F3</i>	Lets you load an existing file into an edit window.
Optimal fill	<i>Ctrl+O F</i>	<i>Ctrl+O F</i>	Toggles optimal fill. Optimal fill begins every line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters.
Save file		<i>F2</i>	Saves the file and returns to the editor.

Table A.4: Other editor commands in depth (continued)

Command	CUA	Alternate	Function
Set place	<i>Shift+Ctrl n*</i> <i>Ctrl+K n*</i>	<i>Ctrl+K n*</i>	Mark up to ten places in text. After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the Find Place Marker command (being sure to use the same marker number). You can have ten places marked in each window.
Show previous error	<i>Alt+F7</i>	<i>Alt+F7</i>	Moves the cursor to the location of the previous error or warning message. This command is available only if there are messages in the Message window that have associated line numbers.
Show next error	<i>Alt+F8</i>	<i>Alt+F8</i>	Moves the cursor to the location of the next error or warning message. This command is available only if there are messages in the Message window that have associated line numbers.
Tab mode	<i>Ctrl+O T</i>	<i>Ctrl+O T</i>	Toggles Tab mode. You can specify the use of true tab characters in the IDE with the Options Environment Editor Use Tab Character option.
Unindent	<i>Ctrl+O U</i>	<i>Ctrl+O U</i>	Toggles Unindent. You can turn Unindent on and off from the IDE with the Options Environment Editor Backspace Unindents option.

* *n* represents a number from 0 to 9.

Precompiled headers

Turbo C++ can generate and subsequently use precompiled headers for your projects. Precompiled headers can greatly speed up compilation times.

How they work

When compiling large C and C++ programs, the compiler can spend up to half of its time parsing header files. When the compiler parses a header file, it enters declarations and definitions into its symbol table. If 10 of your source files include the same header file, this header file is parsed 10 times, producing the same symbol table every time.

Precompiled header files cut this process short. During one compilation, the compiler stores an image of the symbol table on disk in a file called TCDEF.SYM by default. (TCDEF.SYM is stored in the same directory as the compiler.) Later, when the same source file is compiled again (or another source file that includes the same header files), the compiler reloads TCDEF.SYM from disk instead of parsing all the header files again. Directly loading the symbol table from disk is over 10 times faster than parsing the text of the header files.

Precompiled headers will only be used if the second compilation uses one or more of the same header files as the first one, and if a

lot of other things, like compiler options, defined macros and so on, are also identical.

If, while compiling a source file, Turbo C++ discovers that the first **#includes** are identical to those of a previous compilation (of the same source or a different source), it will load the binary image for those **#includes**, and parse the remaining **#includes**.

Use of precompiled headers for a given module is an all or nothing deal: the precompiled header file is not updated for that module if compilation of any included header file fails.

Drawbacks

When using precompiled headers, TCDEF.SYM can become very big, because it contains symbol table images for all sets of includes encountered in your sources. You can reduce the size of this file; see "Optimizing precompiled headers" on page 637.

If a header contains any code, then it can't be precompiled. For example, while C++ class definitions may appear in header files, you should take care that only member functions that are inline are defined in the header; heed warnings such as "Functions containing for are not expanded inline".

Using precompiled headers

You can control the use of precompiled headers in any of the following ways:

- from within the IDE, using the Options | Compiler | Code Generation Options dialog box. The IDE bases the name of the precompiled header file on the project name, creating *PROJECT.SYM*
- from the command line using the **-H**, **-H=filename**, and **-Hu** options (see page 286)
- or from within your code using the pragmas **hdrfile** and **hdrstop** (see Chapter 14)

Setting file names

The compiler uses just one file to store all precompiled headers. The default file name is TCDEF.SYM. You can explicitly set the name with the **-H=filename** command-line option or the **#pragma hdrfile** directive.

Caution! You may notice that your .SYM file is smaller than it should be. If this happens, the compiler may have run out of disk space when writing to the .SYM file. When this happens, the compiler deletes the .SYM in order to make room for the .OBJ file, then starts creating a new (and therefore shorter) .SYM file. If this happens, just free up some disk space before compiling.

Establishing identity

The following conditions need to be identical for a previously generated precompiled header to be loaded for a subsequent compilation.

The second or later source file must:

- have the same set of include files in the same order
- have the same macros defined to identical values
- use the same language (C or C++)
- use header files with identical time stamps; these header files can be included either directly or indirectly

In addition, the subsequent source file must be compiled with the same settings for the following options:

- memory model, including SS != DS (**-mx**)
- underscores on externs (**-u**)
- maximum identifier length (**-iL**)
- Pascal calls (**-p**)
- treat enums as integers (**-b**)
- default char is unsigned (**-K**)
- virtual table control (**-Vx**)

Optimizing precompiled headers

For Turbo C++ to most efficiently compile using precompiled headers, follow these rules:

- Arrange your header files in the same sequence in all source files.
- Put the largest header files first.
- Prime TCDEF.SYM with often-used initial sequences of header files.
- Use `#pragma hdrstop` to terminate the list of header files at well-chosen places. This lets you make the list of header files in different sources look similar to the compiler. `#pragma hdrstop` is described in more detail in Chapter 14.

For example, given the two source files ASOURCE.C and BSOURCE.C, both of which include myhdr.h,

```
ASOURCE.C:  #include "myhdr.h"
             #include "xxx.h"
             <...>
```

```
BSOURCE.C:  #include "zz.h"
             #include <string.h>
             #include "myhdr.h"
             <...>
```

You would rearrange the beginning of BSOURCE.C to:

```
Revised BSOURCE.C:  #include "myhdr.h"
                    #include "zz.h"
                    #include <string.h>
                    <...>
```

Note that myhdr.h are in the same order in BSOURCE.C as they are in ASOURCE.C. You could also make a new source called PREFIX.C containing only the header files, like this:

```
PREFIX.C      #include "myhdr.h"
```

If you compile PREFIX.C first (or insert a `#pragma hdrstop` in both ASOURCE.C and BSOURCE.C after the `#include "myhdr.h"` statement) the net effect is that after the initial compilation of PREFIX.C, both ASOURCE.C and BSOURCE.C will be able to load the symbol table produced by PREFIX.C. The compiler will then only need to parse xxx.h for ASOURCE.C and zz.h and string.h for BSOURCE.C.

Error messages

Turbo C++ error messages include: compile-time, DPMI server, MAKE, run-time, TLIB, and TLINK. We explain them here; user-interface error messages are explained in online Help.

The *type* of message (such as *Compile-time* or *DPMI*) is noted in the column to the left. Most explanations provide a probable cause and remedy for the error or warning message.

Finding a message in this appendix

The messages are listed in ASCII alphabetic order; messages beginning with symbols normally come first, followed by numbers and letters of the alphabet.

Since messages that begin with a variable cannot be alphabetized by what you will actually see when you receive such a message, all such messages are alphabetized by the word following the variable.

For example, if you have a C++ function **goforit**, you might receive the following actual message:

```
goforit must be declared with no arguments
```

In order to look this error message up, you would need to find

function must be declared with no arguments

alphabetized starting with the word “must”.

If the variable occurs later in the text of the error message (for example, “Address of overloaded function *function* doesn't match

Type"), you can find the message in correct alphabetical order; in this case, under the letter A.

Types of messages

The kinds of messages you get are different, depending on where they come from. This section lists each category with a table of variables that it may contain.

Compile-time messages

The Turbo C++ compiler diagnostic messages fall into three classes: fatal errors, errors, and warnings.

Fatal errors are rare. Some of them indicate an internal compiler error. When a fatal error occurs, compilation stops immediately. You must take appropriate action and then restart compilation.

Errors indicate program syntax errors, command-line errors, and disk or memory access errors. The compiler completes the current phase of the compilation and then stop. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing and code-generating).

Warnings do not prevent the compilation from finishing. They indicate conditions that are suspicious, but are usually legitimate as part of the language. The compiler also produces warnings if you use some machine-dependent constructs in your source files.

The compiler prints messages with the message class first, then the source file name and line number where the compiler detected the condition, and finally the text of the message itself.

Line numbers are not exact

You should be aware of one detail about line numbers in error messages: the compiler only generates messages as they are detected. Because C and C++ do not force any restrictions on placing statements on a line of text, the true cause of the error may be one or more lines before or after the line number mentioned.

The following variable names and values are some of those that appear in the compiler messages listed in this appendix (most are self-explanatory). When you get an error message, the appropriate name or value is substituted.

Table C.1
Compile-time message
variables

What you'll see in the manual	What you'll see on your screen
<i>argument</i>	An argument (command-line or other)
<i>class</i>	A class name
<i>filename</i>	A file name (with or without extension)
<i>function</i>	A function name
<i>group</i>	A group name
<i>identifier</i>	An identifier (variable name or other)
<i>language</i>	The name of a programming language
<i>member</i>	The name of a data member or member function
<i>message</i>	A message string
<i>module</i>	A module name
<i>number</i>	An actual number
<i>option</i>	An option (command-line or other)
<i>parameter</i>	A parameter name
<i>segment</i>	A segment name
<i>specifier</i>	A type specifier
<i>symbol</i>	A symbol name
<i>type</i>	A type name
XXXXh	A 4-digit hexadecimal number, followed by <i>h</i>

DPMI server messages

All Dos Protected Mode Interface (DPMI) server error messages but one relate to conditions that are out of the scope of Borland's software control. If the program can't find your machine-type, the message directs you to a solution. Otherwise there is a problem with the configuration of your system, your disks, or the hardware itself. Normally you will receive one of the common messages but if your disk has been damaged and you receive an undocumented message, call Technical Support.

The error messages are all fatal. They contain no variables and there are no warnings.

MAKE messages

MAKE diagnostic messages fall into two classes: errors and fatal errors.

- Errors indicate some sort of syntax or semantic error in the source makefile.
- Fatal errors prevent processing of the makefile. You must take appropriate action and then restart MAKE.

The following generic names and values appear in the messages listed in this section. When you get an error message, the appropriate name or value is substituted.

Table C.2
MAKE error message
variables

What you'll see in the manual	What you'll see on your screen
<i>argument(s)</i>	An argument (command-line or other)
<i>expression</i>	An expression
<i>filename</i>	A file name (with or without extension)
<i>line number</i>	A line number
<i>message</i>	A message string
<i>target</i>	A receiver

Run-time error messages

Turbo C++ has a small number of run-time error messages. These errors occur after the program has successfully compiled and while it is running.

TLIB messages

TLIB has error and warning messages. The following generic names and values appear in TLIB messages. When you get a message, the variable is substituted.

Table C.3
TLIB message variables

What you'll see in the manual	What you'll see on your screen
<i>filename</i>	A file name (with or without extension)
<i>function</i>	A function name
<i>len</i>	An actual number
<i>module</i>	A module name
<i>num</i>	An actual number
<i>path</i>	A path name
<i>reason</i>	Reason given in warning message
<i>size</i>	An actual number
<i>type</i>	A type name

TLINK messages

The linker has three types of messages: fatal errors, errors, and warnings.

- A fatal error causes TLINK to stop immediately; the .EXE file is deleted.
- An error (also called a nonfatal error) does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file. Errors are treated as fatal errors in the IDE.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

Table C.4
TLINK error message variables

What you'll see in the manual	What you'll see on your screen
<i>errorcode</i>	Error code number for internal errors
<i>filename</i>	A file name (with or without extension)
<i>group</i>	A group name
<i>linenum</i>	The line number within a file
<i>module</i>	A module name
<i>segment</i>	A segment name
<i>symbol</i>	A symbol name
<i>XXXXh</i>	A 4-digit hexadecimal number, followed by <i>h</i>

Message explanations

- MAKE fatal error* **)' missing in macro invocation in command *command***
A left parenthesis is required to invoke a macro.
- Compile-time error* **(expected**
A left parenthesis was expected before a parameter list.
- Compile-time error* **) expected**
A right parenthesis was expected at the end of a parameter list.
- Compile-time error* **, expected**
A comma was expected in a list of declarations, initializations, or parameters.
- Compile-time error* **: expected after private/protected/public**
When used to begin a **private/protected/public** section of a C++ class, these reserved words must be followed by a colon.
- Compile-time error* **< expected**
The keyword **template** was not followed by a left angle bracker (**<**). Every template declaration must include the template formal parameters enclosed within angle brackets (**< >**), immediately following the **template** keyword.
- TLIB error* **@ seen, expected a response-files name**
The response file is not given immediately after **@**.
- Compile-time error* **{ expected**
A left brace (**{**) was expected at the start of a block or initialization.

- Compile-time error* **} expected**
 A right brace (`}`) was expected at the end of a block or initialization.
- TLINK fatal error* **32-bit record encountered**
 An object file that contains 80386 32-bit records was encountered, and the `/3` option had not been used.
- Compile-time error* **286/287 instructions not enabled**
 Use the `-2` command-line compiler option or the 80286 options from the Options | Compiler | Code Generation | Advanced Code Generation dialog box to enable 286/287 opcodes. Be aware that the resulting code cannot be run on 8086- and 8088-based machines.
- DPMIINST error* **A20 line already enabled, so test is meaningless**
 Message generated by DPMIINST when you run it (to locate and add information about your machine to the kernel's database). If CONFIG.SYS file contains the line `DOS=HIGH`, comment it out. Reboot and rerun DPMIINST. When DPMIINST runs successfully, remove the comment and reboot. If you encounter a *series* of these messages, boot clean (that is, with a plain generic CONFIG.SYS and AUTOEXEC.BAT) and try again. See the message **Machine not in database (run DPMIINST)** on page 680.
- Run-time error* **Abnormal program termination**
 The program called **abort** because there was not enough memory to execute. Can happen through memory overwrites.
- Compile-time error* **Access can only be changed to public or protected**
 A C++ derived class may modify the access rights of a base class member, but only to **public** or **protected**. A base class member cannot be made **private**.
- TLIB warning* **added file filename does not begin correctly, ignored**
 The librarian has decided that in no way, shape, or form is the file being added an object module, so it will not try to add it to the library. The library is created anyway.
- Compile-time error* **Address of overloaded function function doesn't match type**
 A variable or parameter is assigned/initialized with the address of an overloaded function, and the type of the variable/parameter doesn't match any of the overloaded functions with the specified name.

- TLIB warning* **module already in LIB, not changed!**
 An attempt to use the + action on the library has been made, but there is already a object with the same name in the library. If an update of the module is desired, the action should be +-. The library has not been modified.
- Compile-time error* **Ambiguity between *function1* and *function2***
 Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.
- Compile-time error* **Ambiguous member name *name***
 A structure member name used in inline assembly must be unique. If it is defined in more than one structure all of the definitions must agree in type and offset within the structures. The member name in this case is ambiguous. Use the syntax `(struct xxx).yyy` instead.
- Compile-time warning* **Ambiguous operators need parentheses**
 This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears unparenthesized with a shift operator will produce this warning. Programmers frequently confuse the precedence of these operators.
- Command line fatal error* **Application load & execute error 0001**
Application load & execute error FFE0
 There was insufficient extended memory available for the protected mode command line tool to load.
- Compile-time error* **Array allocated using new may not have an initializer**
 When initializing a vector (array) of classes, you must use the constructor that has no arguments. This is called the *default constructor*, which means that you may not supply constructor arguments when initializing such a vector.
- Compile-time error* **Array bounds missing]**
 Your source file declared an array in which the array bounds were not terminated by a right bracket.
- Compile-time error* **Array must have at least one element**
 ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed). An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with **malloc**. You can still use this trick, but you must declare the array element to have (at least) one element if you are

compiling in strict ANSI mode. Declarations (as opposed to definitions) of arrays of unknown size are still allowed, of course.

For example,

```
char ray[];          /* definition of unknown size -- illegal */
char ray[0];        /* definition of 0 size -- illegal */
extern char ray[];  /* declaration of unknown size -- ok */
```

Compile-time error **Array of references is not allowed**

It is illegal to have an array of references, since pointers to references are not allowed and array names are coerced into pointers.

Compile-time warning **Array size for 'delete' ignored**

With the latest specification of C++, it is no longer necessary to specify the array size when deleting an array; to allow older code to compile, the compiler ignores this construct, and issues this warning.

Compile-time error **Array size too large**

The declared array is larger than 64K.

Compile-time warning **Array variable *identifier* is near**

Whenever you use either the **-Ff** or **-Fm** command-line or the IDE Options | Compiler | Advanced Code Generation... | Far Data Threshold selection to set threshold limit, global variables larger than the threshold size are automatically made far by the compiler. However, when the variable is an initialized array with an unspecified size, its total size is not known when the decision whether to make it near or far has to be made by the compiler, and so it is made near. If the number of initializers given for the array causes the total variable size to exceed the data size threshold, the compiler issues this warning. If the fact that the variable is made near by the compiler causes problems (for example, the linker reports a group overflow due to too much global data), you must make the offending variable explicitly far by inserting the keyword **far** immediately to the left of the variable name in its definition.

Compile-time error **Assembler statement too long**

Inline assembly statements may not be longer than 480 bytes.

- Assigning type to enumeration**
- Compile-time warning* Assigning an integer value to an **enum** type. This is an error in C++, but is reduced to a warning to give existing programs a chance to work.
- Compile-time error* **Assignment to this not allowed, use X::operator new instead**
In early versions of C++, the only way to control allocation of class of objects was by assigning to the **this** parameter inside a constructor. This practice is no longer allowed, since a better, safer, and more general technique is to define a member function **operator new** instead.
- Compile-time error* **Attempt to grant or reduce access to identifier**
A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class. It cannot add or reduce access rights.
- Compile-time error* **Attempting to return a reference to a local object**
In a function returning a reference type, you attempted to return a reference to a temporary object (perhaps the result of a constructor or a function call). Since this object will disappear when the function returns, the reference will then be illegal.
- Compile-time error* **Attempting to return a reference to local variable identifier**
This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal, since the variable referred to disappears when the function exits. You may return a reference to any static or global variable, or you may change the function to return a value instead.
- Compile-time fatal error* **Bad call of intrinsic function**
You have used an intrinsic function without supplying a prototype, or you supplied a prototype for an intrinsic function that was not what the compiler expected.
- TLINK fatal error* **Bad character in parameters**
One of the following characters was encountered in the command line or in a response file:
“ * < = > ? [] |
or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.

- Compile-time error* **Bad define directive syntax**
A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.
- DPMI server fatal error* **bad environment params**
The value for the environmental variable DPMIMEM had incorrect syntax.
- Compile-time error* **Bad file name format in include directive**
Include file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by < > or " ".
- MAKE error* **Bad file name format in include statement**
Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.
- Compile-time error* **Bad file name format in line directive**
Line directive file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.
- TLIB warning* **bad GCD type in GRPDEF, extended dictionary aborted
bad GRPDEF type encountered, extended dictionary aborted**
The librarian has encountered an invalid entry in a group definition (GRPDEF) record in an object module while creating an extended dictionary. The only type of GRPDEF record that the librarian (and linker) supports are segment index type. If any other type of GRPDEF is encountered, the librarian won't be able to create an extended dictionary. It's possible that an object module created by products other than Borland tools may create GRPDEF records of other types. It's also possible for a corrupt object module to generate this warning.
- TLIB error* **Bad header in input LIB**
When adding object modules to an existing library, the librarian has determined that it has a bad library header. Rebuild the library.

- Compile-time error* **Bad ifdef directive syntax**
An **#ifdef** directive must contain a single identifier (and nothing else) as the body of the directive.
- MAKE error* **Bad macro output translator**
Invalid syntax for substitution within macros. For example:
`$(MODEL:=s) or $(MODEL:) or $(MODEL:s)`
- TLINK fatal error* **Bad object file record in library file *filename* in module *module* near module file offset 0xxxxxxx**
Bad object file record in module *filename* near module file offset 0xxxxxxx
An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was pressed.
- TLIB error* **bad OMF record type *type* encountered in module *module***
The librarian encountered a bad Object Module Format (OMF) record while reading through the object module. The librarian has already read and verified the header records on the *module*, so this usually indicates that the object module has become corrupt in some way and should be recreated.
- Compile-time error* **Bad syntax for pure function definition**
Pure virtual functions are specified by appending “= 0” to the declaration. You wrote something similar, but not quite the same.
- Compile-time error* **Bad undef directive syntax**
An **#undef** directive must contain a single identifier (and nothing else) as the body of the directive.
- MAKE error* **Bad undef statement syntax**
An **!undef** statement must contain a single identifier and nothing else as the body of the statement.
- TLINK fatal error* **Bad version number in parameter block**
This error indicates an internal inconsistency in the IDE. If it occurs, exit and restart the IDE. This error will not occur in the standalone version.

- Compile-time error* **Base class *class* contains dynamically dispatchable functions**
Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance. This error occurs because a class which contains DDVT function attempted to inherit DDVT functions from multiple parent classes.
- Compile-time warning* **Base class *class* is inaccessible because also in *class***
It is not legal to use a class as both a direct and indirect base class, since the members are automatically ambiguous. Try making the base class virtual in both locations.
- Compile-time error* **Base class *class* is included more than once**
A C++ class may be derived from any number of base classes, but may be directly derived from a given class only once.
- Compile-time error* **Base class *class* is initialized more than once**
In a C++ class constructor, the list of initializations following the constructor header includes base class *class* more than once.
- Compile-time error* **Base initialization without a class name is now obsolete**
Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list. It is now recommended to include the base class name.

This makes the code much clearer, and is required when there are multiple base classes.

Old way:

```
derived::derived(int i) : (i, 10) { ... }
```


New way:

```
derived::derived(int i) : base(i, 10) { ... }
```
- Compile-time error* **Bit field cannot be static**
Only ordinary C++ class data members can be declared **static**, not bit fields.
- Compile-time error* **Bit field too large**
This error occurs when you supply a bit field with more than 16 bits.
- Compile-time error* **Bit fields must be signed or unsigned int**
In ANSI C, bit fields may only be signed or unsigned **int** (not **char** or **long**, for example).

- Compile-time warning* **Bit fields must be signed or unsigned int**
In ANSI C, bit fields may not be of type signed char or unsigned char; when not compiling in strict ANSI mode, though, the compiler will allow such constructs, but flag them with this warning.
- Compile-time error* **Bit fields must contain at least one bit**
You cannot declare a named bit field to have 0 (or less than 0) bits. You can declare an unnamed bit field to have 0 bits, a convention used to force alignment of the following bit field to a byte boundary (or word boundary, if you select the `-a` alignment option or IDE Options | Compiler | Code Generation | Word Alignment). In C++, bit fields must have an integral type; this includes enumerations.
- Compile-time error* **Bit fields must have integral type**
In C++, bit fields must have an integral type; this includes enumerations.
- Compile-time error* **Body has already been defined for function *function***
A function with this name and type was previously supplied a function body. A function body can only be supplied once.
- Compile-time warning* **Both return and return with a value used**
The current function has `return` statements with and without values. This is legal in C, but almost always an error. Possibly a `return` statement was omitted from the end of the function.
- Compile-time error* **Call of nonfunction**
The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or misspelling the function name.
- Compile-time warning* **Call to function *function* with no prototype**
The "Prototypes required" warning was enabled and you called function *function* without first giving a prototype for that function.
- Compile-time error* **Cannot access an inactive scope**
This message used only by IDE debugger.
You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

Compile-time error

Cannot add or subtract relocatable symbols

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that *Var* is a variable and *Const* is a constant, then the instructions

```
MOV AX,Const+Const
```

and

```
MOV AX,Var+Const
```

are valid, but `MOV AX,Var+Var` is not.

Compile-time error

Cannot allocate a reference

An attempt to create a reference using the **new** operator has been made; this is illegal, as references are not objects and cannot be created through **new**.

Compile-time error

identifier cannot be declared in an anonymous union

The compiler found a declaration for a member function or static member in an anonymous union. Such unions can only contain data members.

Compile-time error

function1 cannot be distinguished from function2

The parameter type lists in the declarations of these two functions do not differ enough to tell them apart. Try changing the order of parameters or the type of a parameter in one declaration.

Compile-time error

Cannot call near class member function with a pointer of type type

Member functions of near classes (remember that classes are near by default in the tiny, small, and medium memory models) cannot be called using far or huge member pointers. (Note that this also applies to calls using pointers to members.) Either change the pointer to be near, or declare the class as far.

Compile-time error

Cannot cast from type1 to type2

A cast from type *type1* to type *type2* is not allowed. In C, a pointer may be cast to an integral type or to another pointer. An integral type may be cast to any integral, floating, or pointer type. A floating type may be cast to an integral or floating type. Structures and arrays may not be cast to or from. You cannot cast from a **void** type.

C++ checks for user-defined conversions and constructors, and if one cannot be found, then the preceding rules apply (except for pointers to class members). Among integral types, only a constant zero may be cast to a member pointer. A member pointer may be cast to an integral type or to a similar member pointer. A similar member pointer points to a data member if the original does, or to a function member if the original does; the qualifying class of the type being cast to must be the same as or a base class of the original.

- Compile-time error* **Cannot convert *type1* to *type2***
 An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal.
- Compile-time error* **Cannot create instance of abstract class *class***
 Abstract classes—those with pure virtual functions—cannot be used directly, only derived from.
- Compile-time error* **Cannot define a pointer or reference to a reference**
 It is illegal to have a pointer to a reference or a reference to a reference.
- Compile-time error* **Cannot find *class::class (class &)* to copy a vector**
 When a C++ class *class1* contains a vector (array) of class *class2*, and you want to construct an object of type *class1* from another object of type *class1*, there must be a constructor ***class2::class2(class2&)*** so that the elements of the vector can be constructed. This constructor takes just one parameter (which is a reference to its class) and is called a *copy constructor*.
 Usually the compiler supplies a copy constructor automatically. However, if you have defined a constructor for class *class2* that has a parameter of type *class2&* and has additional parameters with default values, the copy constructor cannot be created by the compiler. (This is because *class2::class2(class2&)* and *class2::class2(class2&, int = 1)* cannot be distinguished.) You must redefine this constructor so that not all parameters have default values. You can then define a copy constructor or let the compiler create one.
- Compile-time error* **Cannot find *class::operator=(class&)* to copy a vector**
 When a C++ class *class1* contains a vector (array) of class *class2*, and you wish to copy a class of type *class1*, there must be an assignment operator ***class2::operator=(class2&)*** so that the elements of the vector can be copied. Usually the compiler supplies such an operator automatically. However, if you have

defined an **operator=** for class *class2*, but not one that takes a parameter of type *class2&*, the compiler will not supply it automatically—you must supply one.

Compile-time error **Cannot find default constructor to initialize array element of type *class***

When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor (it will be used to initialize the array elements that don't have explicit initializers). The compiler will define a default constructor for a class unless you have defined any constructors for the class.

Compile-time error **Cannot find default constructor to initialize base class *class***

Whenever a C++ derived class *class2* is constructed, each base class *class1* must first be constructed. If the constructor for *class2* does not specify a constructor for *class1* (as part of *class2*'s header), there must be a constructor ***class1::class1()*** for the base class. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class1*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

Compile-time error **Cannot find default constructor to initialize member *identifier***

When a C++ class *class1* contains a member of class *class2*, and you wish to construct an object of type *class1* but not from another object of type *class1*, there must be a constructor ***class2::class2()*** so that the member can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class2*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

TLINK fatal error **Cannot generate COM file: data below initial CS:IP defined**

This error results from trying to generate data or code below the starting address (usually 100) of a .COM file. Be sure that the starting address is set to 100 by using the (ORG 100H) instruction. This error message should not occur for programs written in a high-level language. If it does, ensure that the correct startup (C0x) object module is being linked in.

- TLINK fatal error* **Cannot generate COM file: invalid initial entry point address**
 You used the `/Tdc` or `/t` option, but the program starting address is not equal to 100H, which is required with .COM files.
- TLINK fatal error* **Cannot generate COM file: program exceeds 64K**
 You used the `/Tdc` or `/t` option, but the total program size exceeds the .COM file limit.
- TLINK fatal error* **Cannot generate COM file: segment-relocatable items present**
 You used the `/Tdc` or `/t` option, but the program contains segment-relative fixups, which are not allowed with .COM files.
- TLINK fatal error* **Cannot generate COM file: stack segment present**
 You used the `/Tdc` or `/t` option, but the program declares a stack segment, which is not allowed with .COM files.
- Compile-time error* **Cannot generate *function* from template function *template***
 A call to a template function was found, but a matching template function cannot be generated from the function template.
- Compile-time error* **Cannot have a near class member in a far class**
 All members of a C++ **far** class must be far. This member is in a class that was declared (or defaults to) **near**.
- Compile-time error* **Cannot have a non-inline function in a local class**
Cannot have a static data member in a local class
 All members of classes declared local to a function must be entirely defined in the class definition. This means that such local classes may not contain any static data members, and all of their member functions must have bodies defined within the class definition.
- MAKE error* **Cannot have multiple paths for implicit rule**
 You can only have a single path for each of the extensions in an implicit rule. Multiple path lists are only allowed for dependents in an explicit rule. For example:
- ```

{path1;path2}.c.obj: # Invalid
{path}.c.obj # Valid

```
- MAKE error*    **Cannot have path list for target**  
 You can only specify a path list for dependents of an explicit rule. For example:
- ```

{path1;path2}prog.exe: prog.obj # Invalid
prog.exe: {path1;path2}prog.obj # Valid

```

- Compile-time error* **Cannot initialize a class member here**
Individual members of **structs**, **unions**, and C++ **classes** may not have initializers. A **struct** or **union** may be initialized as a whole using initializers inside braces. A C++ **class** may only be initialized by the use of a constructor.
- Compile-time error* **Cannot initialize *type1* with *type2***
You are attempting to initialize an object of type *type1* with a value of type *type2*, which is not allowed. The rules for initialization are essentially the same as for assignment.
- Compile-time error* **Cannot modify a const object**
This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.
- Compile-time error* **Cannot overload 'main'**
main is the only function which cannot be overloaded.
- Compile-time error* **function cannot return a value**
A function with a return type **void** contains a **return** statement that returns a value; for example, an **int**.
- Compile-time error* **identifier cannot start an argument declaration**
Undefined *identifier* found at the start of an argument in a function declarator. Often the type name is misspelled or the type declaration is missing (usually caused by not including the appropriate header file).
- TLIB error* **cannot write GRPDEF list, extended dictionary aborted**
The librarian cannot write the extended dictionary to the tail end of the library file. This usually indicates lack of space on the disk.
- TLIB error* **can't grow LE/LIDATA record buffer**
Command-line error. See **out of memory reading LE/LIDATA record from object module**.
- Compile-time error* **Case bypasses initialization of a local variable**
In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a case label which can transfer control past this local variable.

- Compile-time error* **Case outside of switch**
The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched braces.
- Compile-time error* **Case statement missing :**
A **case** statement must have a constant expression followed by a colon. The expression in the **case** statement either is missing a colon or has an extra symbol before the colon.
- MAKE or compile-time error* **Character constant must be one or two characters long**
Character constants can be only one or two characters long.
- MAKE fatal error* **Circular dependency exists in makefile**
The makefile indicates that a file needs to be up-to-date BEFORE it can be built. Take, for example, the explicit rules:
- ```
filea: fileb
fileb: filec
filec: filea
```
- This implies that `filea` depends on `fileb`, which depends on `filec`, and `filec` depends on `filea`. This is illegal, since a file cannot depend on itself, indirectly or directly.
- Compile-time error* **Class *class* may not contain pure functions**  
The class being declared cannot be abstract, and therefore it may not contain any pure functions.
- Compile-time error* **Class member *member* declared outside its class**  
C++ class member functions can be declared only inside the class declaration. Unlike nonmember functions, they cannot be declared multiple times or at other locations.
- Compile-time warning* **Code has no effect**  
The compiler encountered a statement with operators that have no effect. For example the statement
- ```
a .+ b;
```
- has no effect on either variable. The operation is unnecessary and probably indicates a bug in your file.
- MAKE error* **Colon expected**
You have forgotten to put the colon at the end of your implicit rule.
- ```
.c.obj: # Correct
.c.obj # Incorrect
```

- MAKE error* **Command arguments too long**  
The arguments to a command were more than the 127-character limit imposed by DOS.
- MAKE error* **Command syntax error**  
This message occurs if
- The first rule line of the makefile contained any leading whitespace.
  - An implicit rule did not consist of *.ext.ext:*.
  - An explicit rule did not contain a name before the `:` character.
  - A macro definition did not contain a name before the `=` character.
- MAKE error* **Command too long**  
The length of a command has exceeded 128 characters. You might want to use a response file.
- TLINK error* **Common segment exceeds 64K**  
The program had more than 64K of near uninitialized data. Try declaring some uninitialized data as far.
- Compile-time error* **Compiler could not generate copy constructor for class *class***  
The compiler cannot generate a needed copy constructor due to language rules.
- Compile-time error* **Compiler could not generate default constructor for class *class***  
The compiler cannot generate a needed default constructor due to language rules.
- Compile-time error* **Compiler could not generate operator= for class *class***  
The compiler cannot generate a needed assignment operator due to language rules.
- Compile-time fatal error* **Compiler table limit exceeded**  
One of the compiler's internal tables overflowed. This usually means that the module being compiled contains too many function bodies. Making more memory available to the compiler will not help with such a limitation; simplifying the file being compiled is usually the only remedy.
- Compile-time error* **Compound statement missing }**  
The compiler reached the end of the source file and found no closing brace. This is often caused by mismatched braces.

Compile-time warning **Condition is always false**

**Condition is always true**

The compiler encountered a comparison of values where the result is always true or false. For example:

```
void proc(unsigned x)
{
 if (x >= 0) /* always 'true' */
 {
 :
 }
}
```

Compile-time error **Conflicting type modifiers**

This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) may be given for a function.

TLINK warning **symbol conflicts with module *module* in module *module***

This indicates an inconsistency in the definition of *symbol*; TLINK found one virtual function and one common definition with the same name.

Compile-time error **Constant expression required**

Arrays must be declared with constant size. This error is commonly caused by misspelling a **#define** constant.

Compile-time warning **Constant is long**

The compiler encountered either a decimal constant greater than 32767 or an octal (or hexadecimal) constant greater than 65535 without a letter *l* or *L* following it. The constant is treated as a **long**.

Compile-time error **Constant member *member* in class without constructors**

A class that contains constant members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.

Compile-time warning **Constant member *member* is not initialized**

This C++ class contains a constant member *member*, which does not have an initialization. Note that constant members may be initialized only, not assigned to.

*Compile-time warning*

**Constant out of range in comparison**

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an **unsigned** quantity to `-1` makes no sense. To get an **unsigned** constant greater than 32767 (in decimal), you should either cast the constant to **unsigned** (for example, `(unsigned)65535`) or append a letter `u` or `U` to the constant (for example, `65535u`).

Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a **char** expression to 4000, the code will still perform the test.

*Compile-time error*

**Constant variable *variable* must be initialized**

This C++ object is declared **const**, but is not initialized. Since no value may be assigned to it, it must be initialized at the point of declaration.

*Compile-time error*

**constructor cannot be declared const or volatile**

A constructor has been declared as **const** and/or **volatile**, and this is not allowed.

*Compile-time error*

**constructor cannot have a return type specification**

C++ constructors have an implicit return type used by the compiler, but you cannot declare a return type or return a value.

*Compile-time warning*

**Conversion may lose significant digits**

For an assignment operator or some other circumstance, your source file requires a conversion from **long** or **unsigned long** to **int** or **unsigned int** type. Since **int** type and **long** type variables don't have the same size, this kind of conversion may alter the behavior of a program.

*Compile-time error*

*This message used only by  
IDE debugger.*

**Conversion of near pointer not allowed**

A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.

*Compile-time error*

**Conversion operator cannot have a return type specification**

This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function **operator** may not specify any return type.

- Compile-time error*    **Conversion to *type* will fail for members of virtual base *class***  
 This warning is issued in some cases when a member pointer is cast to another member pointer type, if the class of the member pointer contains virtual bases, and only if the **-Vv** option or IDE Options | Compiler | Advanced Compiler | Deep Virtual Bases has been used. It means that if the member pointer being cast happens to point (at the time of the cast) to a member of ***class***, the conversion cannot be completed, and the result of the cast will be a NULL member pointer.
- TLIB error*    **could not allocate memory for per module data**  
 The librarian has run out of memory.
- TLIB error*    **could not create list file *filename***  
 The librarian could not create a list file for the library. This could be due to lack of disk space.
- Compile-time error*    **Could not find a match for *argument(s)***  
 No C++ function could be found with parameters matching the supplied arguments.
- Compile-time error*    **Could not find file *filename***  
 The compiler is unable to find the file supplied on the command line.
- TLIB error*    **Could not write output.**  
 The librarian could not write the output file.
- TLIB error*    **couldn't alloc memory for per module data**  
 The librarian has run out of memory.
- TLIB warning*    ***filename* couldn't be created, original won't be changed**  
 An attempt has been made to extract an object ("*\**" action) but the librarian cannot create the object file to extract the module into. Either the object already exists and is read only, or the disk is full.
- TLIB error*    **couldn't get LE/LIDATA record buffer**  
 Command-line error. See **out of memory reading LE/LIDATA record from object module.**
- TLINK warning*    **Debug info switch ignored for .COM files**  
 Turbo C++ does not include debug information for .COM files. See the description of the **/v** option on page 344.
- TLINK warning*    **Debug information in module *module* will be ignored**  
 Object files compiled with debug information now have a version record. The major version of this record is higher than



what TLINK currently supports and TLINK did not generate debug information for the module in question.

- Compile-time error* **Declaration does not specify a tag or an identifier**  
This declaration doesn't declare anything. This may be a **struct** or **union** without a tag or a variable in the declaration. C++ requires that something be declared.
- Compile-time error* **Declaration is not allowed here**  
Declarations cannot be used as the control statement for **while**, **for**, **do**, **if**, or **switch** statements.
- Compile-time error* **Declaration missing ;**  
Your source file contained a declaration that was not followed by a semicolon.
- Compile-time error* **Declaration syntax error**  
Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.
- Compile-time error* **Declaration terminated incorrectly**  
A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body. A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.
- Compile-time error* **Declaration was expected**  
A declaration was expected here but not found. This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.
- Compile-time error* **Declare operator delete (void\*) or (void\*, size\_t)**  
Declare the operator **delete** with a single **void\*** parameter, or with a second parameter of type **size\_t**. If you use the second version, it will be used in preference to the first version. The global operator **delete** can only be declared using the single-parameter form.
- Compile-time warning* **Declare type type prior to use in prototype**  
When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype. For example,
- ```
int func(struct s *ps);
struct s { /* ... */};
```
- Since there is no struct *s* in scope at the prototype for **func**, the type of parameter *ps* is pointer to undefined struct *s*, and is not

the same as the struct *s* which is later declared. This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message. To fix the problem, you can move the declaration for struct *s* ahead of any prototype which references it, or add the incomplete type declaration `struct s;` ahead of any prototype which references struct *s*. If the function parameter is a **struct**, rather than a pointer to **struct**, the incomplete declaration is not sufficient; you must then place the struct declaration ahead of the prototype.

- Compile-time warning* **Identifier is declared but never used**
 Your source file declared the named variable as part of the block just ending, but the variable was never used. The warning is indicated when the compiler encounters the closing brace of the compound statement or function. The declaration of the variable occurs at the beginning of the compound statement or function.
- Compile-time error* **Default argument value redeclared for parameter *parameter***
 When a parameter of a C++ function is declared to have a default value, this value cannot be changed, redeclared, or omitted in any other declaration for the same function.
- Compile-time error* **Default expression may not use local variables**
 A default argument expression is not allowed to use any local variables or other parameters.
- Compile-time error* **Default outside of switch**
 The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched braces.
- Compile-time error* **Default value missing**
 When a C++ function declares a parameter with a default value, all of the following parameters must also have default values. In this declaration, a parameter with a default value was followed by a parameter without a default value.
- Compile-time error* **Default value missing following parameter *parameter***
 All parameters following the first parameter with a default value must also have defaults specified.
- Compile-time error* **Define directive needs an identifier**
 The first non-whitespace character after a **#define** must be an identifier. The compiler found some other character.

- TLINK error or warning* **symbol defined in module *module* is duplicated in module *module***
There is a conflict between two symbols (either public or communal). This usually means that a symbol is defined in two modules. An error occurs if both are encountered in the .OBJ file(s), because TLINK doesn't know which is valid. A warning results if TLINK finds one of the duplicated symbols in a library and finds the other in an .OBJ file; in this case, TLINK uses the one in the .OBJ file.
- Compile-time error* **Delete array size missing]**
The array specifier in an operator is missing a right bracket.
- Compile-time error* **Destructor cannot be declared const or volatile**
A destructor has been declared as **const** and/or **volatile**, and this is not allowed.
- Compile-time error* **Destructor cannot have a return type specification**
It is illegal to specify the return type for a destructor.
- Compile-time error* **Destructor for *class* is not accessible**
The destructor for this C++ class is **protected** or **private**, and cannot be accessed here to destroy the class. If a class destructor is **private**, the class cannot be destroyed, and thus can never be used. This is probably an error. A **protected** destructor can be accessed only from derived classes. This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.
- Compile-time error* **Destructor for *class* required in conditional expression**
If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor, since the variable may or may not have been initialized. The temporary variable can be explicitly created, as with `classname (val, val)`, or implicitly created by some other code. Recast your code to eliminate this temporary value.
- Compile-time error* **Destructor name must match the class name**
In a C++ class, the tilde (~) introduces a declaration for the class destructor. The name of the destructor must be the same as the class name. In your source file, the tilde (~) preceded some other name.

- Run-time error* **Divide error**
 You've tried to divide an integer by zero. For example,
- ```
int n = 0;
n = 2 / n;
```
- You can trap this error with the signal function. Otherwise, Turbo C++ calls **abort** and your program terminates.
- Compile-time error* **Division by zero**  
 Your source file contained a division or remainder operator in a constant expression with a zero divisor.
- Compile-time warning* **Division by zero**  
 A division or remainder operator expression had a literal zero as a divisor.
- MAKE error* **Division by zero**  
 A division or remainder operator in an **!if** statement has a zero divisor.
- Compile-time error* **do statement must have while**  
 Your source file contained a **do** statement that was missing the closing **while** keyword.
- MAKE fatal error* **filename does not exist – don't know how to make it**  
 There's a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.
- TLINK fatal error* **DOS error, ax = number**  
 This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes where this error could occur are read, write, seek, and close.
- Compile-time error* **do-while statement missing (**  
 In a **do** statement, the compiler found no left parenthesis after the **while** keyword.
- Compile-time error* **do-while statement missing )**  
 In a **do** statement, the compiler found no right parenthesis after the test expression.
- Compile-time error* **do-while statement missing ;**  
 In a **do** statement test expression, the compiler found no semicolon after the right parenthesis.

- Compile-time error*    **Duplicate case**  
Each **case** of a **switch** statement must have a unique constant expression value.
- Compile-time error*    **Enum syntax error**  
An **enum** declaration did not contain a properly formed list of identifiers.
- TLIB error*    **error changing file buffer size**  
TLIB is attempting to adjust the size of a buffer used while reading or writing a file but there is not enough memory. It is likely that quite a bit of system memory will have to be freed up to resolve this error.
- Compile-time fatal error*    **Error directive: *message***  
The text of the **#error** directive being processed in the source file is displayed.
- MAKE fatal error*    **Error directive: *message***  
MAKE has processed an **#error** directive in the source file, and the text of the directive is displayed in the message.
- TLIB error*    **error opening *filename***  
TLIB cannot open the specified file for some reason.
- TLIB error*    **error opening *filename* for output**  
TLIB cannot open the specified file for output. This is usually due to lack of disk space for the target library, or a listing file. Additionally this error will occur when the target file exists but is marked as a read only file.
- TLIB error*    **error renaming *filename* to *filename***  
TLIB builds a library into a temporary file and then renames the temporary file to the target library file name. If there is an error, usually due to lack of disk space, this message will be posted.
- Compile-time fatal error*    **Error writing output file**  
A DOS error that prevents Turbo C++ from writing an .OBJ, .EXE, or temporary file. Check the **-n** or Options | Directories | Output directory and make sure that this is a valid directory. Also check that there is enough free disk space.
- Compile-time error*    **Expression expected**  
An expression was expected here, but the current symbol cannot begin an expression. This message may occur where the controlling expression of an **if** or **while** clause is expected or

where a variable is being initialized. It is often due to an accidentally inserted or deleted symbol in the source code.

*Compile-time error*

**Expression of scalar type expected**

The not (!), increment (++), and decrement (–) operators require an expression of scalar type—only types **char**, **short**, **int**, **long**, **enum**, **float**, **double**, **long double**, and pointer types are allowed.

*Compile-time error*

**Expression syntax**

This is a catchall error message when the compiler parses an expression and encounters some serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

*MAKE error*

**Expression syntax error in `!if` statement**

The expression in an `!if` statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

*TLIB warning*

***reason* – extended dictionary not created**

TLIB could not produce the extended dictionary because of the *reason* given in the warning message.

*Compile-time error*

**extern variable cannot be initialized**

The storage class **extern** applied to a variable means that the variable is being declared but not defined here—no storage is being allocated for it. Therefore, you can't initialize the variable as part of the declaration.

*Compile-time error*

**Extra argument in template class name *template***

A template class name specified too many actual values for its formal parameters.

*Compile-time error*

**Extra parameter in call**

A call to a function, via a pointer defined with a prototype, had too many arguments given.

*Compile-time error*

**Extra parameter in call to *function***

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

*Command line fatal error*

**Failed to locate DPMI server (DPMI16BI.OVL)**

**Failed to locate protected mode loader (DPMILOAD.EXE)**

Make sure that DPMI16BI.OVL and DPMILOAD.EXE are somewhere on your path or in the same directory as the

protected mode command line tool you were attempting to use.

- Compile-time error* **File must contain at least one external declaration**  
This compilation unit was logically empty, containing no external declarations. ANSI C and C++ require that something be declared in the compilation unit.
- Compile-time error* **File name too long**  
The file name given in an **#include** directive was too long for the compiler to process. Path names in DOS must be no more than 79 characters long.
- MAKE error* **File name too long**  
The path name in an **!include** directive overflowed MAKE's internal buffer (512 bytes).
- TLIB warning* **filename file not found**  
The command-line librarian attempted to add a nonexisting object but created the library anyway.
- TLIB error* **filename file not found**  
The IDE creates the library by first removing the existing library and then rebuilding. If any objects do not exist, the library is considered incomplete and thus an error. If the IDE reports that an object does not exist, either the source module has not been compiled or there were errors during compilation. Performing either a **Compile | Make** or **Compile | Build** should resolve the problem or indicate where the errors have occurred.
- TLINK fatal error* **filename (linenum): File read error**  
A DOS error occurred while TLINK read the module definition file. This usually means that a premature end of file occurred.
- TLINK error* **Fixup overflow at segment:xxxxh, target = segment:xxxxh in module module**  
**Fixup overflow at segment:xxxxh, target = symbol in module module**  
Either of these messages indicate an incorrect data or code reference in an object file that TLINK must fix up at link time.  
The cause is often a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named

module, so look in the source file of that module for the offending reference.

In an assembly language program, a fixup overflow frequently occurs if you have declared an external variable within a segment definition, but this variable actually exists in a different segment.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or a high-level language besides Turbo C++, there may be other possible causes for this message. Even in Turbo C++, this message could be generated if you are using different segment or group names than the default values for a given memory model.

*Run-time error*

**Floating point error: Divide by 0.**

**Floating point error: Domain.**

**Floating point error: Overflow.**

These fatal errors result from a floating-point operation for which the result is not finite.

- "Divide by 0" means the result is +INF or -INF exactly, such as 1.0/0.0.
- "Domain" means the result is NAN (not a number), like 0.0 /0.0.
- "Overflow" means the result is +INF (infinity) or -INF with complete loss of precision, such as assigning 1e200\*1e200 to a **double**.

*Run-time error*

**Floating point error: Partial loss of precision.**

**Floating point error: Underflow.**

These exceptions are masked by default, and the error messages do not occur. Underflows are converted to zero and losses of precision are ignored. They can be unmasked by calling **\_control87**.

*Run-time error*

**Floating point error: Stack fault.**

The floating-point stack has been overrun. This error does not normally occur and may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

These floating-point errors can be avoided by masking the exception so that it doesn't occur, or by catching the exception with **signal**. See the functions **\_control87** and **signal** (in online Help) for details.



In each of the above cases, the program prints the error message and then calls **abort**, which prints

Abnormal program termination

and calls `_exit(3)`. See **abort** and **\_exit** for more details.

*Compile-time error* **For statement missing (**  
In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

*Compile-time error* **For statement missing )**  
In a **for** statement, the compiler found no right parenthesis after the control expressions.

*Compile-time error* **For statement missing ;**  
In a **for** statement, the compiler found no semicolon after one of the expressions.

*Compile-time error* **Friends must be functions or classes**  
A **friend** of a C++ class must be a function or another class.

*Compile-time error* **Function call missing )**  
The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

*Compile-time error* **Function calls not supported**  
*This message used only by IDE debugger.*  
In integrated debugger expression evaluation, calls to functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

*Compile-time error* **Function defined inline after use as extern**  
Functions cannot become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely.

*Compile-time error* **Function definition cannot be a Typedef'ed declaration**  
In ANSI C a function body cannot be defined using a typedef with a function Type.

*Compile-time error* **Function *function* cannot be static**  
Only ordinary member functions and the operators **new** and **delete** can be declared static. Constructors, destructors and other operators must not be static.

*Compile-time error* **Function *function* should have a prototype**  
A function was called with no prototype in scope.

In C, `int foo();` is not a prototype, but `int foo(int);` is, and so is `int foo(void);`. In C++, `int foo();` is a prototype, and is the same as `int foo(void);`. In C, prototypes are *recommended* for all functions. In C++, prototypes are *required* for all functions. In all cases, a function definition (a function header with its body) serves as a prototype if it appears before any other mention of the function.

*Compile-time warning*

**Function should return a value**

This function was declared (maybe implicitly) to return a value. A **return** statement was found without a return value or the end of the function was reached without a **return** statement being found. Either return a value or declare the function as **void**.

*Compile-time error*

**Function should return a value**

Your source file declared the current function to return some type other than **void** in C++ (or **int** in C), but the compiler encountered a return with no value. This is usually some sort of error. In C **int** functions are exempt, since in old versions of C there was no **void** type to indicate functions which return nothing.

*Compile-time error*

**Functions *function1* and *function2* both use the same dispatch number**

Dynamically dispatched virtual table (DDVT) problem. When you override a dynamically dispatchable function in a derived class, use the same dispatch index. Each function within the same class hierarchy must use a different dispatch index.

*Compile-time warning*

**Functions containing local destructors are not expanded inline in function *function***

You've created an inline function for which Turbo C++ turns off inlining. You can ignore this warning if you like; the function will be generated out of line.

*Compile-time warning*

**Functions containing *reserved word* are not expanded inline**

Functions containing any of the reserved words **do**, **for**, **while**, **goto**, **switch**, **break**, **continue**, and **case** cannot be expanded inline, even when specified as **inline**. The function is still perfectly legal, but will be treated as an ordinary static (not global) function.

*Compile-time error*

**Functions may not be part of a struct or union**

This C **struct** or **union** field was declared to be of type function rather than pointer to function. Functions as fields are allowed only in C++.

- TLINK fatal error*    **General error**  
**General error in library file *filename* in module *module* near module file offset 0xyyyyyyyy.**  
**General error in module *module* near module file offset 0xyyyyyyyy**  
 TLINK gives as much information as possible about what processing was happening at the time of the unknown fatal error. Call Technical Support with information about .OBJ or .LIB files.
- Compile-time error*    **Global anonymous union not static**  
 In C++, a global anonymous union at the file level must be static.
- Compile-time error*    **Goto bypasses initialization of a local variable**  
 In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a **goto** which can transfer control past this local variable.
- Compile-time error*    **Goto statement missing label**  
 The **goto** keyword must be followed by an identifier.
- TLINK fatal error*    **Group *group* exceeds 64K**  
 A group exceeded 64K bytes when the segments of the group were combined.
- Compile-time error*    **Group overflowed maximum size: *group***  
 The total size of the segments in a group (for example, DGROUP) exceeded 64K.
- TLINK warning*    **Group *group1* overlaps group *group2***  
 This means that TLINK has encountered nested groups. This warning only occurs when overlays are used.
- Compile-time error*    ***specifier* has already been included**  
 This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.
- Compile-time warning*    **Hexadecimal value contains more than 3 digits**  
 Under older versions of C, a hexadecimal escape sequence could contain no more than three digits. The ANSI standard allows any number of digits to appear as long as the value fits in a byte. This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as "\x00045"). Older versions of C would interpret such a string differently.

- function1 hides virtual function function2**
- Compile-time warning* A virtual function in a base class is usually overridden by a declaration in a derived class. In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.
- Compile-time error* **Identifier expected**
- An identifier was expected here, but not found. In C, this is in a list of parameters in an old-style function header, after the reserved words **struct** or **union** when the braces are not present, and as the name of a member in a structure or union (except for bit fields of width 0). In C++, an identifier is also expected in a list of base classes from which another class is derived, following a double colon (::), and after the reserved word **operator** when no operator symbol is present.
- Compile-time error* **Identifier *identifier* cannot have a type qualifier**
- A C++ qualifier *class::identifier* may not be applied here. A qualifier is not allowed on **typedef** names, on function declarations (except definitions at the file level), on local variables or parameters of functions, or on a class member except to use its own class as a qualifier (redundant but legal).
- Compile-time error* **If statement missing (**
- In an **if** statement, the compiler found no left parenthesis after the **if** keyword.
- Compile-time error* **If statement missing )**
- In an **if** statement, the compiler found no right parenthesis after the test expression.
- MAKE error* **If statement too long**  
**ifdef statement too long**  
**ifndef statement too long**
- An **if**, **ifdef**, or **ifndef** statement has exceeded 4,096 characters.
- TLIB warning* **ignored *module*, path is too long**
- The path to a specified **.obj** or **.lib** file is greater than 64 characters. The max path to a file for TLIB is 64 characters.
- Compile-time error* **Illegal character *character* (0x*value*)**
- The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed. This can also be caused by extra parameters passed to a function macro.

- MAKE error*    **Illegal character in constant expression *expression***  
MAKE encountered a character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.
- TLINK fatal error*    **Illegal group definition: *group* in module *module***  
This error results from a malformed GRPDEF record in an .OBJ file. This latter case could result from custom-built .OBJ files or a bug in the translator used to generate the .OBJ file. If this occurs in a file created by Turbo C++, recompile the file. If the error persists, contact Borland.
- Compile-time error*    **Illegal initialization**  
In C, initializations must be either a constant expression, or else the address of a global **extern** or **static** variable plus or minus a constant.
- MAKE or compile-time error*    **Illegal octal digit**  
An octal constant was found containing a digit of 8 or 9.
- Compile-time error*    **Illegal parameter to `__emit__`**  
You supplied an argument to **emit** which is not a constant or an address.
- Compile-time error*    **Illegal pointer subtraction**  
This is caused by attempting to subtract a pointer from a non-pointer.
- Compile-time error*    **Illegal structure operation**  
In C or C++, structures may be used with dot (`.`), address-of (`&`), or assignment (`=`) operators, or be passed to or from functions as parameters. In C or C++, structures can also be used with overloaded operators. The compiler encountered a structure being used with some other operator.
- Compile-time error*    **Illegal to take address of bit field**  
It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.
- Compile-time error*    **Illegal use of floating point**  
Floating-point operands are not allowed in shift, bitwise Boolean, indirection (`*`), or certain other operators. The compiler found a floating-point operand with one of these prohibited operators.

- Compile-time error*    **Illegal use of member pointer**  
 Pointers to class members can only be used with assignment, comparison, the `.*`, `->*`, `?:`, `&&` and `||` operators, or passed as arguments to functions. The compiler has encountered a member pointer being used with a different operator.
- Compile-time error*    **Illegal use of pointer**  
 Pointers can only be used with addition, subtraction, assignment, comparison, indirection (`*`) or arrow (`->`). Your source file used a pointer with some other operator.
- Compile-time warning*    **Ill-formed pragma**  
 A pragma does not match one of the pragmas expected by the Turbo C++ compiler.
- Compile-time error*    **Implicit conversion of *type1* to *type2* not allowed**  
 When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to point to the appropriate base class. In this case, such an implicit conversion is illegal.
- Compile-time error*    **Improper use of typedef identifier**  
 Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.
- TLINK fatal error*    **filename (linenum): Incompatible attribute**  
 TLINK encountered incompatible segment attributes in a CODE or DATA statement. For instance, both PRELOAD and LOADONCALL can't be attributes for the same segment.
- Compile-time error*    **Incompatible type conversion**  
 The cast requested can't be done. Check the types.
- MAKE fatal error*    **Incorrect command-line argument: *argument***  
 You've used incorrect command-line arguments.
- Compile-time error*    **Incorrect command-line option: *option***  
 The compiler did not recognize the command-line parameter as legal.
- Compile-time error*    **Incorrect configuration file option: *option***  
 The compiler did not recognize the configuration file parameter as legal; check for a preceding hyphen (-).
- Compile-time error*    **Incorrect number format**  
 The compiler encountered a decimal point in a hexadecimal number.

*Compile-time error*    **Incorrect use of default**

The compiler found no colon after the **default** keyword in a **case** statement.

*Compile-time warning*    **Initializing enumeration with type**

You're trying to initialize an **enum** variable to a different type. For example,

```
enum count { zero, one, two } x = 2;
```

will result in this warning, because 2 is of type **int**, not type **enum count**. It is better programming practice to use an **enum** identifier instead of a literal integer when assigning to or initializing **enum** types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

*Compile-time error*    **Inline assembly not allowed in inline and template functions**

The compiler cannot handle inline assembly statements in a C++ inline or template function. You could eliminate the inline assembly code or, in case of an inline function, make this a macro, or remove the **inline** storage class.

*DPMI server fatal error*    **Insufficient memory available to initialize application**

You need to free some conventional memory to initialize the DPMI server.

*MAKE error*    **Int and string types compared**

You have tried to compare an integer operand with a string operand in an **if** or **!elif** expression.

*TLINK fatal error*    **Internal linker error *errorcode***

An error occurred in the internal logic of TLINK. This error shouldn't occur in practice, but is listed here for completeness in the event that a more specific error isn't generated. If this error persists, write down the *errorcode* number and contact Borland.

*Compile-time error*    **Invalid combination of opcode and operands**

The built-in assembler does not accept this combination of operands. Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, **INC AX,BX**, or **MOV AX**.
- The number of operands is correct, but their types or order do not match the opcode; for example **DEC 1, MOV AX,CL**,

or **MOV 1,AX**. Try prefacing the operands with type overrides; for example **MOV AX, WORD PTR foo**.

- TLINK error* **Invalid entry point offset**  
This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.
- Compile-time error* **Invalid indirection**  
The indirection operator (\*) requires a non-void pointer as the operand.
- TLINK fatal error* **Invalid initial stack offset**  
This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.
- Compile-time error* **Invalid macro argument separator**  
In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.
- TLIB warning* **invalid page size value ignored**  
Invalid page size is given. The page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.
- Compile-time error* **Invalid pointer addition**  
Your source file attempted to add two pointers together.
- Compile-time error* **Invalid register combination (e.g. [BP+BX])**  
The built-in assembler detected an illegal combination of registers in an instruction. Valid index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. Other index register combinations (such as [AX], [BP+BX], and [SI+DX]) are not allowed.
- ➔ Local variables (variables declared in procedures and functions) are usually allocated on the stack and accessed via the BP register. The assembler automatically adds [BP] in references to such variables, so even though a construct like **Local[BX]** (where **Local** is a local variable) appears valid, it is not since the final operand would become **Local[BP+BX]**.
- TLINK fatal error* **Invalid segment definition in module *module***  
The compiler produced a flawed object file. If this occurs in a file created by Turbo C++, recompile the file. If the problem persists, contact Borland.



*Compile-time error*    **Invalid template argument list**  
In a template declaration, the keyword **template** must be followed by a list of formal arguments enclosed within the < and > delimiters; an illegal template argument list was found.

*Compile-time error*    **Invalid template qualified name *template::name***  
When defining a template class member, the actual arguments in the template class name that is used as the left operand for the :: operator must match the formal arguments of the template class. For example:

```
template <class T> class X
{
 void f();
};

template <class T> void X<T>::f() {}
```

The following would be illegal:

```
template <class T> void X<int>::f() {}
```

*Compile-time error*    **Invalid use of dot**  
An identifier must immediately follow a period operator (.).

*Compile-time error*    **Invalid use of template *template***  
Outside of a template definition, it is illegal to use a template class name without specifying its actual arguments. For example, you can use **vector<int>** but not **vector**.

*Compile-time fatal error*    **Irreducible expression tree**  
This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. Whatever the offending expression is, it should be avoided. Notify Borland if the compiler ever encounters this error.

*Compile-time error*    **base is an indirect virtual base class of *class***  
A pointer to a C++ member of the given virtual base class cannot be created; an attempt has been made to create such a pointer (either directly, or through a cast). See page 293 for information on the **-Vv** switch.

*Compile-time warning*    **identifier is assigned a value that is never used**  
The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing brace.

- identifier is declared as both external and static**  
 Compile-time warning This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration. The identifier is taken as static. You should review all declarations for this identifier.
- symbol is duplicated in module module**  
 TLINK error or warning There is a conflict between two symbols (either public or communal) defined in the same module. An error occurs if both are encountered in an .OBJ file. A warning is issued if TLINK finds the duplicates in a library; in this case, TLINK uses the first definition.
- constructor is not a base class of class**  
 Compile-time error A C++ class constructor *class* is trying to call a base class constructor *constructor*, or you are trying to change the access rights of *class::constructor*. *constructor* is not a base class of *class*. Check your declarations.
- identifier is not a member of struct**  
 Compile-time error You are trying to reference *identifier* as a member of **struct**, but it is not a member. Check your declarations.
- identifier is not a non-static data member and can't be initialized here**  
 Compile-time error Only data members can be initialized in the initializers of a constructor. This message means that the list includes a static member or function member.
- identifier is not a parameter**  
 Compile-time error In the parameter declaration section of an old-style function definition, *identifier* is declared but is not listed as a parameter. Either remove the declaration or add *identifier* as a parameter.
- identifier is not a public base class of classtype**  
 Compile-time error The right operand of a *.\**, *->\**, or *::operator* was not a pointer to a member of a class that is either identical to or an unambiguous accessible base class of the left operand's class type.
- member is not accessible**  
 Compile-time error You are trying to reference C++ class member *member*, but it is **private** or **protected** and cannot be referenced from this function. This sometimes happens when attempting to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function. The check for overload resolution is always made before checking for

accessibility. If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

*Compile-time error*

**Last parameter of operator must have type int**

When a postfix **operator++** or **operator--** is declared, the last parameter must be declared with the type **int**.

*TLIB warning*

**library contains COMDEF records – extended dictionary not created**

An object record being added to a library contains a COMDEF record. This is not compatible with the extended dictionary option.

*TLIB error*

**library too large, please restart with /P size  
library too large, restart with library page size size**

The library being created could not be built with the current library page size. In the IDE, the library page size can be set from the Options | Librarian dialog box.

*Compile-time error*

**Linkage specification not allowed**

Linkage specifications such as **extern "C"** are only allowed at the file level. Move this function declaration out to the file level.

*TLINK fatal error*

**Linker stack overflow**

TLINK uses a recursive procedure for marking modules to be included in an executable image from libraries. This procedure can cause stack overflows in extreme circumstances. If you get this error message, remove some modules from libraries, include them with the object files in the link, and try again.

*Compile-time error*

**Lvalue required**

The left hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.

*DPMI server fatal error*

**Machine not in database (run DPMIINST)**

The Dos Protected Mode Interface (DPMI) server searched the kernel's database and could not locate information about your machine. Run DPMIINST (several times, if necessary) to update the database. DPMIINST also generates a .DB file for you to send to Borland. See also **A20 line already enabled, so test is meaningless** on page 644.

- Compile-time error*    **Macro argument syntax error**  
An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.
- Compile-time error*    **Macro expansion too long**  
A macro cannot expand to more than 4,096 characters.
- MAKE error*    **Macro expansion too long**  
A macro cannot expand to more than 4,096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.
- MAKE fatal errors*    **Macro substitute text *string* is too long**  
**Macro replace text *string* is too long**  
The macro substitution or replacement text *string* overflowed MAKE's internal buffer of 512 bytes.
- Compile-time error*    **main must have a return type of int**  
In C++, function **main** has special requirements, one of which is that it cannot be declared with any return type other than **int**. See Chapter 15,
- Compile-time error*    **Matching base class function for *function* has different dispatch number.**  
If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.
- Compile-time error*    **Matching base class function for *function* is not dynamic**  
If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.
- Compile-time warning*    **Maximum precision used for member pointer type *type***  
When a member pointer type is declared, its class has not been fully defined, and the **-Vmd** option has been used, the compiler has to use the most general (and the least efficient) representation for that member pointer type. This may not only cause less efficient code to be generated (and make the member pointer type unnecessarily large), but it can also cause problems with separate compilation; see the **-Vm** compiler switch discussion in Chapter 8, "The command-line compiler" for details.

- Compile-time error*    **Member function must be called or its address taken**  
When a member function is used in an expression, either it must be called, or its address must be taken using the **&** operator. In this case, a member function has been used in an illegal context.
- Compile-time error*    **Member identifier expected**  
The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (->) operator must be the name of a member in the structure or class on the left of the operator.
- Compile-time error*    **Member is ambiguous: *member1* and *member2***  
You must qualify the member reference with the appropriate base class name. In C++ class *class*, member *member* can be found in more than one base class, and was not qualified to indicate which was meant. This happens only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path. The C++ language rules require that this test for ambiguity be made before checking for access rights (**private**, **protected**, **public**). It is therefore possible to get this message even though only one (or none) of the members can be accessed.
- Compile-time error*    **Member *member* cannot be used without an object**  
This means that the user has written *class::member* where *member* is an ordinary (non-static) member, and there is no class to associate with that member. For example, it is legal to write *obj.class::member*, but not to write *class::member*.
- Compile-time error*    **Member *member* has the same name as its class**  
A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class. Only a member function or a non-static member may have a name that is identical to its class.
- Compile-time error*    **Member *member* is initialized more than once**  
In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.
- Compile-time error*    **Member pointer required on right side of .\* or ->\***  
The right side of a C++ dot-star (.\* ) or an arrow-star (->\*) operator must be declared as a pointer to a member of the class

specified by the left side of the operator. In this case, the right side is not a member pointer.

- TLIB warning* **Memory full listing truncated!**  
The librarian has run out of memory creating a library listing file. A list file will be created but is not complete.
- Compile-time error* **Memory reference expected**  
The built-in assembler requires a memory reference. Most likely you have forgotten to put square brackets around an index register operand; for example, **MOV AX,BX+SI** instead of **MOV AX,[BX+SI]**.
- Compile-time error* **Misplaced break**  
The compiler encountered a **break** statement outside a **switch** or looping construct.
- Compile-time error* **Misplaced continue**  
The compiler encountered a **continue** statement outside a looping construct.
- Compile-time error* **Misplaced decimal point**  
The compiler encountered a decimal point in a floating-point constant as part of the exponent.
- Compile-time error* **Misplaced elif directive**  
The compiler encountered an **#elif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.
- MAKE error* **Misplaced elif statement**  
An **!elif** directive is missing a matching **!if** directive.
- Compile-time error* **Misplaced else**  
The compiler encountered an **else** statement without a matching **if** statement. An extra **else** statement could cause this message, but it could also be caused by an extra semicolon, missing braces, or some syntax error in a previous **if** statement.
- Compile-time error* **Misplaced else directive**  
The compiler encountered an **#else** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.
- MAKE error* **Misplaced else statement**  
There's an **!else** directive without any matching **!if** directive.

- Compile-time error*    **Misplaced endif directive**  
The compiler encountered an **#endif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.
- MAKE error*    **Misplaced endif statement**  
There's an **endif** directive without any matching **!if** directive.
- TLINK fatal error*    **filename (linenum): Missing internal name**  
In the IMPORTS section of the module definition file there was a reference to an entry specified via module name and ordinal number. When an entry is specified by ordinal number an internal name must be assigned to this import definition. It is this internal name that your program uses to refer to the imported definition. The syntax in the module definition file should be:
- ```
<internalname>=<modulename>.<ordinal>
```
- Compile-time warning* **Mixing pointers to signed and unsigned char**
You converted a **signed char** pointer to an **unsigned char** pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)
- Compile-time error* **Multiple base classes require explicit class names**
In a C++ class constructor, each base class constructor call in the constructor header must include the base class name when there is more than one immediate base class.
- Compile-time error* **Multiple declaration for identifier**
This identifier was improperly declared more than once. This might be caused by conflicting declarations such as `int a;` `double a;`, a function declared two different ways, or a label repeated in the same function, or some declaration repeated other than an **extern** function or a simple variable (in C).
- Compile-time error* **identifier must be a member function**
Most C++ operator functions may be members of classes or ordinary nonmember functions, but certain ones are required to be members of classes. These are **operator =**, **operator ->**, **operator ()**, and type conversions. This operator function is not a member function but should be.
- Compile-time error* **identifier must be a member function or have a parameter of class type**
Most C++ operator functions must have an implicit or explicit parameter of class type. This operator function was declared

outside a class and does not have an explicit parameter of class type.

- Compile-time error* **identifier must be a previously defined class or struct**
You are attempting to declare *identifier* to be a base class, but either it is not a class or it has not yet been fully defined. Correct the name or rearrange the declarations.
- Compile-time error* **identifier must be a previously defined enumeration tag**
This declaration is attempting to reference *identifier* as the tag of an **enum** type, but it has not been so declared. Correct the name, or rearrange the declarations.
- Compile-time error* **function must be declared with no parameters**
This C++ operator function was incorrectly declared with parameters.
- Compile-time error* **function must be declared with one parameter**
This C++ operator function was incorrectly declared with more than one parameter.
- Compile-time error* **operator must be declared with one or no parameters**
When **operator++** or **operator --** is declared as a member function, it must be declared to take either no parameters (for the prefix version of the operator) or one parameter of type **int** (for the postfix version).
- Compile-time error* **operator must be declared with one or two parameters**
When **operator++** or **operator --** is declared as a nonmember function, it must be declared to take either one parameter (for the prefix version of the operator) or two parameters (the postfix version).
- Compile-time error* **function must be declared with two parameters**
This C++ operator function was incorrectly declared with other than two parameters.
- Compile-time error* **Must take address of a memory location**
Your source file used the address-of operator (**&**) with an expression which cannot be used that way; for example, a register variable (in C).
- Compile-time error* **Need an identifier to declare**
In this context, an identifier was expected to complete the declaration. This might be a **typedef** with no name, or an extra semicolon at file level. In C++, it might be a class name improperly used as another kind of identifier.

- IDE debugger error* **'new' and 'delete' not supported**
In integrated debugger expression evaluation, the **new** and **delete** operators are not supported.
- Compile-time error* **No : following the ?**
The question mark (?) and colon (:) operators do not match in this expression. The colon may have been omitted, or parentheses may be improperly nested or missing.
- Compile-time error* **No base class to initialize**
This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes. Check your declarations.
- MAKE error* **No closing quote**
There is no closing quote for a string expression in a **!if** or **!elif** expression.
- Compile-time warning* **No declaration for function *function***
You called a function without first declaring that function. In C, you can declare a function without presenting a prototype, as in "int func();". In C++, every function declaration is also a prototype; this example is equivalent to "int func(void);". The declaration can be either classic or modern (prototype) style.
- Compile-time error* **No file name ending**
The file name in an **#include** statement was missing the correct closing quote or angle bracket.
- MAKE error* **No file name ending**
The file name in an **!include** statement is missing the correct closing quote or angle bracket.
- Compile-time error* **No file names given**
The command line of the Turbo C++ command-line compiler (BCC) contained no file names. You have to specify a source file name.
- MAKE error* **No macro before =**
You must give a macro a name before you can assign it a value.
- MAKE error* **No match found for wildcard *expression***
There are no files matching the wildcard *expression* for MAKE to expand. For example, if you write

```
prog.exe: *.obj
```

MAKE sends this error message if there are no files with the extension **.OBJ** in the current directory.

TLINK warning **No program starting address defined**
This warning means that no module defined the initial starting address of the program. This is almost certainly caused by forgetting to link in the initialization module C0x.OBJ.

TLINK warning **No stack**
This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Turbo C++, or for any application program that will be converted to a .COM file. For other programs (except DLLs), this indicates an error.

If a Turbo C++ program produces this message for any but the tiny memory model, make sure you are using the correct C0x startup object files.

TLINK warning **No stub for fixup at segment:xxxxh in module module**
This error occurs when the target for a fixup is in an overlay segment, but no stub is found for a target external. This is usually the result of not making public a symbol in an overlay that is referenced from the same module.

MAKE fatal error **No terminator specified for in-line file operator**
The makefile contains either the && or << command-line operators to start an in-line file, but the file is not terminated.

Compile-time error **No type information**
This message used only by IDE debugger. Debugger has no type information for this variable. Module may have been compiled without debug switch turned on, or by another compiler or assembler.

Compile-time warning **Non-const function *function* called for const object**
A non-**const** member function was called for a **const** object. This is an error, but was reduced to a warning to give existing programs a chance to work.

Compile-time warning **Nonportable pointer comparison**
Your source file compared a pointer to a non-pointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.

Compile-time error **Nonportable pointer conversion**
An implicit conversion between a pointer and an integral type is required, but the types are not the same size. This cannot be done without an explicit cast. This conversion may not make any sense, so be sure this is what you want to do.

N

- Compile-time warning* **Nonportable pointer conversion**
 A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same. Use an explicit cast if this is what you really meant to do.
- Compile-time error* **Nontype template argument must be of scalar type**
 A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.
- Compile-time error* **Non-virtual function *function* declared pure**
 Only virtual functions can be declared pure, since derived classes must be able to override them.
- Compile-time warning* **Non-volatile function *function* called for volatile object**
 In C++, a class member function was called for a volatile object of the class type, but the function was not declared with "volatile" following the function header. Only a volatile member function may be called for a volatile object.
- Compile-time error* **Not a valid expression format type**
This message used only by IDE debugger.
 Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).
- Compile-time error* **Not an allowed type**
 Your source file declared some sort of forbidden type; for example, a function returning a function or array.
- MAKE fatal error* **Not enough memory**
 All your working storage has been exhausted.
- TLINK fatal error* **Not enough memory**
 There is not enough memory to run TLINK. Try reducing the size of any RAM disk or disk cache currently active. Then run TLINK again. If you are running real mode, try using the MAKE **-S** option, removing TSRs and network drivers. If you are using protected mode MAKE, try reducing the size of any ram disk or disk cache you may have active.
- TLIB error* **Not enough memory for command-line buffer**
 This error occurs when TLIB runs out of memory.
- DPMI server fatal error* **not enough memory for PM init**
 There was not enough extended memory available for the DPMI server to initialize protected mode.

TLIB warning **module not found in library**

An attempt to perform either a `'_'` or `'*'` on a library has been made and the indicated object does not exist in the library.

Run-time error **Null pointer assignment**

When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes within the program's data segment have changed. These bytes would never be altered by a working program. If they have been changed, the message "Null pointer assignment" is displayed to inform you that (most likely) a value was stored to an uninitialized pointer. The program may appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including "locking" the computer up in the large, compact, and huge memory models). You can use the integrated debugger to track down null pointers.

Compile-time error **Numeric constant too large**

String and character escape sequences larger than hexadecimal `\xFF` or octal `\377` cannot be generated. Two-byte character constants may be specified by using a second backslash. For example, `\x0D\x0A` represents a two-byte constant. A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0D" "12345");
```

This prints a carriage return followed by 12345.

TLIB error **object module filename is invalid**

The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.

Compile-time error **Objects of type type cannot be initialized with {}**

Ordinary C structures can be initialized with a set of values inside braces. C++ classes can only be initialized with constructors if the class has constructors, private members, functions or base classes which are virtual.

MAKE error **Only `<<KEEP` or `<<NOKEEP`**

You have specified something besides `KEEP` or `NOKEEP` when closing a temporary inline file.

- Compile-time error* **Only member functions may be 'const' or 'volatile'**
Something other than a class member function has been declared **const** and/or **volatile**.
- Compile-time error* **Only one of a set of overloaded functions can be "C"**
C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you wish to override the compiler's assigning a new name by declaring the function extern "C", you can do this for only one of a set of functions with the same name. (Otherwise the linker would find more than one global function with the same name.)
- Compile-time error* **Operand of delete must be non-const pointer**
It is illegal to delete a constant pointer value using operator **delete**.
- Compile-time error* **Operator [] missing]**
The C++ **operator[]** was declared as **operator [**. You must add the missing **]** or otherwise fix the declaration.
- Compile-time error* **operator -> must return a pointer or a class**
The C++ **operator->** function must be declared to either return a class or a pointer to a **class** (or **struct** or **union**). In either case, it must be something to which the **->** operator can be applied.
- Compile-time error* **operator delete must return void**
This C++ overloaded operator **delete** was declared in some other way.
- Compile-time error* **Operator must be declared as function**
An overloaded operator was declared with something other than function type.
- Compile-time error* **operator new must have an initial parameter of type size_t**
Operator **new** can be declared with an arbitrary number of parameters, but it must always have at least one, which is the amount of space to allocate.
- Compile-time error* **operator new must return an object of type void ***
The C++ overloaded operator **new** was declared another way.
- Compile-time error* **Operators may not have default argument values**
It is illegal for overloaded operators to have default argument values.

Compile-time fatal error

Out of memory

The total working storage is exhausted. Compile the file on a machine with more memory.

TLIB error

Out of memory

For any number of reasons, TLIB or Turbo C++ ran out of memory while building the library. For many specific cases a more detailed message is reported, leaving "Out of memory" to be the basic catchall for general low memory situations.

If this message occurs when public symbol tables grow too large, you must free up memory. For the command line this could involve removing TSR's or device drivers using real mode memory. In the IDE, some additional memory can be gained by closing editors.

TLINK fatal error

Out of memory

TLINK has run out of dynamically allocated memory needed during the link process. This error is a catchall for running into a TLINK limit on memory usage. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together. You can try reducing the size of RAM disks and/or disk caches that may be active.

TLIB error

out of memory creating extended dictionary

The librarian has run out of memory creating an extended dictionary for a library. The library is created but will not have an extended dictionary.

TLIB error

out of memory reading LE/LIDATA record from object module

The librarian is attempting to read a record of data from the object module, but it cannot get a large enough block of memory. If the module that is being added has a large data segment or segments, it is possible that adding the module before any other modules might resolve the problem. By adding the module first, there will be memory available for holding public symbol and module lists later.

TLIB error

Out of space allocating per module debug struct

The librarian ran out of memory while allocating space for the debug information associated with a particular object module. Removing debugging information from some modules being added to the library might resolve the problem.

- TLIB error* **Output device is full**
The output device is full, usually no space left on the disk.
- TLINK warning* **Overlays generated and no overlay manager included**
This warning is issued if overlays are created but the symbol `__OVRTRAP__` is not defined in any of the object modules or libraries linked in. The standard overlay library (`OVERLAY.LIB`) defines this symbol.
- Compile-time error* **Overlays only supported in medium, large, and huge memory models**
As explained in Chapter 18, "Memory management", only programs using the medium, large, or huge memory models may be overlaid.
- Compile-time warning* **overload is now unnecessary and obsolete**
Early versions of C++ required the reserved word **overload** to mark overloaded function names. C++ now uses a "type-safe linkage" scheme, whereby all functions are assumed overloaded unless marked otherwise. The use of **overload** should be discontinued.
- Compile-time error* **Overloadable operator expected**
Almost all C++ operators can be overloaded. The only ones that can't be overloaded are the field-selection dot (`.`), dot-star (`.*`), double colon (`::`), and conditional expression (`?:`). The preprocessor operators (`#` and `##`) are not C or C++ language operators and thus cannot be overloaded. Other nonoperator punctuation, such as semicolon (`;`), of course, cannot be overloaded.
- Compile-time error* **Overloaded function name ambiguous in this context**
The only time an overloaded function name can be used without actually calling the function is when a variable or parameter of an appropriate type is initialized or assigned. In this case an overloaded function name has been used in some other context.
- Compile-time error* **Overloaded function resolution not supported**
This message used only by IDE debugger.
In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.
- Compile-time warning* **Overloaded prefix 'operator operator' used as a postfix operator**
With the latest specification of C++, it is now possible to overload both the prefix and postfix versions of the `++` and `--`

operators. To allow older code to compile, whenever only the prefix operator is overloaded, but is used in a postfix context, Turbo C++ uses the prefix operator and issues this warning.

Compile-time error

Parameter names are used only with a function body

When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype. A list of parameter names only is not allowed.

Example declarations include:

```
int func();           // declaration without prototype--OK
int func(int, int);  // declaration with prototype--OK
int func(int i, int j); // parameter names in prototype--OK
int func(i, j);      // parameter names only--illegal
```

Compile-time error

Parameter *number* missing name

In a function definition header, this parameter consisted only of a type specifier *number* with no parameter name. This is not legal in C. (It is allowed in C++, but there's no way to refer to the parameter in the function.)

Compile-time warning

Parameter *parameter* is never used

The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

TLIB error

***path* – path is too long**

This error occurs when the length of any of the library file or module file's *path* is greater than 64.

Compile-time error

Pointer to structure required on left side of \rightarrow or \rightarrow^*

Nothing but a pointer is allowed on the left side of the arrow (\rightarrow) in C or C++. In C++ a \rightarrow^* operator is allowed.

Compile-time warning

Possible use of *identifier* before definition

Your source file used the named variable in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the program uses it.

Compile-time warning **Possibly incorrect assignment**

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (that is, part of an **if**, **while** or **do-while** statement). More often than not, this is a typographical error for the equality operator. If you wish to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. Thus,

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```

TLINK error **Program entry point may not reside in an overlay**

Although almost all of an application can be overlaid, the initial starting address cannot reside in an overlay. This error usually means that an attempt was made to overlay the initialization module C0x.OBJ, for instance, by specifying the **b** option before the startup module.

TLIB error **public symbol in module *module1* clashes with prior module *module2***

A public symbol may only appear once in a library file. A module which is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added. The command-line message reports the *module2* name.

TLIB error **public symbol in module *filename* clashes with prior module**

A public symbol may only appear once in a library file. A module which is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added.

TLIB error **record kind *num* found, expected *theadr* or *lheadr* in module *filename***

The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.

TLIB error **record length *len* exceeds available buffer in module *module***

This error occurs when the record length *len* exceeds the available buffer to load the buffer in module *module*. This occurs when TLIB runs out of dynamic memory.

- TLIB error* **record type *type* found, expected theadr or leadr in *module***
TLIB encountered an unexpected type *type* instead of the expected THEADR or LHEADER record in module *module*.
- Compile-time warning* **Redefinition of *macro* is not identical**
Your source file redefined the named macro using text that was not exactly the same as the first definition of the macro. The new text replaces the old.
- MAKE error* **Redefinition of target *filename***
The named file occurs on the left side of more than one explicit rule.
- Compile-time error* **Reference initialized with *type1*, needs lvalue of type *type2***
A reference variable or parameter that is not declared constant must be initialized with an lvalue of the appropriate type. In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.
- Compile-time error* **Reference member *member* in class without constructors**
A class that contains reference members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.
- Compile-time error* **Reference member *member* is not initialized**
References must always be initialized. A class member of reference type must have an initializer provided in all constructors for that class. This means that you cannot depend on the compiler to generate constructors for such a class, since it has no way of knowing how to initialize the references.
- Compile-time error* **Reference member *member* needs a temporary for initialization**
You provided an initial value for a reference type which was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Since there is no obvious place to store this temporary, the initialization is illegal.
- Compile-time error* **Reference variable *variable* must be initialized**
This C++ object is declared as a reference but is not initialized. All references must be initialized at the point of their declaration.
- Compile-time fatal error* **Register allocation failure**
This is a sign of some form of compiler error. Some expression in the indicated function was so complicated that the code generator could not generate code for it. Try to simplify the

offending function. Notify Borland Technical Support if the compiler encounters this error.

TLINK fatal error

Relocation item exceeds 1MB DOS limit

The DOS executable file format doesn't support relocation items for locations exceeding 1MB. Although DOS could never *load* an image this big, DOS extenders can, and thus TLINK supports generating images greater than DOS could load. Even if the image is loaded with a DOS extender, the DOS executable file format is limited to describing relocation items in the first 1MB of the image.

TLINK fatal error

Relocation offset overflow

This error only occurs for 32-bit object modules and indicates a relocation (segment fixup) offset greater than the DOS limit of 64K.

TLINK fatal error

Relocation table overflow

This error only occurs for 32-bit object modules. The file being linked contains more base fixups than the standard DOS relocation table can hold (base fixups are created mostly by calls to far functions).

Compile-time error

*This message used only by
IDE debugger.*

Repeat count needs an lvalue

The expression before the comma (,) in the Watch or Evaluate window must be a manipulable region of storage. For example, expressions like this one are not valid:

```
i++,10d  
x = y, 10m
```

TLIB warning

results are safe in file *filename*

The librarian has successfully built the library into a temporary file, but cannot rename the file to the desired library name. The temporary file will not be removed (so that the library can be preserved).

MAKE error

Rule line too long

An implicit or explicit rule was longer than 4,096 characters.

TLINK fatal error

Segment *segment* exceeds 64K

This message occurs if too much data is defined for a given data or code segment when TLINK combines segments with the same name from different source files.

TLINK warning

Segment *segment* is in two groups: *group1* and *group2*

The linker found conflicting claims by the two named groups. Usually, this only happens in assembly language programs. It means that two modules assigned the segment to two different groups.

Compile-time error

Side effects are not allowed

This message used only by IDE debugger.

Side effects such as assignments, ++, or -- are not allowed in the debugger watch window. A common error is to use $x = y$ (not allowed) instead of $x == y$ to test the equality of x and y .

Compile-time error

Size of *identifier* is unknown or zero

This identifier was used in a context where its size was needed. A **struct** tag may only be declared (the **struct** not defined yet), or an **extern** array may be declared without a size. It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declaration so that the size of *identifier* is available.

Compile-time error

sizeof may not be applied to a bit field

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

Compile-time error

sizeof may not be applied to a function

sizeof may be applied only to data objects, not functions. You may request the size of a pointer to a function.

Compile-time error

Size of the type is unknown or zero

This type was used in a context where its size was needed. For example, a **struct** tag may only be declared (the **struct** not defined yet). It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declarations so that the size of this type is available.

Compile-time error

***identifier* specifies multiple or duplicate access**

A base class may be declared **public** or **private**, but not both. This access specifier may appear no more than once for a base class.

Run-time error

Stack overflow

The default stack size for Turbo C++ programs is 5120 bytes. This should be enough for most programs, but those which execute recursive functions or store a great deal of local data can overflow the stack. You will only get this message if you have stack checking enabled. If you do get this message, you can try increasing the stack size or decreasing your program's dependence on the stack. Change the stack size by altering the

global variable `_stklen`. Try switching to a larger memory model to fit the larger stack.

To decrease the amount of local data used by a function, look at the example below. The variable `buffer` has been declared static and does not consume stack space like `list` does.

```
void anyfunction(void)
{
    static int buffer[2000]; /* resides in the data segment */
    int list[2000];         /* resides on the stack */
}
```

There are two disadvantages to declaring local variables as static.

1. It now takes permanent space away from global variables and the heap. (You have to rob Peter to pay Paul.) This is usually only a minor disadvantage.
2. The function may no longer be reentrant. What this means is that if the function is called recursively or asynchronously and it is important that each call to the function have its own unique copy of the variable, you cannot make it static. This is because every time the function is called, it will use the same exact memory space for the variable, rather than allocating new space for it on each call. You could have a sharing problem if the function is trying to execute from within itself (recursively) or at the same time as itself (asynchronously). For most DOS programs this is not a problem.

Compile-time error

Statement missing ;

The compiler encountered an expression statement without a semicolon following it.

Compile-time error

Storage class *storage class* is not allowed here

The given storage class is not allowed here. Probably two storage classes were specified, and only one may be given.

MAKE error

String type not allowed with this operand

You have tried to use an operand which is not allowed for comparing string types. Valid operands are `==`, `!=`, `<`, `>`, `<=`, and `>=`.

Compile-time warning

Structure passed by value

A structure was passed by value as an argument to a function without a prototype. It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.

Compile-time error

Structure required on left side of . or .*

The left side of a dot (.) operator (or C++ dot-star operator) must evaluate to a structure type. In this case it did not.

Compile-time error

Structure size too large

Your source file declared a structure larger than 64K.

Compile-time warning

Style of function definition is now obsolete

In C++, this old C style of function definition is illegal:

```
int func(p1, p2)
int p1, p2;
{
:
}
```

This practice may not be allowed by other C++ compilers.

Compile-time error

Subscripting missing]

The compiler encountered a subscripting expression which was missing its closing bracket. This could be caused by a missing or extra operator, or mismatched parentheses.

Compile-time warning

Superfluous & with function

An address-of operator (&) is not needed with function name; any such operators are discarded.

Compile-time warning

Suspicious pointer conversion

The compiler encountered some conversion of a pointer which caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.

Compile-time error

Switch selection expression must be of integral type

The selection expression in parentheses in a **switch** statement must evaluate to an integral type (**char, short, int, long, enum**). You may be able to use an explicit cast to satisfy this requirement.

S

- Compile-time error* **Switch statement missing (**
In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.
- Compile-time error* **Switch statement missing)**
In a **switch** statement, the compiler found no right parenthesis after the test expression.
- TLINK fatal error* **filename (linenum): Syntax error**
TLINK found a syntax error in the module definition file. The filename and line number tell you where the syntax error occurred.
- TLINK fatal error* **Table limit exceeded**
One of linker's internal tables overflowed. This usually means that the programs being linked have exceeded the linker's capacity for public symbols, external symbols, or for logical segment definitions. Each instance of a distinct segment name in an object file counts as a logical segment; if two object files define this segment, then this results in two logical segments.
- Compile-time error* **Template argument must be a constant expression**
A non-type actual template class argument must be a constant expression (of the appropriate type); this includes constant integral expressions, and addresses of objects or functions with external linkage or members.
- Compile-time error* **Template class nesting too deep: 'class'**
The compiler imposes a certain limit on the level of template class nesting; this limit is usually only exceeded through a recursive template class dependency. When this nesting limit is exceeded, the compiler will issue this error message for all of the nested template classes, which usually makes it easy to spot the recursion. This is always followed by the fatal error **Out of memory**.

For example, consider the following set of template classes:

```
template<class T> class A
{
    friend class B<T*>;
};

template<class T> class B
{
    friend class A<T>;
};

A<int> x;
```

This snippet will be flagged with the following errors:

```
Error: Template class nesting too deep: 'B<int * * * * * >'  
Error: Template class nesting too deep: 'A<int * * * * * >'  
Error: Template class nesting too deep: 'B<int * * * * * >'  
Error: Template class nesting too deep: 'A<int * * * * * >'  
Error: Template class nesting too deep: 'B<int * * * * * >'  
Error: Template class nesting too deep: 'A<int * * * * * >'  
Error: Template class nesting too deep: 'B<int * * * * * >'  
Error: Template class nesting too deep: 'A<int * * * * * >'  
Error: Template class nesting too deep: 'B<int * * * * * >'  
Error: Template class nesting too deep: 'A<int * * * * * >'  
Fatal: Out of memory
```

Compile-time error **Template function argument *argument* not used in argument types**

The given argument was not used in the argument list of the function. The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.

Compile-time error **Template functions may only have type-arguments**

A function template was declared with a non-type argument. This is not allowed with a template function, as there is no way to specify the value when calling it.

Compile-time error **Templates can only be declared at file level**

Templates cannot be declared inside classes or functions, they are only allowed in the global scope (file level).

Compile-time error **Templates must be classes or functions**

The declaration in a template declaration must specify either a class type or a function.

Compile-time warnings **Temporary used to initialize *identifier***

Temporary used for parameter *number* in call to *function*

Temporary used for parameter *parameter* in call to *function*

Temporary used for parameter *number*

Temporary used for parameter *parameter*

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

For example, here function **f** requires a reference to an **int**, and **c** is a **char**:

```
f(int&);  
char c;  
f(c);
```

Instead of calling **f** with the address of **c**, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

TLINK fatal error **Terminated by user**

You canceled the link.

TLIB error **The combinations '+*' or '*+' are not allowed**

It is not legal to add and extract an object module from a library in one action. The action probably desired is a '+-'.

Compile-time error **The constructor *constructor* is not allowed**

Constructors of the form **X::(X)** are not allowed. The correct way to write a copy constructor is **X::(const X&)**.

Compile-time error **The value for *identifier* is not within the range of an int**

All enumerators must have values which can be represented as an integer. You attempted to assign a value which is out of the range of an integer. In C++ if you need a constant of this value, use a **const** integer.

Compile-time error **'this' can only be used within a member function**

In C++, **this** is a reserved word that can be used only within class member functions.

Compile-time warning **This initialization is only partly bracketed**

Result of IDE Options | Compiler | Messages | ANSI violations selection. Initialization is only partially bracketed. When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces may be used. This ensures that your idea and the compiler's idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

- Compile-time error* **Too few arguments in template class name *template***
A template class name was missing actual values for some of its formal parameters.
- Compile-time error* **Too few parameters in call**
A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.
- Compile-time error* **Too few parameters in call to *function***
A call to the named function (declared using a prototype) had too few arguments.
- Compile-time error* **Too many decimal points**
The compiler encountered a floating-point constant with more than one decimal point.
- Compile-time error* **Too many default cases**
The compiler encountered more than one **default** statement in a single **switch**.
- Compile-time error* **Too many error or warning messages**
A maximum of 255 errors and warnings can be set before the compiler stops.
- TLINK error* **Too many error or warning messages**
The number of messages reported by the compiler has exceeded its limit. This error indicates that TLINK reached its limit.
- Compile-time error* **Too many exponents**
The compiler encountered more than one exponent in a floating-point constant.
- Compile-time error* **Too many initializers**
The compiler encountered more initializers than were allowed by the declaration being initialized.
- Compile-time error* **Too many storage classes in declaration**
A declaration may never have more than one storage class.
- MAKE error* **Too many suffixes in *.SUFFIXES* list**
You have exceeded the 255 allowable suffixes in the suffixes list.
- Compile-time error* **Too many types in declaration**
A declaration may never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum**, or **typedef-*name***.

- Compile-time error* **Too much global data defined in file**
 The sum of the global data declarations exceeds 64K bytes. Check the declarations for any array that may be too large. Also consider reorganizing the program or using **far** variables if all the declarations are needed.
- Compile-time error* **Trying to derive a far class from the huge base base**
 If a class is declared (or defaults to) **huge**, all derived classes must also be **huge**.
- Compile-time error* **Trying to derive a far class from the near base base**
 If a class is declared (or defaults to) **near**, all derived classes must also be **near**.
- Compile-time error* **Trying to derive a huge class from the far base base**
 If a class is declared (or defaults to) **far**, all derived classes must also be **far**.
- Compile-time error* **Trying to derive a huge class from the near base base**
 If a class is declared (or defaults to) **near**, all derived classes must also be **near**.
- Compile-time error* **Trying to derive a near class from the far base base**
 If a class is declared (or defaults to) **far**, all derived classes must also be **far**.
- Compile-time error* **Trying to derive a near class from the huge base base**
 If a class is declared (or defaults to) **huge**, all derived classes must also be **hugh**.
- Compile-time error* **Two consecutive dots**
 Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), there is no way two consecutive dots can legally occur in a C program.
- Compile-time error* **Two operands must evaluate to the same type**
 The types of the expressions on both sides of the colon in the conditional expression operator (**?:**) must be the same, except for the usual conversions like **char** to **int** or **float** to **double**, or **void*** to a particular pointer. In this expression, the two sides evaluate to different types that are not automatically converted. This may be an error or you may merely need to cast one side to the type of the other.

Type mismatch family



When compiling C++ programs, the following messages that refer to this note are always preceded by another message that explains the exact reason for the type mismatch; this is usually

“Cannot convert ‘type1’ to ‘type2’”, but the mismatch may be due to many other reasons.

Compile-time error

Type mismatch in default argument value

Type mismatch in default value for parameter *parameter*

The default parameter value given could not be converted to the type of the parameter. The first message is used when the parameter was not given a name.

Compile-time error

Type mismatch in parameter *number*

The function called, via a function pointer, was declared with a prototype; the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *number* in call to function

Your source file declared the named function with a prototype, and the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *parameter*

Your source file declared the function called via a function pointer with a prototype, and the named parameter could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *parameter* in call to function

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type. See entry for **Type mismatch in parameter *parameter***.

Compile-time error

Type mismatch in parameter *parameter* in template class name *template*

Type mismatch in parameter *number* in template class name *template*

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in redeclaration of *identifier*

Your source file redeclared with a different type than was originally declared. This can occur if a function is called and subsequently declared to return something other than an

T

integer. If this has happened, you must declare the function before the first call to it.

Compile-time error

Type name expected

One of these errors has occurred:

- In declaring a file-level variable or a **struct** field, neither a type name nor a storage class was given.
- In declaring a **typedef**, no type for the name was supplied.
- In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
- In supplying a C++ base class name, the name was not the name of a class.

Compile-time error

Type qualifier *identifier* must be a struct or class name

The C++ qualifier in the construction *qual::identifier* is not the name of a **struct** or **class**.

Compile-time fatal error

Unable to create output file *filename*

The work disk is full or write-protected or the output directory does not exist. If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write-protected, move the source files to a writable disk and restart the compilation.

Compile-time error

Unable to create *turboc.\$ln*

The compiler cannot create the temporary file TURBOC.\$LN because it cannot access the disk or the disk is full.

MAKE fatal error

Unable to execute command

A command failed to execute; this may be because the command file could not be found, it was misspelled, there was no disk space left in the specified swap directory, swap directory does not exist, or (less likely) because the command itself exists but has been corrupted.

Compile-time error

Unable to execute command *command*

TLINK or TASM cannot be found, or possibly the disk is bad.

TLINK fatal error and TLIB error

Unable to open file *filename*

unable to open *filename*

This occurs if the named file does not exist or is misspelled.

TLIB error

unable to open *filename* for output

TLIB cannot open the specified file for output. This is usually due to lack of disk space for the target library, or a listing file. Additionally this error will occur if the target file exists but is marked as a read only file.

Compile-time error **Unable to open include file *filename***
The compiler could not find the named file. This could also be caused if an **#include** file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try FILES=20). Check whether the named file exists.

MAKE error **Unable to open include file *filename***
The compiler could not find the named file. This could also be caused if an **!include** file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try FILES=20). Check whether the named file exists.

Compile-time error **Unable to open input file *filename***
This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is on the proper disk or directory.

Command line fatal error **unable to open 'dpmimem.dll'**
Make sure that DPMIMEM.DLL is somewhere on your path or in the same directory as the protected mode command line tool you were attempting to use.

MAKE fatal error **Unable to open makefile**
The current directory does not contain a file named MAKEFILE, MAKEFILE.MAK, or does not contain the file you specified with **-f**.

MAKE fatal error **Unable to redirect input or output**
MAKE was unable to open the temporary files necessary to redirect input or output. If you are on a network, make sure you have rights to the current directory.

TLIB error **unable to rename *filename* to *filename***
TLIB builds a library into a temporary file and then renames the temporary file to the target library file name. If there is an error, usually due to lack of disk space, this message will be posted.

Compile-time error **Undefined label *identifier***
The named label has a **goto** in the function, but no label definition.

Compile-time warning **Undefined structure *identifier***
The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

- Compile-time error* **Undefined structure *structure***
Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.
- Compile-time error* **Undefined symbol *identifier***
The named identifier has no declaration. This could be caused by a misspelling either at this point or at the declaration. This could also be caused if there was an error in the declaration of the identifier.
- TLINK error* **Undefined symbol *symbol* in module *module***
The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check to make sure the symbol is spelled correctly.

You will usually see this error from TLINK for Turbo C++ symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs. If you are linking C++ code with C modules, you might have forgotten to wrap C external declarations in `extern "C" {...}`. You might have a case mismatch between two symbols. See the /C and /c switches.
- Compile-time error* **Unexpected }**
An extra right brace was encountered where none was expected. Check for a missing {.
- TLIB error* **Unexpected char *X* in command line**
TLIB encountered a syntactical error while parsing the command line.
- MAKE error* **Unexpected end of file**
The end of the makefile was reached without a temporary inline file having been closed.
- Compile-time error* **Unexpected end of file in comment started on *line number***
The source file ended in the middle of a comment. This is normally caused by a missing close of comment (`*/`).

- MAKE or compile-time error* **Unexpected end of file in conditional started on line *line number***
 The source file ended before the compiler (or MAKE) encountered an **endif**. The **endif** was either missing or misspelled.
- Compile-time error* **union cannot be a base type**
 A union cannot be used as a base type for another class type.
- Compile-time error* **union cannot have a base type**
 A union cannot be derived from any other class.
- Compile-time error* **Union member *member* is of type class with constructor**
Union member *member* is of type class with destructor
Union member *member* is of type class with operator=
 A union may not contain members that are of type **class** with user-defined constructors, destructors, or operator=.
- Compile-time error* **unions cannot have virtual member functions**
 A union may not have virtual functions as its members.
- Compile-time warning* **Unknown assembler instruction**
 The compiler encountered an inline assembly statement with a disallowed opcode. Check the spelling of the opcode. This warning is off by default.
- See *BASM.DOC* for more on
 opcode spelling.
- TLIB warning* **unknown command line switch *X* ignored**
 A forward slash character (/) was encountered on the command line or in a response file without being followed by one of the allowed options.
- Compile-time error* **Unknown language, must be C or C++**
 In the C++ construction
- ```
extern "name" type func(/*...*/);
```
- The name given in quotes must be "C" or "C++"; other language names are not recognized. For example, you can declare an external Pascal function without the compiler's renaming like this:
- ```
extern "C" int pascal func( /*...*/ );
```
- A C++ (possibly overloaded) function may be declared Pascal and allow the usual compiler renaming (to allow overloading) like this:
- ```
extern int pascal func(/*...*/);
```



- TLINK fatal error*    **Unknown option**  
A forward slash character (/), hyphen (-), or DOS switch character was encountered on the command line or in a response file without being followed by one of the allowed options. This might mean that you used the wrong case to specify an option.
- Compile-time error*    **Unknown preprocessor directive: *identifier***  
The compiler encountered a # character at the beginning of a line, and the name following was not a legal directive name, or the rest of the directive was not well-formed.
- MAKE error*    **Unknown preprocessor statement**  
A ! character was encountered at the beginning of a line, and the statement name following was not **error**, **undef**, **if**, **elif**, **include**, **else**, or **endif**.
- Compile-time warning*    **Unreachable code**  
A **break**, **continue**, **goto** or **return** statement was not followed by a label or the end of a loop or function. The compiler checks **while**, **do** and **for** loops with a constant test condition, and attempts to recognize loops which cannot fall through.
- Compile-time error*    **Unterminated string or character constant**  
The compiler found no terminating quote after the beginning of a string or character constant.
- Compile-time error*    **Use . or -> to call *function***  
You tried to call a member function without giving an object.
- Compile-time error*    **Use . or -> to call *member*, or & to take its address**  
A reference to a non-static class member without an object was encountered. Such a member may not be used without an object, or its address must be taken using the & operator.
- Compile-time error*    **Use :: to take the address of a member function**  
If **f** is a member function of class **c**, you take its address with the syntax **&c::f**. Note the use of the class type name, not the name of an object, and the **::** separating the class name from the function name. (Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)
- TLIB warning*    **use /e with TLINK to obtain debug information from library**  
The library was built with an extended dictionary and also includes debugging information. TLINK will not extract debugging information if it links using an extended dictionary, so in order to obtain debugging information in an executable

from this library, the linker must be told to ignore the extended dictionary using the /e switch. NOTE: The IDE linker does NOT support extended dictionaries therefore no settings need be altered in the IDE.

*MAKE error*    **Use of : and :: dependants for target target**

You have tried to use the target in both single and multiple description blocks (using both the : and :: operators).

Examples:

```
filea: fileb
filea:: filec
```

*Compile-time warning*    **Use qualified name to access nested type type**

In older versions of the C++ specification, typedef and tag names declared inside classes were directly visible in the global scope. With the latest specification of C++, these names must be prefixed with a **class::** qualifier if they are to be used outside of their class' scope. To allow older code to compile, whenever such a name is uniquely defined in one single class, Turbo C++ will allow its usage without **class::** and issues this warning.

*TLINK or compile-time error*    **User break**

You pressed *Ctrl-Break* while compiling or linking in the IDE, aborting the process. (This is not an error, just a confirmation.)

*DPMI server fatal error*    **v86 task without vcpi**

Another application is running, preventing the DPMI server from switching to protected mode. Remove the interfering application, such as a desktop manager or debugger, then reboot.

*Compile-time error*    **Value of type void is not allowed**

A value of type **void** is really not a value at all, and thus may not appear in any context where an actual value is required. Such contexts include the right side of an assignment, an argument of a function, and the controlling expression of an **if**, **for**, or **while** statement.

*Compile-time error*    **Variable variable has been optimized.**

You have tried to inspect, watch, or otherwise access a variable which the optimizer removed. This variable is never assigned a value and has no stack location.

- Compile-time error*    **Variable *identifier* is initialized more than once**  
This variable has more than one initialization. It is legal to declare a file level variable more than once, but it may have only one initialization (even if two are the same).
- Compile-time error*    **'virtual' can only be used with member functions**  
A data member has been declared with the **virtual** specifier; only member functions may be declared **virtual**.
- Compile-time error*    **Virtual function *function1* conflicts with base class *base***  
A virtual function has the same argument types as one in a base class, but a different return type. This is illegal.
- Compile-time error*    **virtual specified more than once**  
The C++ reserved word **virtual** may appear only once in a member function declaration.
- Compile-time error*    **void & is not a valid type**  
A reference always refers to an object, but an object cannot have the type void. Thus the type void is not allowed.
- Compile-time warning*    **Void functions may not return a value**  
Your source file declared the current function as returning **void**, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.
- Compile-time error*    **function was previously declared with the language *language***  
Only one language can be used with **extern** for a given function. This function has been declared with different languages in different locations in the same module.
- Compile-time error*    **While statement missing (**  
In a **while** statement, the compiler found no left parenthesis after the **while** keyword.
- Compile-time error*    **While statement missing )**  
In a **while** statement, the compiler found no right parenthesis after the test expression.
- TLINK fatal error*    **Write failed, disk full?**  
This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.
- Compile-time error*    **Wrong number of arguments in call of macro *macro***  
Your source file called the named macro with an incorrect number of arguments.

- \"
- escape sequence (display double quote) 359
- \'
- escape sequence (display single quote) 359
- \?
- (display question mark) 59
- escape sequence (display question mark) 359
- \\
- (display backslash) 59
- escape sequence (display backslash character) 359
- /\*\*/ (comments) 351
- /\*\*/ (token pasting) 351
- \*\* (all dependents macro) 321
- \", (display double quote) 59
- \', (display single quote) 59
- \$(all out of date dependents macro) 321
- \$(base file name macro) 319
- » (chevron) in dialog boxes 32
- // (comments) 128, 200, 352
- == (equal to operator) 73
- \$. (file name and extension macro) 320
- \$& (file name only macro) 320
- \$(file name path macro) 320
- \$< (full file name macro) 319
- \$@ (full name with path macro) 320
- >= (greater than or equal to operator) 73
- ++ (increment operator) 63
- <= (less than or equal to operator) 73
- && (logical AND operator) 74
- || (logical OR operator) 74
- ? MAKE help option 300
- != (not equal to operator) 73
- operator
- decrement 425, 428
- ? : operator
- conditional expression 426, 438
- :: (scope resolution operator) 136, 138, 158, 205, 426, 452
- 1 TCC (extended 80186 instructions) *See also* 80186 processor, generating extended instructions
- 2 TCC option (80286 instructions) 275
- 1 TCC option (extended 80186 instructions) 275
- /3 TLINK option (32-bit code) 338
- & (ampersand) MAKE command (multiple dependents) 305
- .\* and ->\* operators (dereference pointers) 426, 440
- = (assignment operator) 63
- 32-bit code 338
- 87 environment variable 599
- \ (escape sequence character) 58
- ^ (exclusive OR operator) 64
- ;(for empty loops) 83, 203
- > (greater than operator) 73
- (hyphen) MAKE command (ignore exit status) 305
- :(labeled statement) 367
- < (less than operator) 73
- # (MAKE comment character) 304
- % (modulus) 61
- ! (NOT operator) 74
- != operator
- huge pointer comparison and 578
- not equal to 426, 436
- && operator
- logical AND 425, 437
- MAKE 305, 306
- ++ operator
- increment 425, 427, 428
- << operator
- MAKE 305
- overloading *See* overloaded operators
- put to *See* overloaded operators, << (put to)
- shift bits left 64, 425, 433

- <= operator
  - less than or equal to 426, 434
- == operator
  - equal to 435
  - huge pointer comparison and 578
- >= operator
  - greater than or equal to 426, 434
- >> operator
  - get from *See* overloaded operators, >> (get from)
  - MAKE 305
  - overloading *See* overloaded operators
  - shift bits right 64, 425, 433
- || operator
  - logical OR 425, 437
- > operator (selection) 426
  - overloading 485
  - structure member access 411, 427
  - union member access 427
- | (OR operator) 64
- \* (pointer declarator) 367
- (decrement operator) 63
- ;(Semi-colon)
  - null statement 366
- ;(semi-colon)
  - null statement 442
  - statement terminator 366, 442
- \ (string continuation character) 363
- ## symbol
  - overloading and 480
  - preprocessor directives 424
  - token pasting 351, 507
- ≡ (System) menu 23
- \$ editor macros *See* individual names of macros
- @ MAKE command 305
- ! operator
  - logical negation 425, 430
- % operator
  - modulus 425, 431
  - remainder 425, 431
- & operator
  - address 425, 429
  - AND 64
  - bitwise AND 425, 436
    - truth table 436
  - position in reference declarations 383, 450
- \* operator
  - indirection 425, 429
    - pointers and 402
  - multiplication 425, 431
- + operator
  - addition 425, 432
  - overloading *See* overloaded operators, addition (+)
  - unary plus 425, 430
- , operator
  - evaluation 426, 440
  - function argument lists and 366
- operator
  - subtraction 425, 432
  - unary minus 425, 430
- / operator
  - division 425, 431
  - rounding 432
- < operator
  - less than 426, 434
- = operator
  - assignment 425, 439
    - compound 439
  - overloading 484
  - equal to 426
  - initializer 368
- > operator
  - greater than 426, 434
- ^ operator
  - bitwise XOR 425, 436
    - truth table 436
- | operator
  - bitwise inclusive OR 425, 437
    - truth table 436
- ~ operator
  - 1's complement 64
  - bitwise complement 425, 430
  - destructors 174
- . operator (selection) 426
  - structure member access 411, 427
- 1's complement *See* operators, 1's complement
- 1's complement (~) 425, 430
- # symbol
  - conditional compilation and 510
  - converting strings and 508
  - null directive 503
  - overloading and 480

- preprocessor directives *368, 424, 502*
- # symbol (directives) *46*
- 80x87 coprocessors *See numeric coprocessors*
- 80x86 processors
  - 32-bit code *338*
  - address segment:offset notation *575*
  - instructions *275*
    - extended *275*
    - registers *572-574*
- (arrows) in dialog boxes *31*
- \_OvrInitEms (function) *594*
- \_OvrInitExt (function) *595*

## A

- \a (audible bell) *59*
- a command-line option (word alignment) *413*
- \a escape sequence (audible bell) *359*
- a MAKE option (autodependency check) *300, 312*
- a TCC option (align integers) *275*
- A TCC option (ANSI keywords) *281, 521*
- abort (function)
  - destructors and *478*
- abstract classes *530, See also classes, abstract*
- ACBP field *341*
- access
  - classes *129, 464-466*
    - structures vs. *141*
  - data members and member functions *129, 140, 187, 204, 462*
  - friend classes *465*
  - friend functions *463*
  - functions and variables *113, 138*
  - information hiding and *140*
  - inheritance and *146*
  - overriding *463*
  - structure members *411, 427, 463*
  - structures, classes vs. *141*
  - unions, members *427, 463*
- accounting applications *602*
- action symbols *See TLIB (librarian)*
- activating, menu bar *23*
- active page
  - defined *620*
  - setting *619*
- active window *See windows, active*
- adapters, video *See video adapters*

- Add Item command *253*
- Add Watch command
  - hot key *27*
- addition operator (+) *425, 432*
- address operator (&) *425, 429*
- addresses, memory *See memory addresses*
- adjustfield, ios data member *547*
- aggregate data types *See data types*
- alert (\a) *59, 359*
- algorithms
  - #include directive *509*
- aliases *See referencing and dereferencing*
- alignment
  - attribute *341*
  - integers *275*
  - word *413*
- alloc.h (header file)
  - malloc.h and *567*
- allocate, streambuf member function *556*
- allocation, memory *See memory, allocation*
- American National Standards Institute *See ANSI*
- ampersand (&) MAKE command (multiple dependents) *305*
- ancestors *See classes, base*
- AND operator (&) *64, 425, 436*
  - truth table *436*
- AND operator (&&) *425, 437*
- angle brackets *509*
- anonymous unions
  - member functions and *416*
- ANSI
  - Turbo C++ keywords and *281*
  - C standard *3*
  - compatible code *281*
  - floating point conversion rules *277*
  - keywords *353*
    - option *281*
    - predefined macro *521*
    - violations *282*
- app, ios data member *548*
- argc (argument to main) *523*
- ARGS.EXE *524*
- argsused pragma *515*
- arguments *48, See also parameters*
  - actual
    - calling sequence *408*

- command line, passing to main 523
  - wildcards and 525
- command-line compiler 265
- constructors 139
- conversions 408
- converting to strings 508
- default 139, 180
  - C++ 203
  - constructors and 183, 468, 470
- to #define directive 506
- function calls and 408
- functions and 90
- functions taking none 406
- matching number of 409
- mode 195
- passing, C-language style 392
- passing in C++ 165
- type checking 405
- variable list 278
- variable number of 367
  - Pascal and 394
- Arguments command 250
- argv (argument to main) 523
- arithmetic
  - operations 60
    - combining with assignment operator 63
  - pointers 116
- arithmetic types 384
- arrays 403
  - of classes
    - initializing 454
  - constructors for
    - order of calling 473
  - declaring and initializing 104, 106
  - delete operator and 454
  - elements 105
    - comparing 434
  - evaluating 232
    - number of values 233
  - indeterminate 403
    - structures and 404
  - indexes 119
  - initialization 387, 388
  - inspecting 230
  - multidimensional 106, 403
  - new operator and 173, 453
  - passing to functions 220
  - range errors 106
  - sizeof and 431
  - strings and 108
  - subscripts 365, 426
    - overloading 484
- arrows (→s) in dialog boxes 31
- ASCII codes
  - characters 69
- asm (keyword)
  - how to use 441
- .ASM files *See* assembly language
- asm statement
  - inline pragma and -B TCC option and 517
- aspect ratio
  - determining current 627
  - setting 619
- assembly language
  - assembling from the command line 265
  - built-in assembler *See* The online document
  - UTIL.DOC
  - compiling 285
  - default assembler 286
  - directory 291
  - huge functions and 396, 518
  - inline
    - floating point in 600
    - inline pragma and 517
    - option (-B) 517
  - inline routines 285
  - options 287
  - output files 286
  - projects and 258
  - routines
    - overlays and 593
    - statement syntax 441
- assembly level debugger *See* Turbo Debugger
- assignment operator (=) 425, 439
  - compound 439
  - overloading 484
- assignments
  - assignment operator (=)
    - combining with other operators 63
  - combination 57
  - defined 56
  - multiple 57
- associativity 422, *See also* precedence
- expressions 420

- rules 67
- table 68
- asterisk (\*) 367
- AT TCC option (Turbo C++ keywords) 281
- ate, ios data member 548
- atexit (function)
  - destructors and 477
- attach
  - filebuf member function 544
  - fstreambase member function 546
- attributes
  - ACBP 341
  - alignment 341
  - big 341
  - cell
    - blink 613
    - colors 612
    - combining 341
    - control functions 609
    - screen cells 605, 612
- AU option (UNIX keywords) 281
- audible bell (\a) 59
- auto (keyword) 389
  - class members and 457
  - external declarations and 380
  - register keyword and 374
- auto variables *See* variables, automatic
- .autodepend MAKE directive 323
- autodependencies *See* automatic dependencies
- autoindent mode 631, 633
- automatic dependencies
  - checking 257
    - MAKE (program manager) 300, 312
  - information
    - disabling 279
  - MAKE option 323
- automatic objects 374, *See also* objects, automatic
- automatic variables *See* variables, automatic
- auxiliary carry flag 574
- AX register 573

## B

- \b (backspace character) 59, 359
- b command-line option (enumerations) 417
- B MAKE option (build all) 300

- b TCC option (allocate whole word for enums) 275
- B TCC option (inline assembler code)
  - inline pragma and 517
- B TCC option (process inline assembler code) 285
- background color *See* graphics, colors, background
- backslash character (\) 359
  - hexadecimal and octal numbers and 358
  - line continuation 508
  - printing 59
- backspace character (\b) 59, 359
- Backward compatibility options 292
- bad, ios member function 549
- bad (member function) 195
- banker's rounding 604
- bar *See* run bar
- bar, title 28
- base, streambuf member function 556
- base address register 573
- base classes *See* classes, base
- base file name macro (MAKE) 319
- \_based (keyword) 568
- basefield, ios data member 547
- BASM (built-in assembler) *See* The online document BASM.DOC
- batch files, MAKE 307
- BBS segment *See also* segments
- BCD 602
  - converting 603
  - number of decimal digits 603
  - range 603
  - rounding errors and 603
- beep 59
- bell (\a) 59, 359
- BGI *See* Borland Graphics Interface (BGI)
- BGIOBJ (graphics converter) *See also* The online document UTIL.DOC
  - initgraph function and 617
- BIDS *See* The online document CLASSLIB.DOC
- big attribute 341
- binary, ios data member 548
- binary coded decimal *See* BCD
- binary files
  - opening 123
- binary numbers 50



- binary operators *See* operators
- binding *See* C++, binding
- BIOS
  - video output and 613
- bit fields
  - hardware registers and 415
  - integer 415
  - portable code and 415
  - structures and 414
  - unions and 417
- bit images
  - functions for 619
- bit-mapped fonts *See* fonts
- bitalloc, ios member function 549
- bits
  - blink enable 609
  - color 609
  - manipulating 64
  - shifting 425, 433
- bitwise
  - AND operator (&) 425, 436
    - truth table 436
  - complement operator (~) 425, 430
  - OR operator (|) 425, 437
    - truth table 436
  - XOR operator (^) 425, 436
    - truth table 436
- blen, streambuf member function 556
- blink enable bit 609
- block
  - copy 630, 632
    - Borland-style 633
  - cut 632
  - delete 630, 632
  - extending 630
  - hide and show 630
    - Borland-style 633
  - indent 630
  - move 630, 632
    - Borland-style 633
  - print 630
  - read from disk 630, 632
  - scope 372
  - set beginning of 630
    - Borland-style 633
  - set end of 630
    - Borland-style 633
  - statements 441
  - unindent 630
  - write to disk 630, 632
- block commands 631
- block operations (editor) *See* editing, block operations
- blocks, text *See* editing, block operations
- Boolean data type 134, 443
- Borland
  - contacting 7
- Borland C++ *See also* C++; integrated environment
- Borland Graphics Interface (BGI) *See also* graphics
  - EGA palettes and *See* Enhanced Graphics Adapter (EGA), palette, IDE option (/p)
  - files
    - calling 91
    - using 91
- Turbo C++
  - extensions 353
- Turbo C++
  - keywords
    - as identifiers 281
- boundary conditions 244
- boxes *See* check boxes; dialog boxes; list boxes
- bp, ios data member 547
- BP register 573
  - overlays and 593
- braces 365
- brackets 365, 426
  - overloading 484
- branching *See* if statements; switch statements
- Break Make On
  - Make dialog box 255
- break statements 84, 446
  - loops and 446
- breakpoints *See* watch expressions
- buffers
  - C++ streams and 543, 545
  - file 195
  - overlays
    - default size 592
- bugs 217
- build
  - IDE option (/b) 20
- BUILTINS.MAK 301

## buttons

- choosing 31
- in dialog boxes 31
- radio 31

BX register 573

## C

C++ 199-215, 449-490, *See also* C language;

Turbo C++

arguments 180, 203

passing 165

binding

early vs. late 161

late 132, 160, 214, *See also* member

functions, virtual

example 164

C code and 520

classes *See* classes

comments 128, 200, 352

compiling 142

compiling files as 286

complex numbers *See* complex numbers

constants 202, *See* constants

constructors *See also* constructors

conbuf 541

constream 543

filebuf 543

fstream 545

fstreambase 545, 546

ifstream 537, 546

ios 547

iostream 551

iostream\_withassign 551

istream 551

istream\_withassign 553

istrstream 553

ofstream 537, 553, 554

ofstream\_withassign 555

ostrstream 555

streambuf 556

strstream 560

strstreambase 558

strstreambuf 559

conversions *See* conversions, C++

data members *See* data members

declarations 203, *See also* declarations

#define and 202

destructors *See* destructors

dynamic objects *See* objects

encapsulation 127

defined 126

enumerations *See* enumerations

examples

dictionary 204

file buffers 195

file operations *See* files

fill characters 534

floating-point precision 534

for loops *See* loops, for, C++

formatting *See* formatting

Fourier transforms example 601

functions *See also* member functions

C functions and 376

default arguments for 180

friend 187, 188, 457

access 463

declaring 208, *See* C++, functions

inherited 158

inline 143, 179, 202, *See also* functions

classes and 204

command-line option (-vi) 279

debugging and 279

header files and 144

virtual tables and 287, 289

name mangling and 376

one line 202

overloading *See* overloaded functions

pointers to 399

taking no arguments 406

virtual 214, 485

virtual keyword and 163

graphics classes 133

header files 148, 200

hierarchies *See* classes

I/O 189

flushing cout 201

formatting 193

performing 200

inheritance *See* inheritance

initialization 206

initializers 389

inline functions *See* C++, functions, inline

I/O

disk 193

- formatting 191
- put and write functions and 192
- iterators *See* The online document CLASSLIB.DOC
- keywords 353
- member functions *See* member functions
- members *See also* data members; member functions
  - initialization list 159
- name spaces 413
- objects
  - declaring 135
- operators *See* operators, C++; overloaded operators
- output *See* output, C++
- parameters *See* parameters
- polymorphism *See* polymorphism
- programs
  - compiling 142
  - referencing and dereferencing *See* referencing and dereferencing
- scope *See* scope
- Smalltalk vs. 127
- streams *See* streams, C++
- strings
  - concatenating 184
- structures *See* structures
- templates
  - generating 290
- this
  - nonstatic member functions and 457
  - static member functions and 459
- Turbo C++ implementation 3
- tutorial 199-215
- types
  - reference *See* reference types
- unions *See* unions
- variables
  - declaring anywhere 202
- virtual tables *See* virtual tables
- visibility *See* visibility
- warnings 283
- c TCC option (compile but don't link) 285
- C TCC option (nested comments) 281
- /c TLINK option (case sensitivity) 338
- COFx.OBJ 335

- C language *See also* C++
  - argument passing 392
  - Turbo C++ and 278
  - C++ code and 520
  - calling conventions 519, 521
- COx.OBJ 335
- Call Stack
  - window 240
- Call Stack command
  - hot key 27
- calling conventions *See* parameters, passing; Pascal
- calling sequence, functions 240
- calls
  - far, functions using 396
  - near, functions using 396
- Cancel button 31
- carriage return character 359
- carriage return character (\r) 59
- carry flag 574
- case
  - preserving 394
  - sensitivity
    - forcing 392
    - global variables and 392
    - identifiers and 354
    - pascal identifiers and 355
  - statements *See* switch statements
- case sensitivity 58
  - TLINK and 338
- case statements *See* switch statements
- cast expressions
  - syntax 429
- \_\_CDECL\_\_ macro 519
- cdecl (keyword) 392, 394
  - function modifiers and 396
- \_cdecl (keyword)
  - Microsoft C 569
- cdecl statement 278
- cells, screen *See* screens, cells
- cerr (C++ stream) 189
- cerr, functions of 122
- .CFG files *See* configuration files
- characters
  - ASCII 69
  - blinking 613
  - carriage return (\r) 59

- char data type *See* data types, char
- colors 612, 613
- constants *See* constants, character
- control
  - IDE and 32
- data type char *See* data types, char
- delete 630
- displaying 71
- escape sequence 58
- fill
  - setting 534
- formfeed (\f) 59
- getch function and 70
- getche function and 70
- in screen cells 605
- intensity
  - setting 609
- newline (\n) 59
  - inserting 534
- nonprinting
  - Inspector window and 230
- null
  - defined 71
  - strings and 108
- printable 59
- putch and 71
- reading
  - from keyboard 69
- set of 69
- special, displaying 59
- storage 69
- strings and 72
- tab (\t) 59
- unsigned char data type
  - range 52, 363
- whitespace
  - extracting 534
- charts *See* graphics, charts
- check boxes 32
- chevron symbol (») 32
- .CHR files *See* fonts, files
- cin (C++ stream) 189
  - functions of 122
  - introduced 46
  - using 201
- circles
  - roundness of 619
- CL options
  - command-line compiler options and 563
- class (keyword) 204
- class arguments
  - passing by value 292
- classes 455-468, *See also* C++; individual class
  - names; inheritance; structures
    - abstract 487, 530
    - access 208, 464-466
    - default 464
    - qualified names and 465
    - structures vs. 141
  - arrays of
    - initialization 454
  - auto keyword and 457
  - base 145, 209
    - calling constructor from derived class 475
    - constructors 476
    - defined 131
    - pointers to, destructors and 478
    - private, friend keyword and 465
    - protected keyword and 464
    - unions and 464
    - virtual 466
      - constructors and 473
  - class keyword 204
  - class names and 456
  - constructors 137
    - arguments 139
    - defining 139
    - inline 139
    - naming 139
  - container *See* The online document  
CLASSLIB.DOC
  - data types and 382
  - declarations
    - incomplete 456
  - defined 128
  - derived 145, 209
    - base class access and 464
    - calling base class constructor from 475
    - constructors 476
    - creating 147, 211
    - defined 131
  - deriving 151
  - destructors 138
  - extern keyword and 457

- friend functions and 188, 208
- friends 466-468
  - access 465
- graphics 133
- hierarchies
  - common attributes in 166
  - ios family 531
  - streambuf 530
- initialization *See* initialization, classes
- initializing automatically 137
- inline keyword and 204
- instantiation and 128
- istream, ostream, and iostream 189
- libraries 148
- member functions *See* member functions
- members
  - access 204
  - private, accessing 208
  - members, defined 457
  - naming *See* identifiers
  - objects 455, 457
    - initialization *See* initialization, classes
  - overloaded operators and 190
  - projects and 148
  - register keyword and 457
  - relative position 197
  - scope *See* scope, classes
  - sharing objects 288
  - sizeof operator and 431
  - streambuf 189
  - streams and 529
    - files 529
    - formatted I/O 530
    - memory buffers 529, 530
    - strings 529
  - structures vs. 129
  - syntax 455
  - TLIB and 148
  - unions and 417
- \_clear87 (function)
  - floating point exceptions and 600
- clear (function)
  - C++ stream errors and 195
- clear (function), ios member function 549
- Clear command 632
  - hot key 26
- Clipboard 631
  - copy to 630
  - cut to 630
  - paste from 630, 632
- clipping, defined 621
- clog (C++ stream) 189
  - functions of 122
- close
  - filebuf member function 544
  - fstreambase member function 546
- close boxes 28
- Close command
  - hot key 26
- creol, conbuf member function 541
- clrscr
  - conbuf member function 541
  - constream member function 543
- code-generation
  - command-line compiler options 275
- Code Generation dialog box 413
- code models *See* memory models
- code segment 574
  - group 285
    - naming and renaming 284
    - storing virtual tables in 288
- colons 367
- Color/Graphics Adapter (CGA) *See also*
  - graphics; graphics drivers; video adapters
  - background and foreground colors 624
  - color palettes 623, 624
  - resolution 623
    - high 624
- colors *See* graphics, colors
  - changing IDE text 37
- colors and palettes
  - EGA *See* Enhanced Graphics Adapter (EGA), palette, IDE option (/p)
- columns
  - numbers 28
- .COM files
  - generating 335
    - TLINK 343
  - limitations 343
  - memory models and 578
  - size 343
- combination assignments 57
- combining attribute 341

## COMDEFs

- generating 275, 567
- comma
  - operator 426, 440
  - separator 366
- command line, options *See* command-line
- compiler, options
- command-line compiler 266
  - arguments 265
  - compiling and linking with 265
  - configuration files *See* configuration files
  - directives *See* directives
  - INCLUDE environment variable and 562
  - LIB environment variable and 562
  - MAKE and 323
  - nested comments 352
  - options 267, 272
    - 1 (80186 instructions) 275
    - 2 (80286 instructions) 275
    - A and -AT (Turbo C++ keywords) 281
    - b
      - enumerations 417
    - H (precompiled headers) 286
    - O (jump optimization) 280
    - P (C++ and C compilation) 286
    - X (disable autodependency information) 279
    - Y
      - overlays 279
    - Yo (overlays) 279
    - a (align integers) 275
    - AK (Kernighan and Ritchie keywords) 281
    - alignment (-a) 413
    - allocate whole word for enum (-b) 275
  - ANSI
    - compatible code 281
    - keywords ( $\pm$ A) 521
    - keywords (-A) 281
    - violations 282
  - assembler
    - code 285, 286, 287
    - to use (-E) 286
  - assume DS = SS (-Fs) 276
  - AU (UNIX keywords) 281
  - autodependency information (-X) 279

## -b

- allocate whole word for enums 275
- B (inline assembler code)
  - inline pragma and 517
- B (process inline assembler) 285
- Turbo C++ keywords (-A- and -AT) 281
- C++ and C compilation (-P) 286
- C++ inline functions (-vi) 279
- c (compile and assemble) 285
- C (nested comments) 281
- changing from within programs 517
- CL options versus 563
- code-generation 275
- code segment
  - class name 284
  - group 285
- .COM file names (-tDc) 291
- comments, nesting (-C) 281
- compatibility 565
- compilation control 285
- compile and assemble (-c) 285
- compile C++ (-P) 142
- configuration files and 267
- D (macro definitions) 274
- d (merge literal strings) 275
- data segment
  - class name 285
  - group 284, 285
  - name 284, 285, 584
- debugging information ( $\pm$ v) 278, 340
- define identifiers (-D) 505
- #defines 274
  - ganging 274
- directory (-n) 291
- E (assembler to use) 286
- e (EXE program name) 290
- emulate 80x87 (-f) 276
- enable -F options (-Fm) 276
- enumerations (-b) 417
- environment 291
- error reporting 282
- .EXE file names 290, 291
- expanded memory 287
- extended 80186 instructions (-1) 275
  - f287 (inline 80x87 code) 277
  - f87 (inline 80x87 code) 277
- f (emulate 80x87) 276

far global variables (-Ff) 276  
 far objects (-zE, -zF, and -zH) 284, 285, 584  
 far virtual table segment  
     class name 285  
 fast floating point (-ff) 276  
 fast huge pointers (-h) 277  
 -Fc (generate COMDEFs) 275  
 -Ff (far global variables) 276  
 floating point  
     code generation (-f87) 598  
     emulation (-f) 598  
     fast (-ff) 598  
 -Fm (enable -F options) 276  
 frequent errors 283  
 Fs (assume DS = SS) 276  
 functions, void 282  
 -G (speed optimization) 279  
 generate COMDEFs (-Fc) 275  
 generate underscores (-u) 278  
 gn (stop on n warnings) 282  
 -h (fast huge pointers) 277  
 identifiers, length (-i) 281  
 include files 294  
     directory (-I) 267, 291  
 inline 80x87 code (-f87) 277  
 inline assembler code ( $\pm$ B)  
     inline pragma and 517  
 integer alignment (-a) 275  
 -jn (stop on n errors) 282  
 jump optimization (-O) 280  
 -k (standard stack frame) 277  
 -K (unsigned characters) 277  
 Kernighan and Ritchie keywords (-AK) 281  
 -l (linker options) 291  
 -L (object code and library directory) 267, 291  
 libraries 294  
     directory (-L) 267, 291  
 line numbers (-y) 279  
 link map (-M) 291  
 linker (-l) 290, 291  
 -M (link map) 291  
 macro definitions (-D) 274  
 member pointers (-V and -Vn) 289  
 memory model (-mx) 273  
 merge literal strings (-d) 275  
 -n (.OBJ and .ASM directory) 291  
 -N (stack overflow logic) 278  
 nested comments (-C) 281  
 object code and library directory (-L) 267, 291  
 object files (-o) 286  
 order of evaluation 272  
     response files and 271  
 overlays (-Y) 279, 521, 592  
 overlays (-Yo) 279, 590  
 -P (compile C++) 142  
 Pascal  
     calling conventions (-p) 394, 395, 519, 521  
     conventions (-p) 278  
     identifiers 278  
 pass options to assembler (-Tstring) 287  
 pointer conversion, suspicious 282  
 portability warnings 283  
 pragmas for 517  
 precedence 272  
     response files and 271  
     rules 267  
 precompiled headers (-H) 286  
 process inline assembler (-B) 285  
 produce .ASM but don't assemble (-S) 286  
 -Q (expanded memory) 287  
 -rd (register variables) 280  
 register variables 280  
 remove assembler options (-T-) 287  
 -S (produce .ASM but don't assemble) 286  
 segment-naming control 284  
 speed optimization (-G) 279  
 stack overflow error message (-N) 278  
 standard stack frame (-k) 277  
 stop on n errors (-jn) 282  
 stop on n warnings (-gn) 282  
 structures and 282  
 symbolic debugger 279  
 syntax 270  
 -T- (remove assembler options) 287  
 -Tstring (pass options to assembler) 287  
 template (-Jg) 290  
 toggling 267  
 undefine (-U) 505  
 undefine (-U) 274

- underscores (-u) 278
- UNIX keywords (-AU) 281
- using 266
- v (debugging information) 278, 340
- vi (C++ inline functions) 279
- virtual tables (-V and -Vn) 287
- warnings (-wxxx) 282-284
- word alignment (-a) 413
- Y
  - overlays 521, 592
- y (line numbers) 279
- zV (far virtual table segments) 285
- zX (code and data segments) 284, 285, 584
- Pascal calling conventions, option (-p) 527
- response files 271
- syntax 266
- TLINK and 337
- Turbo Assembler and 270
- using 266
- command-line options
  - IDE 20
- command sets
  - CUA and Alternate 24
  - Native option 27
- commands *See also* individual command names; command-line compiler choosing 23, 24
- editor
  - block operations 630, 631-632
  - cursor movement 629
  - insert and delete 630
- printing
  - MAKE option 323
- commas
  - nested
    - macros and 507
- comments 351
  - // 128, 200, 352
  - /\*\*/ 351
  - as whitespace 350, 352
  - in makefiles 304
  - nested 281, 351
  - token pasting and 351
- Common User Access (CUA) command set 24
- communal variables 275
- \_\_COMPACT\_\_ macro 519
- compact memory model *See* memory models
- compatibility
  - command-line options 565
  - initialization modules 335
  - MAKE 300
  - with Microsoft C 561-569
- compilation 273, *See also* compiler
  - command-line compiler options 285
  - conditional
    - # symbol and 510
    - rules governing 270
    - speeding up 516, 517
- Compile command
  - hot key 27
- compiler
  - diagnostic messages 640-712
- compiler directives *See* directives
- compiling *See* compiler; compilation
- complement
  - bitwise 425, 430
- complex declarations *See* declarations
- complex.h (header file)
  - complex numbers and 601
- complex numbers
  - << and >> operators and 601
  - C++ operator overloading and 601
  - example 601
  - header file 601
  - using 601
- component selection *See* operators, selection (. and ->)
- compound assignment operators 439
- conbuf (class) 541
- concatenating strings *See* strings, concatenating
- conditional breakpoints *See* watch expressions
- conditional compilation
  - # symbol and 510
  - \_\_cplusplus macro and 520
- conditional execution directives (MAKE) 325
  - expressions in 327
- conditional operator (? :) 438
- conditions, boundary 244
- configuration files 33
  - command-line compiler 267, 271
    - creating 272
    - overriding 267, 271, 272
    - priority rules 272



- contents of 33
- IDE 33-35
  - TCCONFIG.TC 33
- conio.h (header file)
  - console control and 607
  - constream and 539
- console
  - I/O
    - functions 607
- console, I/O
  - example program 220
- Console stream manipulators 539, 540
- const (keyword) 102, 391
  - C++ and 391
  - formal parameters and 408
  - pointers and 391, 401
- constant expressions 364
- constants 102, 355, 391, *See also* numbers
  - Turbo C++ 359
  - C++ 202, 391
- case statement
  - duplicate 444
- character 356, 358
  - extending 359
  - integer and 386
  - two-character 359
  - wide 360
- data types 357
- decimal 355, 356
  - data types 357
  - suffixes 357
- enumerations *See* enumerations
- expressions *See* constant expressions
- floating point 356, 360, 361
- fractional 356
- hexadecimal 356, 357
  - too large 282
- integer 355, 356
- internal representations of 363
- manifest 519, *See also* macros
- manifest or symbolic *See* macros
- octal 356
  - too large 282
- pointers and 401
- string *See* strings, literal
- suffixes and 357
- symbolic *See* macros
- syntax 356
  - ULONG\_MAX and UINT\_MAX 433
- constants used by function setf 547
- constrea.h 539, 541
- constream (class) 543
- constructors 137, 211, 468-474, *See also* C++,
  - constructors; initialization
    - accepting default arguments 183
    - arguments 139
  - arrays
    - order of calling 473
  - base class
    - calling
      - from derived class 475
      - order 476
  - calling 469
  - calling with no arguments 183
  - class initialization and 474
- classes
  - base 150
  - derived 150
  - virtual base 473
- copy 471
  - class object initialization and 474
- default 150
- default arguments and 468, 470
- default parameters 471
- defaults 470
- defining 139
- delete operator and 469
- derived class
  - order of calling 476
- inheritance and 468
- inline 139, 144
- invoking 469
- naming 139
- new operator and 138, 469
- non-inline
  - placement of 476
- order of calling 472
  - example 159
- overloaded 472
- unions and 469
- virtual 468

- consumer (streams) 529

- Contents command
    - hot key 26
  - continue statements 84, 447
    - loops and 447
  - continuing lines 350, 363, 508
  - \_control87 (function)
    - floating point exceptions and 600
  - control character
    - insert 631
  - control characters
    - entering in IDE 32
  - control lines *See* directives
  - conventions
    - typographic 6
  - conversion specifications *See* format specifiers
  - conversions 385
    - argument *See* arguments, conversions
    - arguments to strings 508
    - arrays 404
    - BCD 603
    - C++ 534
      - setting base for 534
    - character
      - integers and 386
    - decimal 534
    - floating point
      - ANSI rules 277
    - hexadecimal 534
    - integers
      - character and 386
    - octal 534
    - pointers 403
      - suspicious 282
    - rules 61
    - sign extension and 386
    - special 386
    - specifications *See* format specifiers
    - standard 386
    - table 62
  - coordinates
    - origin 607
    - returning 610
    - starting positions 606, 610
  - coprocessors *See* numeric coprocessors
  - copy and paste *See* editing, copy and paste
  - copy block
    - Borland-style 633
  - Copy command
    - hot key 26
  - copy constructors *See* constructors, copy
  - copy protection 13
  - copy to Clipboard 630
  - coupling
    - loose 242
  - cout (C++ stream) 189
    - flushing 201
    - functions of 122
    - introduced 46
  - cout.precision 47
  - \_\_cplusplus macro 520
  - CPP (preprocessor) *See* The online document
    - UTIL.DOC
  - .CPP files *See* C++
  - CPP.EXE (preprocessor) 501
  - cpm, functions of 122
  - CPU (central processing unit) *See* 80x86
    - processors
  - creating new files *See* files
  - cross-reference
    - IDE 37
  - \_cs (keyword) 392, 582
  - CS register 574, 576
  - CUA command set 24
  - cursor *See also* editing
    - changing 610
    - control
      - header file 607
      - manipulating onscreen 608
    - position
      - setting 608
  - Cursor through tabs 631, 633
  - customer assistance 7
  - Cut command
    - hot key 26
  - cut to Clipboard 630
  - CWx.LIB 336
  - Cx.LIB 336
  - CX register 573
- ## D
- \D escape sequence (display a string of octal digits) 359
  - \$d MAKE macro (defined test) 318
    - expressions and 328

- D MAKE option (define identifier) 300, 316
- D TCC option 505
- D TCC option (macro definitions) 274
- d TCC option (merge literal strings) 275
- /d TLINK option (duplicate symbols) 339
- data
  - declaring 46
  - hiding *See* access
  - inspecting 229
  - structures *See also* arrays; structures
    - naming 109
    - pointers and 112
- data members *See also* member functions
  - access 129, 140, 462
  - defined 128
  - dereference pointers 426, 440
  - member functions and 138
  - private 187, 463
  - protected 463
  - public 462
  - scope 147, 460-463
  - static 459
    - declaration 460
    - definition 460
    - uses 460
- data models *See* memory models
- data segment
  - group 284, 285
  - naming and renaming 284, 285, 584
  - removing virtual tables from 288
- data segments 574
- data structures *See* arrays; structures
- data types 369, *See also* data, *See also* constants;
  - floating point; integers; numbers
    - aggregate 382
    - arithmetic 384
    - BCD *See* BCD
    - Boolean 134, 443
    - C++ streams and 532, 536
  - char 384
    - default, changing 277
    - range 52, 363
    - signed and unsigned 359, 384
    - strings and 72
  - choosing appropriate 53
  - classes and 382
  - conversions *See* conversions
  - converting *See* conversions
  - declarations 383
  - declaring 382
  - default 382
  - derived 382
  - enumerations *See also* enumerations
    - range 52, 363
    - using 110
  - floating point *See* floating point
  - function return types 405
  - fundamental 382, 383
    - creating 384
  - identifiers and 370, 371
  - integers *See* integers
  - integral 384
  - internal representations 384
  - memory use 430
  - new 111
    - defining 390
  - numeric 50
  - parameterized *See* templates
  - promotion 61
  - ranges 52, 363
  - renaming 109
  - scalar 382
    - initializing 387
  - size\_t 431, 481, 482
  - table of 52, 363
  - taxonomy 382
  - template argument 491
  - text\_modes 611
  - typedef and 109
  - types of 382
    - unsigned char
      - range 52, 363
    - void 383
    - wchar\_t 360
- date *See also* time
  - macro 520
  - \_\_DATE\_\_ macro 520
  - #define and #undef directives and 506
- deallocation, memory *See* memory, allocation
- Debug menu 218
- debugging *See also* integrated debugger
  - breakpoints *See* watch expressions
  - Call Stack window 240

- data
  - changing values 234
  - inspecting 229
  - defined 217
  - designing programs for minimum 241
  - Evaluate field and 232
  - exercises 245
    - answers 248
  - expressions
    - changing values 235
  - functions 225
  - hot keys 27
  - information 340
    - command-line compiler option 278
    - in .EXE or OBJ files 278
  - inspectors and 229
  - locating a function 240
  - MAKE 300
  - map files 341
  - multiple files 241
  - multiple variables 234
  - overlays 593
  - PLOTEMP.C 220
  - Step Over command 224
  - syntax errors 223
  - TLINK and 344
  - Trace Into command 225
  - tutorials 217-250
  - User screen and 224
  - values
    - changing 234
    - watch expressions *See* watch expressions
  - windows 239
- dec, ios data member 548
- dec (manipulator) 192, 533, 534
- decimal constants *See* constants, decimal
- declarations 369
  - arrays 403
  - C++ 382
    - incomplete 456
  - complex 397
    - examples 397, 398
  - data *See* data, declaring
  - data types 382
  - defining 370, 375, 377, 388
    - extern keyword and 389
  - examples 383
    - external 375, 380
      - storage class specifiers and 380
    - function 48, *See* functions, declaring
    - global 94
    - incomplete class 456
    - with initializers, bypassing 447
    - location, C++ 203
    - mixed languages 394
    - modifiers and 391
    - objects 135, 378
    - Pascal 394
    - point of 488
    - pointers 400
    - referencing 370, 377
      - extern keyword and 389
    - simple 388
    - static data members 460
    - structures *See* structures, declaring
    - syntax 377, 378
    - tentative definitions and 377
    - unions 417
  - declarators
    - pointers (\*) 367
    - syntax 398
  - decrement operator (--) 63, 425, 428
  - default (label)
    - switch statements and 444
  - default arguments *See* arguments, default
  - default assembler 286
  - default buttons 31
  - default constructors *See* constructors, default
  - #define directive 503
    - argument lists 506
    - command-line compiler options 274
      - ganging 274
    - constants and 102
    - global identifiers and 506
    - keywords and 506
    - redefining macros with 504
      - with no parameters 503
      - with parameters 506
  - defined operator 511
  - defined test macro (MAKE) 318
  - defining declarations *See* declarations, defining
  - definitions *See also* declarations, defining
    - function *See* functions, definitions
    - tentative 377

- delete (operator) 452
  - arrays and 454
  - constructors and destructors and 469
  - destructors and 138, 174, 477, 478
  - dynamic duration objects and 374
  - overloading 481
  - pointers and 477
  - syntax 174
- delete block 630
- delete characters 630
- Delete Item command 253
- delete lines 630
- delete words 630
- delline, conbuf member function 541
- dependencies
  - automatic *See* automatic dependencies
  - checking, MAKE (program manager) 312
- dereferencing *See* referencing and dereferencing
- derived classes *See* classes, derived
- derived data types *See* data types
- descendants *See* classes, derived
- designing programs 87
- desktop files *See* files, desktop
- destructors 468, 476-479, *See also* initialization
  - abort function and 478
  - atexit function and 477
  - auto objects and 174
  - base class pointers and 478
  - calling 469
  - class initialization and 474
  - deallocating memory and 174
  - defined 206
  - delete operator and 138, 174, 469, 477, 478
  - dynamic objects and 174
  - exit function and 477
  - global variables and 477
  - implicit 174
  - inheritance and 468
  - invoking 469, 477
    - explicitly 478
  - new operator and 469, 478
  - pointers and 477
  - #pragma exit and 477
  - static objects and 174
  - unions and 469
  - virtual 468, 478
- DI register 573
- diagnostic messages
  - compiler 640-712
- dialog boxes *See also* buttons; check boxes; list boxes; radio buttons
  - arrows in 31
  - defined 31
  - entering text 32
  - Preferences 633
- dictionary example 204
- digits
  - hexadecimal 356
  - nonzero 356
  - octal 356
- dir.h 567
- direct.h 567
- direct member selector *See* operators, selection (. and ->)
- direct video output 613
- direction flag 574
- directives 304, 511, *See also* individual directive names; macros
  - ## symbol
    - overloading and 480
  - # symbol 368
    - overloading and 480
  - conditional 511
    - nesting 511
  - conditional compilation and 510
  - #define
    - C++ and 202
    - constants and 102
  - defined 46
  - error messages 514
  - #include 46
  - keywords and 506
  - line control 513
  - MAKE *See* MAKE (program manager), directives
  - Microsoft compatibility 566
  - pragmas *See* pragmas
  - sizeof and 431
  - syntax 502
  - usefulness of 501
- directories
  - .ASM and .OBJ
    - command-line option 291

- include files *267, 291, 293*
  - example *295*
  - MAKE *300*
- libraries *294*
  - command-line option *267, 291*
  - example *295*
  - project files *34*
  - projects *254*
- disk space
  - running out of *637*
- disks, distribution, defined *14*
- displays *See screens*
- distribution disks *3*
  - backing up *13*
  - defined *14*
- division *See floating point, division; integers, division*
- division operator (/) *425, 431*
  - rounding *432*
- do while loops *See loops, do while*
- doallocate, strtstreambuf member function *559*
- DOS
  - commands
    - MAKE and *308*
  - environment
    - 87 variable *599*
  - environment strings
    - macros and *318*
  - paths
    - MAKE *323*
- DOS MODE command *See MODE command (DOS)*
- DOS Shell command *23*
- dot directives (MAKE) *323*
- dot operator (selection) *See operators, selection (. and ->)*
- double (floating point) *See floating point*
- double quote character
  - displaying *59, 359*
  - strings and *72*
- DPMI
  - server messages *641*
  - use of extended and expanded memory *16*
- DPMIINST
  - protected mode and *15, 266*
- DPMIMEM environment variable *15*
- DPMIRES protected mode utility *16*

- drawing color *See graphics, colors*
- drawing functions *617*
- \_ds (keyword) *392, 582*
- DS register *574, 576*
- .DSK files
  - default *35*
  - projects and *35*
- dual monitor mode *21*
- dual monitors *See monitors, dual*
- duplicate case constants *444*
- duplicate symbols
  - .LIB and .OBJ files and *339*
  - TLINK and *339*
- duration *99, 373*
  - dynamic
    - memory allocation and *374*
  - local
    - scope and *374*
  - pointers *400*
  - static *373*
- DX register *573*
- dynamic binding *See C++, binding, late*
- dynamic duration
  - memory allocation and *374*
- dynamic memory allocation *See memory, allocation*
- dynamic objects *See objects, dynamic*

## E

- /e IDE option (expanded memory) *See memory, expanded, IDE option (/e)*
- e MAKE option *300*
- E TCC option (assembler to use) *286*
- e TCC option (EXE program name) *290*
- /e TLINK option *339*
- early binding *See C++, binding*
- eatwhite, istream member function *551*
- eback, streambuf member function *556*
- ebuf, streambuf member function *556*
- Edit *See also editing*
  - windows
    - loading files into *256*
- editing *See also Edit; text*
  - block operations *630, 631-632*
  - deleting *632*
  - reading and writing *632*

- commands
  - cursor movement 629
  - insert and delete 630
- copy and paste *See also* Clipboard
  - hot key 26
- hot keys 26
- miscellaneous commands 633-634
- pair matching *See* pair matching
- pasting *See* editing, copy and paste
- selecting text 631
- syntax highlighting 37
- EGA *See* Enhanced Graphics Adapter
- egptr, streambuf member function 556
- elaborated type specifier 456
- elements, parsing 350
- elements of arrays 105
- #elif directive 511
- !elif MAKE directive 325
  - defined test macro and 318
  - macros and 317
- ellipsis (...) 23, 31, 367
  - prototypes and 406, 409
- else clauses *See* if statements
- #else directive 511
- !else MAKE directive 325
- \_\_emit\_\_() 568
- \_emit (keyword) 568
- empty loops 83, 203
- empty statements 442
- empty strings 362
- EMS *See* extended and expanded memory
- EMU.LIB 336, 337
- emulating the 80x87 math coprocessor *See*
  - floating point, emulating
- emulation
  - 80x87 276
- encapsulation 126, *See also* C++
- enclosing block 372
- #endif directive 511
- !endif MAKE directive 325
- endl (manipulator) 192, 534
- ends (manipulator) 192, 534
- Enhanced Graphics Adapter (EGA) *See also*
  - graphics drivers; video adapters
  - color control on 625
  - palette
    - IDE option (/p) 22
- enum (keyword) *See* enumerations
- enum open\_mode, ios data member 548
- enumerations 110, 417
  - C++ 418
  - class names and 456
  - command-line option (-b) 417
  - constants 356, 361, 418
    - default values 361
  - conversions 386
  - default type 417
  - name space 372
  - range 363
  - scope, C++ 419
  - structures and
    - name space in C++ 413
  - tags 418
  - name spaces 419
- enumerations (enum)
  - assigning integers to 282
  - range 52
  - treating as integers 275
- env (argument to main) 523
- environ (global variable) 524
- environment *See also* integrated environment
  - DOS
    - 87 variable 599
    - macros and 318
    - variables 562
- eof, ios member function 549
- eof (member function) 195
- epptr, streambuf member function 557
- equal-to operator (==) 73, 426, 435
- equality operators *See* operators, equality
- error
  - show next 634
  - show previous 634
- #error directive 514
- !error MAKE directive 328
- errors *See also* warnings
  - ANSI 282
  - array size 106
  - C++ streams
    - clearing 195
  - command line
    - defined 640
  - compiler 640-712
    - defined 640

- disk access 640
- DPMI server 641
- expressions 423
- fatal 640
- floating point
  - disabling 600
- frequent 283
- graphics, functions for handling 625
- IDE hot keys 27
- TLIB 642
- MAKE 641
- math, masking 600
- memory access
  - defined 640
- messages 5
  - compile time 255, 256
  - graphics 626
  - list 640-712
  - removing 257
  - saving 257
- next
  - hot key 256
- out of memory 571
- preprocessor directive for 514
- previous
  - hot key 27, 256
- reporting
  - command-line compiler options 282
- run-time 642
- syntax
  - defined 640
  - project files 255, 256
- TLINK (list) 642
- tracking
  - project files 255, 256
- undocumented 641
- \_es (keyword) 392, 582
- ES register 574
- Esc shortcut 31
- escape sequences 58, 356, 358
- Evaluate command
  - inspectors vs. 231
- Evaluate field
  - arrays 232, 233
  - copying into 234
  - debugging and 232
  - display format 233
  - expressions in
    - rules governing 232
  - format specifiers and 233
  - memory dump and 233
  - pointers and 233
  - variables 232
- Evaluate/Modify command
  - hot key 27
- evaluation order *See also* precedence
  - command-line compiler options 272
  - in response files 271
- examples
  - library and include directories 295
  - MAKE (program manager) 301
    - batch files 307
  - relational operators 73
- exclusive OR operator (^) 64, 425, 436
- truth table 436
- .EXE files
  - .COM files and 344
  - creating 25, 27
  - debugging information 344
  - TLINK and 343
  - user-selected name for 290
- executable files *See* .EXE files
- execution
  - bar *See* run bar
  - line-by-line 225
  - stepping over functions 224
- exit (functions)
  - destructors and 477
- exit codes
  - MAKE and 305
- exit pragma 515
- exit the IDE 631
- exiting Turbo C++ 23
- expanded memory 16, *See also* extended and
  - expanded memory
    - controlling use of 287
  - IDE option *See* memory, expanded, IDE
  - option (/e)
  - TLINK and 344
- explicit
  - library files 291
  - rules (MAKE) 304, 309
- explicit template function 493
- exponents 356



- expressions
  - assigning values in 68
  - associativity 420
  - cast, syntax 429
  - conditional 74
  - constant 364
  - conversions and 385
  - debugging
    - changing values of 235
  - decrementing 428
  - defined 56, 66
  - displaying value of 232
  - empty (null statement) 366, 442
  - errors and overflows 423
  - evaluating 66
  - function
    - sizeof and 431
  - grouping 365
  - incrementing 428
  - MAKE and 327, 328
  - precedence 420, 422, 423
  - statements 366, 442
  - syntax 421
  - table 421
- extended 80186 instructions 275
- extended and expanded memory
  - \_OvrInitEms and 594
  - \_OvrInitExt and 595
  - overlays and 594
  - RAM disk and *See* RAM disk, IDE option
  - (/r) and
  - swapping 594, 595
- extended memory 16, 17
  - IDE option (/x) and 22
  - TLINK and 345
- extension keywords
  - ANSI and 281
- extensions 353
- extensions, file, supplied by TLINK 333
- extent *See* duration
- extern (keyword) 389, *See also* identifiers,
  - external
    - arrays and 403
    - class members and 457
    - const keyword and 391
    - linkage and 375
    - name mangling and 376

- using 100
- external
  - declarations 375
  - identifiers *See* identifiers, external
  - linkage *See* linkage
- External option
  - C++ Virtual Tables
    - command-line option 288
- extra segment 574
- extraction operator (>>) *See* overloaded
  - operators, >> (get from)
- extractors *See* input, C++

## F

- f287 option (inline 80x87 code) 277
- f87 command-line compiler option (generate floating-point code) 598
- f87 option (inline 80x87 code) 277
- \f (formfeed character) 59
- f command-line compiler option (emulate floating point) 598
- \f escape sequence (formfeed) 359
- f MAKE option (MAKE file name) 298, 300
- f TCC option (emulate 80x87) 276
- fail, ios member function 549
- fail (member function) 195
- far
  - calls
    - memory model and 592
    - requirement 592
  - functions *See* functions, far
  - objects *See* objects, far
  - pointers *See* pointers, far
  - variables 276
- far (keyword) 392, 576, 582
- far calls 587
- far objects *See* objects, far
- far virtual table segment
  - naming and renaming 285
- fast huge pointers 277
- \_fastcall (keyword) 392
- fatal errors *See also* errors
  - Compile-time 640
- Fc TCC option (generate COMDEFs) 275
- fd, filebuf member function 544
- features of Turbo C++ 1

- ff command-line compiler option (fast floating point) 276, 598
- Ff TCC option (far global variables) 276
- field width *See* formatting
- field width, C++ 192
- \_\_FILE\_\_ macro 520
  - #define and #undef directives and 506
- file descriptor 544
- file-inclusion directive (`#include`) 324
- file-name macros (MAKE) 320, 321
- file scope *See* scope
- filebuf (class) 543
- filename macros (MAKE) 320
- files *See also* individual file-name extensions
  - ARGS.EXE 524
  - .ASM *See* assembly language
  - assembly language *See* assembly language
  - batch *See* batch files
  - binary 123
  - buffers
    - C++ 543, 545
  - buffers, C++ 195
  - .COM 335, 343
    - .EXE files and 344
    - TLINK and 343, 344
  - compiling as C++ or C 286
  - configuration 33, *See* configuration files
  - current
    - macro 520
  - desktop (.DSK)
    - default 35
    - projects and 35
  - disk
    - copying using C++ 193
    - reading 122
    - writing 122
  - editing *See* editing
  - executable *See* .EXE files
  - extensions 333
  - font *See* fonts
  - graphics driver, linking 617
  - header *See* header files
  - HELPME!.DOC 14, 17
  - I/O
    - example program 220
  - include *See* include files
  - including 509
  - information in dependency checks 257
  - library (.LIB) *See* libraries
  - loading into editor 256
  - make *See* MAKE (program manager)
  - map *See* map files
  - modifying 18
  - multiple *See* projects
  - names
    - extensions (meanings) 333
  - opening 548, 631, 633
    - default mode 537, 543
    - hot key 25
    - openprot 543
  - out of date, recompiled 257
  - position seeking 548
  - project 33, *See also* projects
    - graphics library listed in 614
  - README 17
  - README.DOC 14
  - response *See* response files
  - saving 631, 633
    - hot key 25
  - scope *See* scope
  - source
    - .ASM
      - command-line compiler and 265
  - streams
    - C++ operations 545
  - .TC *See* configuration files, integrated environment
  - using
    - example program 123
    - WILDARGS.OBJ 525, 526
- fill, ios member function 549
- fill characters
  - C++ 534
- filling functions 617
- financial applications 602
- Find command *See also* searching
- fixed, ios data member 548
- flags
  - format state *See also* formatting
  - ios (class)
    - setting 534
  - ios member function 549
  - register 572, 573
- floatfield, ios data member 547

floating point *See also* data types; integers; numbers; numeric coprocessors  
 ANSI conversion rules 277  
 constants *See* constants  
 conversions *See* conversions  
 defined 54  
 division 54  
 double  
   defined 55  
   long *See* floating point, long double  
   range 52, 363  
 emulating 598  
 exceptions  
   disabling 600  
 expressions  
   precedence 423  
 fast 276, 598  
 inline 80x87 operations 277  
 libraries 276, 597  
   TLINK and 336  
 long double  
   defined 55  
   range 52, 363  
 math coprocessor and 277  
 Microsoft C and 568  
 precision  
   setting 534  
 ranges 52, 363  
 registers and 600  
 using 597  
 flow-control statements *See* if statements; switch statements  
 flush (manipulator) 192, 534  
 flush, ostream member function 554  
 -Fm TCC option (enable -F options) 276  
 fonts  
   bit-mapped  
     stroked vs. 621  
     when to use 621  
   clipping 621  
   files  
     loading and registering 621  
   height and width 621  
   information on current settings 628  
   registering 622  
   setting size 621  
   stroked  
     advantages of 621  
   for loops *See* loops, for  
   foreground color *See* graphics  
   format parameters *See* parameters, formal  
   format flags 548  
   format specifiers *See also* formatting  
     characters and strings and 72  
     Evaluate field and 233  
     memory dump 233  
     pointers and 233  
   format state flags *See* formatting, C++, format state flags  
   formatting *See also* format specifiers  
     C++  
       classes for 530  
       field width 192  
       fill character 534  
       format state flags 191, 548  
       formatting flags 547  
       I/O 534, *See also* manipulators  
       output 532  
       padding 534  
       put and write functions and 192  
       width functions *See also* manipulators  
         setting 534  
     C++ I/O 191, 193  
     escape sequences and 58  
     streams and  
       clearing 534  
   formfeed character (\f) 59, 359  
   fortran (keyword) 568  
     \_pascal keyword and 568  
   forward references 370  
   Fourier transforms  
     complex number example 601  
   FP87.LIB 337  
   FP\_OFF 586  
   FP\_SEG 586  
   free (function)  
     delete operator and 452  
     dynamic duration objects and 374  
   freeze, ostreambuf member function 559  
   frequent errors 283  
   friend (keyword) 457, 466-468  
     base class access and 465  
     classes and 188

- functions and *See* C++, functions
- friend functions *See* C++, functions
- Fs TCC option (assume DS = SS) 276
- fstream (class) 545
- fstream.h (header file) 122
- fstreambase (class) 545
- full file name macro (MAKE) 319
- full link map 291
- function call operator *See* parentheses
- function operators *See* overloaded operators
- function signature 162
- function template 492
- functions 404-409, *See also* individual function
  - names; member functions; scope
  - accessing variables and 113
  - arguments
    - no 406
  - attribute control 609
  - C-type 278
  - calling 240, 408, *See also* parentheses
    - operators ( ) 427
    - overloading operator for 484
    - rules 408
  - cdecl and 395
  - class names and 456
  - color control 622
  - comparing 436
  - console
    - I/O 607
  - debugging 224, 225
  - declarations
    - global 94
  - declaring 404, 405
    - as near or far 585
    - under Kernighan and Ritchie 89
  - default types for memory models 396
  - defined 45
  - defining 89
  - definitions 404, 407
  - drawing 617
  - duration 374
  - error-handling, graphics 625
  - exit 515
  - external 389
    - declarations 375
  - far 396
    - declaring 585

- memory model size and 585
- filling 617
- finding 240
- friend *See* C++, functions, *See* C++, functions, friend
- graphics *See also* graphics
  - drawing operations 617
  - fill operations 618
    - using 614-628
- graphics system control 615
- header 89
- huge 396
  - assembly language and 396
  - \_loads and 396
  - saving registers 518
- image manipulation 619
- inline
  - assembly language *See* assembly language
  - C++ 179, 202, 458
    - classes and 204
    - linkage 460
    - precompiled headers and 636
    - syntax 180
  - inspecting 231
- internal linkage 390
- interrupt *See* interrupts
- linking C and C++ 376
- main 404
- member *See* member functions
- memory
  - models and 392
- mode control 609
- multiple 91
- name mangling and 376
- near 396
  - declaring 585
    - memory models and 585
- no arguments 383
- not returning values 383
- one line 202
- operators *See* overloaded operators
- ordinary member *See* member functions, ordinary
- overloaded *See* overloaded functions
- parameters *See* arguments; parameters
- Pascal
  - calling 394

- passing arrays to C++ 220
- pixel manipulation 619
- pointers 113, 119, 399
  - calling overlaid routines 593
  - object pointers vs. 398
  - void 399
- prototypes 88, *See also* prototypes
- recursive
  - memory models and 585
  - return statements and 447
  - return types 405
  - return values 90
  - scope *See* scope
  - screen manipulation 619
  - signature 162
  - sizeof and 431
  - startup 515
  - state queries 610, 626
  - static 375
  - stdarg.h header file and 406
  - stepping over 224
  - storage class specifiers and 376
  - structures and 411
  - text
    - manipulation 608
    - output
      - graphics mode 620
  - tracing through 225
  - type, modifying 396
  - usefulness of 50
  - user-defined 88
  - viewport manipulation 619
  - virtual *See also* member functions, virtual
    - specifying pure 486
  - void
    - returning a value 282
  - windows 609
  - writing your own 88
- fundamental data types *See* data types

## G

- G TCC option (speed optimization) 279
- ganging
  - command-line compiler options
    - #define 274
    - macro definition 274
  - defined 274, 293

- IDE 294
  - library and include files 294
- gbump, streambuf member function 557
- gcount, istream member function 551
- generic pointers 383, 400
- generic types 490
- get
  - istream member function 552
- get (function) 193
  - streams and 122
- get, istream member function 552
- get from operator (>>) *See* overloaded operators, >> (get from)
- getch (function) 70
- getche (function) 70
- getline, istream member function 552
- getmaxx (function)
  - example 95
- getmaxy (function)
  - example 95
- global declarations *See* declarations, global
- global identifiers *See* identifiers, global
- global menus *See* menus
- global variables 372, *See also* variables
  - case sensitivity and 392
  - destructors and 477
  - environ 524
  - \_ovrbuffer 589, 592
  - scope and 100
  - underscores and 392
  - word-aligning 275
  - \_wscroll 608
- gn TCC option (stop on *n* warnings) 282
- Go to Cursor command
  - hot key 25, 27
- good, ios member function 549
- goto statements 447
  - labels
    - name space 372
- gotoxy, conbuf member function 541
- gptr, streambuf member function 557
- grammar, tokens *See* tokens
- graphics *See also* graphics drivers
  - buffers 620
  - charts 220
  - circles
    - aspect ratio 619

- classes 133
- colors *See also* graphics, palettes
  - background 609
    - CGA 624
    - defined 612, 623
    - list 613
    - setting 609
  - CGA 623, 624
  - drawing 623
  - EGA/VGA 625
  - foreground 609
    - CGA 624
    - defined 612
    - list 613
    - setting 609
  - functions 622
  - header file 95
  - information on current settings 628
  - symbolic names 95
- coordinates 95, *See also* coordinates
- default settings
  - restoring 616
- displaying 623
- drawing functions 617
- errors
  - functions to handle 625
- fill
  - operations 618
  - patterns 618
    - header file 95
    - using 627
- functions
  - using 614-628
- header file 94, 614
- library 614
  - TLINK and 336
- line style 618
- memory for 617
- page
  - active
    - defined 620
    - setting 619
  - visual
    - defined 620
    - setting 619
- palettes *See also* graphics, colors
  - defined 622
  - functions 622
  - information on current 628
- pixels *See also* screens, resolution
  - colors
    - current 628
    - functions for 619
    - setting color of 622
  - setting
    - clearing screen and 620
  - state queries 626
  - system
    - control functions 615
    - shutting down 616
    - state queries 627
  - text and 620
  - viewports
    - defined 607
    - functions 619
    - information on current 628
- graphics drivers *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics; video adapters; Video Graphics Array Adapter (VGA)
  - current 616, 627
    - returning information on 628
  - linking 617
  - loading and selecting 616, 617
  - new
    - adding 616
    - registering 617
    - returning information on 627, 628
    - supported by Turbo C++ 615
    - testing for presence 94
  - graphics.h (header file) 94, 614
  - greater-than operator (>) 73, 426, 434
  - greater-than or equal to operator (>=) 73
  - greater-than or equal-to operator (>=) 426, 434
  - grep *See* The online document UTIL.DOC

## H

- /h IDE option (list options) *See* help, IDE option (/h)
- h MAKE option (help) 300
- h TCC option (fast huge pointers) 277
- H TCC option (precompiled headers) 286

- hardware
  - requirements
  - mouse 3
  - to run Turbo C++ 3
- hdrfile pragma 516, 636, 637
- hdrstop pragma 517, 636, 638
- header files *See also* include files
  - Turbo C++ versus Microsoft C 567
  - C++ 148
  - complex numbers 601
  - defined 46, 97
  - extern keyword and 377
  - fstream.h 122
  - function prototypes and 406
  - graphics 614
  - graphics.h 94
  - #include directive and 509
  - inline C++ functions and 144
  - Microsoft C 567
  - name mangling and 377
  - precompiled 516, 517, *See also* precompiled
  - headers
    - prototypes and 404
    - searching for 294
    - stream.h vs. stdio.h 200
    - using 98
    - variable parameters 406
- heap 374
  - extended memory for *See* extended memory,
  - IDE option (/x) and
  - objects *See* objects, dynamic
- Help 631
  - button 31
  - hot keys 25, 26
  - IDE and 36
  - IDE option (/h) 21
  - index 631
  - MAKE 300
  - status line 31
  - topic search 631
- HELPME!.DOC file 14, 17
- Hercules card *See* graphics drivers; video adapters
- hex, ios data member 548
- hex (manipulator) 192, 533, 534
- hexadecimal
  - constants *See* constants, hexadecimal digit 356
  - numbers *See* numbers, hexadecimal
- hexadecimal numbers *See* numbers, hexadecimal
- hidden objects 373
- hiding *See* scope, C++
- hierarchies *See* classes
- highvideo, conbuf member function 542
- history lists 32
- horizontal tab character 359
- horizontal tab character (\t) 59
- hot keys
  - debugging 27
  - editing 26
  - help 25, 26
  - make project 256
  - menus 24, 25
  - next error 256
  - previous error 256
  - using 24
- huge
  - functions
    - saving registers and 518
    - memory model *See* memory models
    - pointers *See* pointers, huge
  - \_\_HUGE\_\_ macro 519
- huge (keyword) 392, 576, 582
  - assembly language and 396
- hyphen (-) MAKE command (ignore exit status) 305

|

- i MAKE option (ignore exit status) 300
- I MAKE option (include files directory) 300, 301
- i TCC option (identifier length) 281
- I TCC option (include files directory) 267, 291
- /i TLINK option (uninitialized trailing segments) 340
- icons used in books 6
- IDE 19, *See also* integrated environment; Integrated Development Environment
- command-line options 20
- syntax 20
- control characters and 32

- menu cross-referenced to information 37
- overlays and 591
- starting up 20
- identifiers 57, 354
  - Turbo C++ keywords as 281
  - case 394
    - sensitivity and 354
  - classes 455
  - data structures 109
  - data types and 370, 371
  - declarations and 370
  - declaring 388
  - defined operator and 511
  - defining 316, 505
  - duplicate 373
  - duration 373
  - enumeration constants 361
  - external *See also* extern (keyword)
    - name mangling and 376
  - global 519
    - #define and #undef directives and 506
  - linkage 375
    - attributes 375
    - maximum length 354
    - mixed languages 394
    - name spaces *See* name spaces
    - no linkage attributes 376
    - Pascal 394
    - pascal (keyword)
      - case sensitivity and 355
    - Pascal-type 278
    - rules for creating 354
    - scope *See* scope
    - significant length of 275, 281
    - storage class and 371
    - testing for definition 512
    - undefining 274, 505
    - underscore for 278
    - unique 375
    - variables 57
- IEEE
  - floating-point formats 385
  - rounding 604
- #if directive 511
- !if MAKE directive 325
  - defined test macro and 318
  - macros and 317
- if statements 74, 443
  - else 75
  - nested 443
  - nesting 76
- #ifdef directive 512
- !ifdef MAKE directive 325
- #ifndef directive 512
- !ifndef MAKE directive 325
- ifstream (class) 546
  - constructor 537
  - insertion operations 537
- ignore, istream member function 552
- ignore exit status (MAKE command) 305
- .ignore MAKE directive 323
- implicit
  - library files 291
  - rule (MAKE) 304
- in, ios data member 548
- in\_avail, streambuf member function 557
- #include directive *See also* include files
  - angled brackets and 294
  - quotes and 294
- !include directive (MAKE) 301, 324
- INCLUDE environment variable 562
- include files *See also* header files
  - automatic dependency checking (MAKE) 312
  - command-line compiler options 294
  - defined 46
  - directories 267, 291, 293
    - multiple 295
  - #include directive and 509
  - MAKE 301, 324
    - directories 300
  - paths 562
  - projects 253
  - search algorithm for 509
  - searching for 294
  - user-specified 267, 291
- Include Files command 253
- #include directive 509
  - defined 46
  - search algorithm 509
- inclusive OR operator (|) 425, 437
  - truth table 436
- incomplete declarations
  - classes 456
  - structures 414



- increment operator (++) 63, 425, 427, 428
- incremental search 33
- indent block 630
- indeterminate arrays 403
  - structures and 404
- Index command
  - hot key 26
- indexes *See* arrays
- indirect member selector *See* operators, selection
- indirection 120
- indirection operator (\*) 425, 429
  - pointers and 402
- inequality operator (!=) 426, 436
- information hiding *See* access
- inheritance 130, 144, *See also* classes
  - access and 146
  - base and derived classes and 145
  - constructors and destructors 468
  - defined 126
  - example 209
  - functions and 158
  - multiple 132, 155
    - base classes and 466
    - defined 146
  - overloaded assignment operator and 484
  - overloaded operators and 481
  - rules 147
- init, ios member function 549
- initgraph (function)
  - path argument and 91
- initialization 386, *See also* constructors; destructors
  - arrays 387
  - classes 474
    - objects 474
      - copy constructor and 474
  - constructors and destructors and 137
  - operator 368
  - pointers 400
  - static member definitions and 460
  - structures 387
  - unions 387, 417
  - variables 388
- initialization modules 565
  - compatibility 335
  - used with TLINK 334, 335
- initialize a reference variable 475
- initialized data segment *See* data segment
- initializer list 475
- initializers
  - automatic objects 389
  - C++ 389
    - (:) 426
  - new operator and 454
- initializing
  - arrays 104, 106
  - variables 55
- inline
  - assembly language code *See* assembly language, inline
  - expansion 458
  - functions *See* functions, inline
  - keyword 458
  - pragma 517
- inline (keyword) 202
  - classes and 204
  - constructors and 139
  - member functions and 136
- inline assembly code 285
- inline code *See* assembly language, inline routines; numeric coprocessors
- inline functions, C++ *See* C++, functions, inline
- input
  - C++
    - user-defined types 536
- input boxes 32
- insert lines 630
- insert mode 630
- inserters types 532
- inserters *See* output, C++
- insertion operator *See* overloaded operators, << (put to)
- insertion operator (<<) *See* overloaded operators, << (put to)
- insline, conbuf member function 542
- Inspect command
  - hot key 26, 27
- inspectors
  - abilities of 229
  - arrays 230
  - data 229
  - Evaluate command vs. 231
  - functions 231

- strings 230
- structures 230
- unions 230
- installation 14-17
  - on a laptop system 17
- instances *See* classes, instantiation and; classes, objects
- instantiation *See* classes, instantiation and
- integers 384, *See also* data types; floating point; numbers
  - aligned on word boundary 275
  - assigning to enumeration 282
  - C++ streams and 532
  - constants *See* constants
  - conversions *See* conversions
  - division 54
  - expressions
    - precedence 423
  - long 384
    - range 52, 363
    - using 53
  - memory use 384
  - range 52, 363
  - short 384
  - sizes 384
  - suffix 356
  - unsigned
    - range 52, 363
  - using 52
- integral data types *See* characters; integers
- integrated debugger *See also* debugging
  - breakpoints *See* watch expressions
- Integrated Development Environment *See also* integrated environment
- integrated development environment
  - nested comments command 352
- integrated environment
  - configuration files *See* configuration files, IDE
  - customizing 18
  - debugging *See* debugging
  - editing *See* editing
  - ganging 294
  - INCLUDE environment variable and 562
  - LIB environment variable and 562
  - makes 257
- menus *See* menus
- multiple library directories 294
- Programmer's Workbench and 561
- wildcard expansion and 526
- intensity
  - setting 609
- internal, ios data member 548
- internal linkage *See* linkage
- internal representations of data types 384
- interrupt (keyword) 392, 393
- interrupts
  - flag 574
  - functions
    - memory models and 393
    - void 393
  - handlers 392
  - modules and 593
  - registers and 393
- INTRO2.CPP (sample program) 48
- I/O
  - C++ *See also* C++, I/O
    - formatting 534
    - precision 534
  - disk 193
- iomanip.h (header file) 192
  - manipulators in 533
- ios (class) 530, 547
  - flags *See* formatting
- ios data members 547
- iostream (class) 551
- iostream.h (header file) 46
  - manipulators in 533
- iostream library 530
  - introduced 45
- iostream\_withassign (class) 551
- IP (instruction pointer) register 572
- is\_open, filebuf member function 544
- istream 189
- istream (class) 551
  - derived classes of 537
- istream\_withassign (class) 553
- istrstream (class) 553
- iteration statements *See* loops

## J

- jg family of template switches 497
- jn TCC option (stop on *n* errors) 282

jump optimization 280  
jump statements *See* break statements; continue statements; goto statements; return statements

## K

-K MAKE option (keep temporary files) 300, 306  
-k TCC option (standard stack frame) 277  
-K TCC option (unsigned characters) 277  
K&R *See* Kernighan and Ritchie (K&R)  
Keep Messages command  
  toggle 257  
Kernighan and Ritchie (K&R)  
  function declarations 89  
  keywords 281  
keyboard  
  choosing commands with 23, 31  
  reading characters from 69  
  stream 122  
keys, hot *See* hot keys  
keywords 353, *See also* individual keyword names  
  ANSI  
    command 281  
    predefined macro 521  
  auto 101  
  Turbo C++  
    using, as identifiers 281  
  C++ 353  
  class 204  
  combining 384  
  const 102  
  extern 100  
  inline 202  
    classes and 204  
  Kernighan and Ritchie  
    using 281  
  macros and 506  
  Microsoft C 568  
  new  
    malloc and 205  
  operator 184  
  register 102  
  static 101  
  typedef 109

UNIX  
  using 281  
  void 88

## L

/l IDE option (LCD screen) *See* LCD displays, IDE option (/l)  
-l TCC option (linker options) 291  
-L TCC option (object code and library directory) 267, 291  
/l TLINK option (line numbers) 340  
labeled statements 442  
labels  
  creating 367  
  default 444  
  function scope and 372  
  goto statement and 447  
laptops  
  IDE option (/l) 22  
  installing Turbo C++ onto 17  
\_\_LARGE\_\_ macro 519  
large memory model *See* memory models  
late binding *See* C++, binding  
LCD displays  
  IDE option (/l) 22  
  installing Turbo C++ for 17  
left, ios data member 548  
less-than operator (<) 73, 426, 434  
less-than or equal to operator (<=) 73  
less-than or equal-to operator (<=) 426, 434  
lexical grammar *See* elements  
LIB environment variable 562  
.LIB files *See* libraries  
librarian *See* TLIB  
libraries 336  
C  
  linking to C++ code 376  
  class 148  
  command-line compiler options 294  
  container class *See* The online document CLASSLIB.DOC  
  directories 293  
    command-line option 267, 291  
    multiple 295  
  duplicate symbols in 339  
  explicit and implicit 291  
  files 267, 291

- floating point 276
  - TLINK and 336
  - using 597
- graphics 614
  - TLINK and 336
- iostream 530
- memory models 337
- numeric coprocessor 336
- order of use 336
- overriding in projects 260
- paths 562
- prototypes and 409
- rebuilding 279
- routines
  - 80x87 floating-point emulation 277
- run time
  - TLINK and 337
- searching for 294
- streams 188, 529
- TLINK and 334, 336
  - ignoring 342
  - user-specified 291
  - using 98
  - utility *See* TLIB
- library files *See* libraries
- license statement 13
- line
  - mark a 630
- \_\_LINE\_\_ macro 520
  - #define and #undef directives and 506
- #line directive 513
- line numbers *See* lines, numbering
- lines
  - continuing 350, 363, 508
  - delete 630
  - insert 630
  - numbering 28
    - in object files 279
    - TLINK and 340
  - numbers 513
    - macro 520
- LINK (Microsoft)
  - TLINK versus 566
- link map, full 291
- linkage 375
  - C and C++ programs 376
  - external 375
    - C++ constants and 391
      - name mangling and 376
    - internal 375
    - no 375, 376
    - rules 375
    - static member functions 460
    - storage class specifiers and 375
    - type-safe 409, 493, 495
- linker *See also* TLINK
  - command-line compiler options 290, 291
  - link map
    - creating 291
  - mixed modules and 586
  - options
    - from command-line compiler 291
    - using directly 586
- list boxes 33
- List command
  - hot key 26
- literal strings *See* strings, literal
- load operations
  - redundant, suppressing 280
- \_loadds (keyword) 392
  - huge functions and 396
  - uses for 396
- local duration 374
- local menus *See* menus
- Local Options
  - C++ Virtual Tables
    - command-line option 288
    - command 253
- Locate Function command 240
- logical AND operator (&&) 74, 425, 437
- logical negation operator (!) 425, 430
- logical OR operator (||) 74, 425, 437
- long double (floating point) *See* floating point, long double
- long integers *See* integers, long
- loops 444
  - break statement and 446
  - choosing 87
  - continue statement and 447
  - defined 79
  - do while 81, 445
    - exiting 84
  - empty 83, 203
  - exiting 84

- for 82, 445
  - C++ 446
  - while loop and 84
- nested 86
- while 80, 444
  - exiting 84
  - for loop and 84
  - string scanning and 445
- loose coupling 242
- lowvideo, conbuf member function 542
- lvalues 370, *See also* rvalues
  - examples 397
  - modifiable 370

## M

- /m IDE option (MAKE) *See* MAKE (program manager), IDE option (/m)
- m MAKE option (display time/date stamp) 300
- M TCC option (link map) 291
- macros 87, *See also* MAKE (program manager), macros
  - argument lists 506
  - calling 506
  - command-line compiler 274
  - commas and, nested 507
  - defining 503
    - conflicts 504
    - global identifiers and 506
  - DOS
    - environment strings and 318
    - path (MAKE) 323
  - expansion 503
  - far pointer creation 586
  - ganging 274
  - invocation
    - defined 316
  - keywords and 506
  - MAKE *See* MAKE (program manager), macros
  - MK\_FP 586
  - parameters and 506
    - none 503
  - parentheses and
    - nested 507
  - precedence in
    - controlling 365

- predefined 519, *See also* individual macro names
  - ANSI keywords 521
  - C and C++ compilation 521, 522
  - C calling conventions 519
  - conditional compilation 520
  - current file 520
  - current line number 520
  - date 520
  - DOS 521
  - memory models 519
  - overlays 521
  - Pascal calling conventions 521
  - templates 521
  - time 521
- random 108
- redefining 504
- side effects and 508
- transfer *See* The online document UTIL.DOC
- Turbo editor *See* The online document UTIL.DOC
- undefining 504
  - global identifiers and 506
- main (function) 45, 404, 523-527
  - arguments passed to 523
    - example 524
    - wildcards 525
  - compiled with Pascal calling conventions 527
  - declared as C type 527
  - pascal keyword and 394
  - value returned by 527
- MAKE (program manager)
  - automatic dependency checking 300, 312
  - batching files and 307
  - BUILTINS.MAK file 301
  - clocks and 298
  - commands
    - @ (hide commands) 305
    - ampersand (&) (multiple dependents) 305
    - hiding (@) 305
    - hyphen (-) (ignore exit status) 305
    - num (stop on exit status num) 305
  - compatibility 300
  - debugging 300
  - directives
    - .noautodepend 323
    - .autodepend 323

- command-line compiler options and 323
- conditional execution 325
  - expressions in 327
- defined 322
- dot 323
- !elif 325
  - macros and 317
- !else 325
- !endif 325
- !error 328
- file inclusion 324
- !if 325
  - macros and 317
- !ifdef 325
- !ifndef 325
- !ignore 323
- !include 324
- !noignore 323
- !nosilent 323
- !noswap 323
- !silent 323
- !swap 323
- !undef 328
- DOS commands and 308
- example 301
- exit codes and 305
- explicit rules *See* MAKE (program manager), rules
- external commands and 308
- functionality 297
- hide commands 305
- IDE option (/m) 22
- implicit rules *See* MAKE (program manager), rules
- !include directive 301
- integrated environment makes and 257
- macros 305, 313, 315, 318
  - \$? 305
  - \$\*\* 305
  - all dependents (\$\*\*) 321
  - all out of date dependents (\$) 321
  - base file name (\$) 319
  - defined test 318
  - !elif directive and 317, 318
  - example 313
  - file name and extension (.\$) 320
  - file name only (\$&) 320
  - file name path (.\$) 320
  - full file name (\$<) 319
  - full name with path (\$&) 320
  - !if directive and 317, 318
  - in expressions \$d 328
  - \_\_MAKE\_\_ 318
  - predefined 318
  - undefining 328
  - version number 318
  - \_\_MAKE\_\_ macro 318
- makefiles
  - comments in 304
  - creating 303
  - defined 302
  - naming 303
  - parts of 304
- Microsoft C and 563
- multiple dependents and 305
- NMAKE vs. 329
- operators 327
- options 299
  - ? (help) 300
  - (increase compatibility) -N 300
  - (print commands but don't execute) -n 300
  - (save options) -W 300
  - automatic dependency checking (-a) 312
  - build all (-B) 300
  - default (-w) 300
  - define identifier (-D) 300
    - conditional execution 325
  - display rules (-p) 300
  - display time/date stamp (-m) 300
  - don't print commands (-s) 300
  - environment variables(-e) 300
  - file name (-f) 298, 300
  - help (-? and -h) 300
  - ignore BUILTINS.MAK (-r) 300
  - ignore exit status (-i) 300
  - include files directory (-I) 300, 301
  - keep files (-K) 300, 306
  - saving (-w) 300
  - swap MAKE out of memory (-S) 300
  - undefine (-U) 300
  - using 299
- .path directive 323
- .precious directive 323
- printing commands 323

- redirection operators 305
- rules
  - explicit
    - considerations 310
    - defined 309
    - example 304, 311
  - implicit 303
    - discussion 312
    - example 304
    - explicit rules and 313
  - stopping makes 255
  - swapping in memory 323
  - syntax 299
  - wildcards and 309
- Make command, hot key 25, 27
- MAKE.EXE 298
- makefiles *See* MAKE (program manager)
- malloc (function)
  - dynamic duration objects and 374
  - new and 205
  - new operator and 452
- malloc.h (header file)
  - alloc.h and 567
- mangled names 376
- manifest constants 519, *See also* macros
- manipulators 192, 533, *See also* formatting;
  - individual manipulator names
  - header file for 192
  - introduced 47
  - parameterized 192, 533
  - syntax 534
  - user-defined 197
- manual, using 6
- map files 291
  - debugging 341
  - generated by TLINK 340
- marker
  - find 631, 633
  - set 631, 634
- math
  - BCD *See* BCD
  - coprocessors *See* numeric coprocessors
  - errors
    - masking 600
  - math coprocessors *See* numeric coprocessors
  - matherr (function)
    - proper use of 600
  - MATHx.LIB 336
  - maximize *See* Zoom command
  - \_\_MEDIUM\_\_ macro 519
  - medium memory model *See* memory models
  - mem.h (header file)
    - memory.h and 567
  - member functions 135, 457, *See also* C++,
    - functions; data members
      - access 129, 140, 204, 462
      - access to variables 138
      - adding 135
      - calling 136
      - choosing type 172
    - constructors *See* constructors
    - data
      - access 129
      - defined 128, 457
      - defined outside the class 136
    - destructors *See* destructors
    - example 138
    - friend 457, *See also* functions, friend
    - inline 135, 136, *See* functions, inline, C++
    - nonstatic 457
    - open and close 193
    - ordinary
      - problems with inherited 161
      - virtual vs. 161, 167, 172
    - overriding 159
    - positioning in hierarchy 166
    - private 463
    - protected 463
    - public 462
    - scope 460-463
    - signature 162
    - static 459
      - linkage 460
      - this keyword and 459
    - stream state 195
    - structures and 411
    - this keyword and 457, 459
    - unions and 416
    - virtual 160, 161, 163, *See also* C++, binding,
      - late
      - ordinary vs. 167, 172
      - pros and cons 166
      - syntax 163

- member pointers
  - controlling 289
- members
  - data *See* data members
  - functions *See* member functions
- members, classes *See* data members; member functions
- members, structures *See* structures, members
- memory *See also* memory addresses
  - allocation 374
    - assembly language code and huge functions and 396
    - graphics system 617
    - new and delete operators and 452
    - structures 413
  - Turbo C++'s usage of 571
  - data types 430
  - deallocating
    - destructors and 174
  - dump
    - Evaluate field and 233
  - expanded 16
    - controlling 287
    - IDE option (/e) 21
  - extended 16
  - extended and expanded *See* extended and expanded memory
  - freeing
    - automatically 101
  - heap 374
  - memory models and 579
  - overlays and 589
  - paragraphs 575
    - boundary 575
  - protected mode and 14
  - RAM disk and *See* RAM disk, IDE option (/r) and
  - segments in 575
  - strings and 71
  - swapping MAKE in 323
  - word alignment and
    - structures 413
- memory addresses *See also* memory
  - calculating 573, 575-576
  - constructors and destructors 468
  - far pointers and 576
  - near pointers and 576
  - pointing to 586
  - segment:offset notation 575
  - standard notation for 575
- memory.h (header file)
  - mem.h and 567
- memory models 581, 571-587
  - command-line options 273
  - compact 579
    - default function type 396
  - comparison 582
  - compatibility libraries for 335
  - default
    - overriding 396
  - defined 578
  - function pointers and 399
  - functions
    - default type
    - overriding 392
  - graphics library 614
  - huge 579
    - default function type 396
  - illustrations 579-581
  - initialization modules 335
  - interrupt functions and 393
  - large 579
    - default function type 396
  - libraries 337
  - macros and 519
  - medium 578
    - default function type 396
  - memory apportionment and 579
  - Microsoft C and 567
  - mixing 586
    - function prototypes and 587
  - overlays and 590, 592
  - pointers 576, 583
    - modifiers and 395
  - predefined macros and 519
  - small 578
    - default function type 396
  - startup modules 337
  - tiny 578
    - default function type 396
    - library 337
  - TLINK and 334, 336
- menu bar *See* menus; run bar



- menus *See also* individual menu names (IDE) 23
  - commands *See* individual command names
  - hot keys 24, 25
  - with an ellipsis (...) 31
- Message Tracking
  - toggle 256
- Message window 257
- messages *See also* errors; warnings
  - Compile-time 640
  - DPMI 641
  - TLIB 642
  - MAKE 641
  - run-time 642
  - TLINK (list) 642
- methods *See* member functions
- mice *See* mouse
- Microsoft *See* Microsoft C
- Microsoft C
  - Turbo C++ projects and 561
  - \_cdecl keyword 569
  - CL options
    - TCC options versus 563
  - COMDEFs and 567
  - converting from 561-569
  - environment variables and 562
  - floating-point return values 568
  - header files 567
    - Turbo C++ header files versus 567
  - keywords 568
  - MAKE and 563
  - memory models and 567
  - structures 569
  - TLINK and 565
- mixed modules
  - linking 586
- MK\_FP (run-time library macro) 586
- mode arguments 195
- MODE command (DOS) 21
- models, memory *See* memory models
- modifiable lvalues *See* lvalues
- modifiable objects *See* objects
- modifiers 391
  - function type 396
  - pointers 395, 583
  - table 391
- Modify/New Transfer Item dialog box 259

- Modula-2
  - variant record types 415
- modularity *See* encapsulation
- modules
  - linking mixed 586
  - size limit 582
- modulus operator (%) 61, 425, 431
- monitors *See also* screens
  - dual
    - IDE option (/d) 21
- Monochrome Display Adapter *See* graphics drivers; video adapters
- mouse
  - choosing commands with 24, 31
  - compatibility 3
- mouse buttons
  - right and left 24
- moving text *See* editing, moving text; editing, block operations
  - \_\_MSC macro 566
  - \_\_MSDOS\_\_ macro 521
- multi-source programs *See* projects
- multidimensional arrays 106, *See also* arrays
- multiple assignments 57
- multiple dependents
  - MAKE and 305
- multiple files *See* projects
- multiple inheritance *See* inheritance
- multiple listings
  - command-line compiler options
    - #define 274
    - include and library 294
    - macro definition 274
- multiplication operator (\*) 425, 431
- mx options (memory models) 273

## N

- \n (newline character) 59, 359
- N MAKE option (increase compatibility) 300
- n MAKE option (print commands but don't execute) 300
- n TCC option (.OBJ and .ASM directory) 291
- N TCC option (stack overflow logic) 278
- /n TLINK option (ignore default libraries) 342
- name mangling 376
- name spaces
  - scope and 372

- structures 413
  - C++ 413
- names *See also* identifiers, *See* identifiers qualified 461
- Native command set option 27
- near (keyword) 392, 576, 582
- near functions *See* functions, near
- near pointers *See* pointers, near
- negation
  - logical (!) 425, 430
- negative offsets 573
- nested
  - classes 461
  - comments 281, 351, 352
  - conditional directives 511
  - if statements 76
  - loops 86
  - types 461
- new (keyword)
  - recommended return value 205
- new (operator) 452
  - arrays and 173, 453
  - constructors and 138
  - constructors and destructors and 469
  - destructors and 478
  - dynamic duration objects and 374
  - dynamic objects and 173
  - handling return errors 453
  - initializers and 454
  - malloc function and 205
  - overloading 454, 481
  - prototypes and 453
  - syntax 173
- new lines
  - creating in output 59
- \_new\_handler (for new operator) 453
- newline character (\n) 59
- newline characters (\n)
  - creating in output 359
  - inserting 534
- Next command
  - hot key 25, 26
- next error
  - show 634
- Next Error command
  - hot key 27

- NMAKE
  - MAKE vs. 329
- NMAKE (Microsoft's MAKE utility) 563
- no linkage *See* linkage
- No-Nonsense License Statement 13
- .noautodepend MAKE directive 323
- nocreate, ios data member 548
- .noignore MAKE directive 323
- nondefining declarations *See* declarations, referencing
- nonfatal errors *See* errors
- nonzero digit 356
- noreplace, ios data member 548
- normalized pointers *See* pointers, normalized
- normvideo, conbuf member function 542
- .nosilent MAKE directive 323
- .noswap MAKE directive 323
- not equal to operator (!=) 73, 426, 436
- NOT operator (!) 74, 425, 430
- notation
  - postfix 63
  - prefix 63
- NULL
  - pointers and 400
  - using 400
- null
  - character *See* characters, null
  - directive (#) 503
  - inserting in string 534
  - pointers 400
  - statement 366, 442
  - strings 362
- null character *See* characters, null
- num MAKE command 305
- number of arguments 367
- numbers *See also* constants; data types; floating point; integers
  - base
    - setting for conversion 534
  - BCD *See* BCD
  - binary 50
  - converting *See* conversions
  - decimal
    - conversions 534
  - hexadecimal 356
    - backslash and 358

- constants
  - too large 282
- conversions 534
- displaying 59, 359
- lines *See* lines, numbers
- octal 356
  - backslash and 358
  - constants
    - too large 282
  - conversions 534
  - displaying 59, 359
  - escape sequence 359
- random
  - generating 108
- real *See* floating point
- typecasting 62
- numeric coprocessors *See also* floating point;
  - 80x86 processors
  - autodetecting 599
  - built in 597
  - emulating 276
  - floating-point emulation 598
  - generating code for 276, 277
  - inline instructions 277
  - libraries
    - TLINK and 336
  - registers and 600

## O

- O TCC option (jump optimization) 280
- o TCC option (object files) 286
- /o TLINK option (overlays) 342
- .OBJ files
  - compiling 286
  - converting .BGI files to 617
  - directories 291
  - duplicate symbols in 339
  - line numbers in 279
- object files *See* .OBJ files
- object-oriented programming *See* C++
- objects 369, *See also* C++
  - aliases 449
  - auto
    - destructors and 174
  - automatic 374
    - initializers 389
  - class names and 456

- duration 373
- dynamic 172
  - allocating and deallocating 174
  - destructors and 174
  - new operator and 173
- far
  - class names 284, 584
  - combining into one segment 584
  - declaring 584
  - group names 285
  - option pragma and 584
  - segment names 284
- hidden 373
- initializers 389
- list of declarable 378
- modifiable 393
- pointers 399
  - function pointers vs. 398
- static
  - destructors and 174
  - initializers 389
- temporary 451
- volatile 393
- OBJXREF *See* The online document UTIL.DOC
- oct, ios data member 548
- oct (manipulator) 192, 533, 534
- octal constants *See* constants, octal
- octal digit 356
- octal numbers *See* numbers, octal
- offsets 576
  - component of a pointer 586
- ofstream (class) 553
  - base class 537
  - constructor 537
  - insertion operations 537
- OK button 31
- one-line functions
  - C++ 202
- one's complement *See* operators, 1's complement
- online help *See* help
- OOP *See* C++
- opcodes *See* assembly language
- open
  - filebuf member function 544
  - fstream member function 545
  - fstreambase member function 546

- ifstream member function 547
- ofstream member function 554
- open (function) 195
  - C++ formatting and 193
- Open a File dialog box 633
- Open command 633
  - hot key 25, 26
- open file 631, 633, *See also* files
- open mode *See* files, opening
- open\_mode, ios data member 548
- openprot
  - filebuf data member 543
- operating mode of screen *See* screens, modes
- operator (keyword) 184
  - overloading and 480
- operator functions *See* overloaded operators
- operators 423, 423-426
  - 1's complement 64
  - 1's complement (~) 425, 430
  - addition (+) 425, 432
  - address (&) 425, 429
  - AND (&) 64, 425, 436
    - truth table 436
  - AND (&&) 425, 437
  - arithmetic 60
  - assignment (=) 63, 425, 439
    - compound 439
    - overloading 484
  - associativity *See* associativity
  - binary 425
    - overloading 483
  - bit manipulation
    - using 64
  - bitwise
    - AND (&) 425, 436
      - truth table 436
    - complement (~) 425, 430
    - inclusive OR (|) 425, 437
      - truth table 436
    - truth table 436
    - XOR (^) 425, 436
      - truth table 436
  - C++ 424, *See also* overloaded operators
    - delete 452, *See also* delete (operator), *See* delete (operator)
    - dereference pointers 426, 440
    - get from (>>) *See* overloaded operators
    - new *See* new (operator)
    - pointer to member *See* operators, C++, dereference pointers
    - put to (<<) *See* overloaded operators
    - scope (::) 426, 452
    - scope resolution (::) 136, 138, 158
  - combining 63
  - conditional (? :) 426, 438
  - context and meaning 424
  - decrement (--) 428
  - decrement (--) 63, 425, 428
  - defined operator 511
  - division (/) 425, 431
    - rounding 432
  - equal to (==) 73
  - equality 426, 435
  - evaluation (comma) 426, 440
  - exclusive OR (^) 425, 436
    - truth table 436
  - exclusive OR operator (^) 64
  - function call ( ) 427
  - greater than (>) 73
  - greater than or equal to (>=) 73
  - inclusive OR (|) 425, 437
    - truth table 436
  - increment (++) 63, 425, 427, 428
  - indirection (\*) 425, 429
    - pointers and 402
  - inequality (!=) 426, 436
  - less than (<) 73
  - less than or equal to (<=) 73
  - list 424
  - logical
    - AND (&&) 74, 425, 437
    - negation (!) 425, 430
    - OR (||) 74, 425, 437
  - MAKE 305, 327
  - manipulators *See* manipulators
  - modulus (%) 61, 425, 431
  - multiplication (\*) 425, 431
  - NOT (!) 74
  - not equal to (!=) 73
  - one's complement *See* operators, 1's complement
  - OR (^) 425, 436
    - truth table 436

- OR (|) 64, 425, 437
  - truth table 436
- OR (||) 425, 437
- overloading *See* overloaded operators
- postfix 426
- precedence *See also* precedence
  - defined 60
  - rules 67
  - table 68
- prefix 426
- relational 72, 426, 433
- remainder (%) 61, 425, 431
- scope resolution (::) 205
- selection (. and ->) 426, 427
  - overloading 485
    - structure member access and 411, 427
- shift bits (<< and >>) 64, 425, 433
- sizeof 119, 430
- subtraction (-) 425, 432
- unary 64
  - overloading 482
- unary minus (-) 425, 430
- unary plus (+) 425, 430
- Optimal Fill option 631, 633
- optimizations
  - command-line compiler options 279
  - precompiled headers 637
  - register usage 280
- option pragma 517
  - far objects and 584
- Options
  - backward compatibility 292
  - C++ template generation
    - command-line option 290
- options *See* specific entries (such as command-line compiler, options)
- OR operator 64
  - bitwise inclusive (|) 425, 437
    - truth table 436
  - logical (||) 425, 437
- ordinary member functions *See* member functions, ordinary
- ostream (class) 554
  - derived classes of 537
  - flushing 534
- ostream\_withassign (class) 555
- ostrstream (class) 555
- out, ios data member 548
- out of memory error 571
- out\_waiting, streambuf member function 557
- output
  - C++
    - user-defined types 536
  - directing 613
  - functions 608
- output formatting 548
- overflow
  - conbuf member function 542
  - filebuf member function 544
  - strstreambuf member function 559
- overflows
  - expressions and 423
  - flag 574
- \_\_OVERLAY\_\_ macro 521
- overlays 587-596
  - assembly language routines and 593
  - BP register and 593
  - buffers
    - default size 592
  - cautions 593
  - command-line options (-Yo) 590
  - debugging 593
  - designing programs for 592
  - extended and expanded memory and 594
  - generating 279
  - how they work 588
  - large programs 587
  - memory map 589
  - memory models and 590, 592
  - predefined macro 521
  - routines, calling via function pointers 593
  - TLINK and 342
- overloaded constructors *See* constructors, overloaded
- overloaded functions 132, 181
  - defined 457
  - templates and 491
- overloaded operators 184, 422, 423, 479-485
  - >> (get from) 144, 196, 535
    - complex numbers and 601
  - << (put to) 144, 196, 200, 531
    - complex numbers and 601
  - addition (+) 184
  - assignment (=) 484

- binary 483
- brackets 484
- class for 190
- complex numbers and 601
- creating 458
- defined 200, 457
- delete 481
- functions and 422
- inheritance and 481
- new 454, 481
- operator functions and 480
- operator keyword and 480
- parentheses 484
- precedence and 422
- restrictions 186
- selection (->) 485
- unary 482
- overview
  - IDE menus and options 36
- \_ovrbuffer (global variable) 589, 592

## P

- P
  - TCC option (C++ and C compilation) 286
  - TCC option (compile C++) 142
- p command-line option
  - Pascal calling conventions
    - main function and 527
- p command-line option (Pascal calling convention) 394, 519, 521
  - cdecl and 395
- /p IDE option (EGA palette) *See* Enhanced Graphics Adapter (EGA), palette, IDE option (/p)
- p MAKE option (display rules) 300
- p MAKE option (ignore BUILTINS.MAK) 300
- p TCC option (Pascal conventions) 278
- padding (C++) 534
- pages
  - active
    - defined 620
    - setting 619
  - buffers 620
  - visual
    - defined 620
    - setting 619
- painting *See* graphics, fill, operations
- pair matching 631
- palettes *See* graphics, palettes
- paragraphs *See* memory, paragraphs
- parameter-passing sequence, Pascal 278
- parameterized manipulators 192
- parameterized types 490
- parameters *See also* arguments
  - default
    - constructors 471
  - ellipsis and 367
  - empty lists 383
  - fixed 406
  - formal 408
    - C++ 408
    - scope 408
  - function calls and 408
  - passing
    - C 392, 394
    - Pascal 392, 394
  - reference 203
  - variable 406
- parentheses 365
  - as function call operators 427
  - macros and 365
  - nested
    - macros and 507
    - overloading 484
- parity flag 574
- parsing 350
- Pascal
  - calling conventions
    - compiler option (-p) 394
    - compiling main with 527
  - functions 394
  - identifiers 394
    - case sensitivity and 355
    - identifiers of type 278
  - parameter-passing sequence 278, 392
  - variant record types 415
  - \_\_PASCAL\_\_ macro 521
  - pascal (keyword) 392, 394
    - function modifiers and 396
    - preserving case while using 394
  - \_pascal (keyword)
    - fortran keyword and 568

- pass-by-address, pass-by-value, and pass-by-var
  - See parameters; referencing and dereferencing
- Paste command
  - hot key 26
- paste from Clipboard 630, 632
- pasting See editing, copy and paste
- .path directive (MAKE) 323
- paths
  - .BGI files 91
- pbase, streambuf member function 557
- pbump, streambuf member function 557
- pcount, ostrstream member function 556
- peek, istream member function 552
- period as an operator See operators, selection (. and ->)
- PF87.LIB 336
- phrase structure grammar See elements
- place marker
  - find 631, 633
  - set 631, 634
- PLANETS.CPP (sample program) 91
- plasma displays
  - installing Turbo C++ for 17
- PLOTEMP.C (sample program) 220
- pointer-to-member operators See operators, C++, dereference pointers
- pointers 113, 398, See also referencing and dereferencing
  - advancing 402
  - arithmetic 116, 402, 577
  - assignments 400
  - base class
    - destructors and 478
  - C++ 449
    - reference declarations 403
  - to class members 426, 440
  - comparing 434, 436, 443, 577
    - while loops 445
  - const 391
  - constants and 401
  - conversions See conversions
  - declarations 400
  - declarator (\*) 367, 402
  - declaring 113
  - default data 581
  - delete operator and 477
  - dereference 426, 440
  - far 392
    - adding values to 577
    - comparing 576
    - memory model size and 586
    - registers and 576
  - far memory model and 576
  - fast huge 277
  - format specifiers and 233
  - function 399
    - C++ 399
      - modifying 396
      - object pointers vs. 398
      - void 399
  - functions and 119
  - generic 383, 400
  - huge 277, 392, 577
    - comparing
      - != operator 578
      - == operator 578
    - overhead of 578
  - huge memory model and 576
  - initializing 400
  - keywords for 392
  - manipulating 576
  - memory models and 576, 583
  - to memory addresses 586
  - modifiers 395, 582
  - near 392, See also segments, pointers
    - memory model size and 586
    - registers and 576
  - near memory model and 576
  - normalized 577
  - null 400
  - NULL and 400
  - operator (->)
    - overloading 485
    - structure and union access 411, 426, 427
  - overlays and 593
  - pointers to 399
  - range 52, 363
  - reassigning 400
  - referencing and dereferencing 429
  - segment 392, 582, 583
  - to self See this (keyword)
  - stack 573
  - strings and 115

- structure members as 411
- structures and 116
- suspicious conversion 282
- typecasting 403
- virtual table
  - 32-bit 288
  - void 400
- polymorphism 132
  - defined 126
  - example 214
  - virtual functions and 132, 214
- pop-up menus *See* menus
- portability warnings 283
- portable code
  - bit fields and 415
- positive offsets 573
- postdecrement operator (--) 425, 428
- postfix operator ++
  - overload 483
- postfix operator —
  - overload 483
- postfix operators 426
- postincrement operator (++) 425, 427
- pptr, streambuf member function 557
- pragma directives
  - templates and 498
- #pragma exit
  - destructors and 477
- #pragma hdrfile 636, 637
- #pragma hdrstop 636, 638
- #pragma directives 515
  - argsused 515
  - exit 515
  - hdrfile 516
  - hdrstop 517
  - inline 517
  - option pragma 517
    - far objects and 584
  - saveregs 518
  - startup 515
  - warn 519
- precedence 422, *See also* associativity
  - command-line compiler options 267, 272
    - response files and 271
  - controlling 365
  - defined 60
  - expressions 420
    - floating point 423
    - integer 423
  - overloading and operators 422
    - rules 67
    - table 68
  - precious directive (MAKE) 323
- precision, ios member function 549
- precompiled headers 635-638
  - command-line options 286
  - controlling 636
  - drawbacks 636
  - how they work 635
  - inline member functions and 636
  - optimizing use of 637
  - rules for 637
  - storage file 516
- predecrement operator (--) 425, 428
- predefined macros *See* macros, predefined
- Preferences dialog box 633
- prefix operators 426
- preincrement operator (++) 425, 428
- preprocessor directives *See also* directives, *See* directives
  - introduced 45
- previous error
  - show 634
- Previous Error command
  - hot key 27
- Previous Topic command
  - hot key 26
- printable characters 59
- printbase characters 59
- printers
  - streams 122
- private (keyword) 140
  - classes and 141
  - data members and member functions 463
  - derived classes and 464
  - unions and 417
- PRJ2MAK *See* The online document UTIL.DOC
- .PRJ files *See* projects
- PRJCFG *See* The online document UTIL.DOC
- PRJCNVT *See* The online document UTIL.DOC
- procedures *See* functions
- producer (streams) 529
- profilers 593



- program manager (MAKE) *See* MAKE (program manager)
- Program Reset command 250
  - hot key 27
- Programmer's Platform *See* Integrated Development Environment
- Programmer's Workbench
  - integrated environment and 561
- programming
  - object-oriented 125
- programs
  - basic operations 44
  - C++ *See* C++
  - creating 349
  - designing 87, 241
  - multi-source *See* projects
  - multifunction 91
  - multiple
    - debugging 241
  - performance
    - improving 389
  - prototypes and 220, 222
  - robust 243
  - size
    - reducing 389
  - solar system 91
  - swap 120
  - testing 242
  - transfer
    - list 259
  - very large
    - overlying 587
- project files 33
  - contents of 34
- Project Manager *See* projects
- Project Notes window 263
- projects
  - automatic dependency checking and 257
  - building 251
  - changing 35
  - classes and 148
  - default 35
  - desktop files and 35, 33-35
  - directories 254
  - directory 34
  - error tracking 255, 256
  - files
    - adding 253
    - deleting 253
    - graphics library listed in 614
    - include 253
    - information 258
    - list 253
    - options 253
    - out of date 257
  - IDE configuration files and 34
  - include files 253
  - information in 251
  - libraries and
    - overriding 260
  - loading and opening 34
  - makes and 257
  - making
    - hot key for 256
  - Microsoft C and 561
  - naming 252
  - new 253
  - notes 263
  - saving 254
  - translators *See also* Transfer
    - default 258
    - example 259
    - multiple 258
    - specifying 259
- promotions *See* conversions
- protected (keyword) 141, 212
  - data members and member functions 463
  - derived classes and 464
  - unions and 417
- protected mode 14
  - command-line compiler 266
  - DPMIMEM variable 15
  - DPMIRES utility 16
- prototypes 88, 405-407
  - arguments and
    - matching number of 409
  - C++ 404
  - ellipsis and 406, 409
  - examples 405, 406
  - function calls and 408
  - function definitions and
    - not matching 409
  - header files and 406

- introduced 48
- libraries and 409
- mixing modules and 587
- new operator and 453
- scope *See* scope
- pseudovariables
  - register 354
    - using as identifiers 281
- public (keyword) 141, 211
  - data members and member functions 462
  - derived classes and 464
  - unions and 417
- Public option
  - C++ Virtual Tables
    - command-line option 288
- pull-down menus *See* menus
- punctuators 365, 365-368
- pure specifier 381
- pure virtual functions
  - specifying 486
- put
  - ostream member function 554
- put (function) 192
- put to operator (<<) *See* overloaded operators, << (put to)
- putback, istream member function 552
- putch (function)
  - characters and 71
- puts (function)
  - strings and 72

## Q

- Q TCC options (expanded memory) 287
- qualified names 461
- question mark
  - colon conditional operator 426, 438
  - displaying 59, 359
- Quit
  - command (IDE) 23
- quotes
  - displaying 59, 359

## R

- \r (carriage return character) 59, 359
- /r IDE option (RAM disk) *See* RAM disk, IDE option (/r) and

- r TCC option (register variables) 280
- radio buttons 31
- RAM
  - Turbo C++'s use of 571
- RAM disk
  - IDE option (/r) and 22
- random (macro) 108
- random numbers *See* numbers, random
- range errors, arrays 106
- ranges
  - floating-point constants 361
- rd option (register variables) 280
- rdbuf
  - constream member function 543
  - fstream member function 545
  - fstreambase member function 546
  - ifstream member function 547
  - ios member function 550
  - ofstream member function 554
  - strstreambase member function 559
- rdstate, ios member function 550
- rdstate (member function) 195
- read, istream member function 552
- read block 630
- README 17
- README.DOC 14
- real numbers *See* floating point
- rebuilding libraries 279
- records *See* structures
- recursive functions
  - memory models and 585
- redirection
  - operators
    - MAKE 305
- redo 631
- Redo command
  - hot key 26
- reference declarations 403
  - position of & 383, 450
- reference parameters 203
- reference types 165
- references
  - forward 370
- referencing and dereferencing 203, 429, *See also*
  - pointers
    - asterisk and 367
    - C++ 449

- functions 450
  - simple 450
- pointers 426, 440
- referencing declarations *See* declarations
- register (keyword) 389
  - class members and 457
  - external declarations and 380
  - formal parameters and 408
  - local duration and 374
  - using 102
- registers
  - AX 573
  - base point 573
  - BP 573
    - overlays and 593
  - BX 573
  - CS 574, 576
  - CX 573
  - DI 573
  - DS 574, 576
    - \_loadds and 396
  - DX 573
  - ES 574
  - flags 572, 573
  - hardware
    - bit fields and 415
  - iAPx86 572-574
  - index 573
  - interrupts and 393
  - IP (instruction pointer) 572
  - LOOP and string instruction 573
  - math operations 573
  - numeric coprocessors and 600
  - pseudovariables 354
    - using as identifiers 281
  - saving with huge functions 518
  - segment 573, 574
  - SI 573
  - SP 573
  - special-purpose 573
  - SS 574
  - values
    - preserving 396
  - variable declarations and 389
  - variables 280, 389
    - suppressed 280
    - toggle 280

- relational operators *See* operators, relational
- relative position
  - C++ and 197
- remainder operator (%) 61, 425, 431
- Remove Messages command 257
- resetiosflags (manipulator) 192, 533, 534
- resize corner 28
- resolution *See* screens, resolution
- response files
  - defined 271, 333
  - option precedence 271
  - TLINK and 333
- return
  - statements
    - functions and 447
    - maximum number 94
  - types 405
  - values 48
    - functions 90
- right, ios data member 548
- Ritchie, Dennis *See* Kernighan and Ritchie (K&R)
- robust programs 243
- rounding
  - banker's 604
  - direction
    - division 432
  - errors 602
- routines, assembly language *See* assembly language
- run bar 224
- Run command
  - hot key 27
- rvalues 371, *See also* lvalues

## S

- /s IDE option (thrash control) *See* thrash control, IDE (/s) and
- s MAKE option (don't print commands) 300
- S MAKE option (swap MAKE out of memory) 300
- S TCC option (produce .ASM but don't assemble) 286
- sample programs
  - INTRO2.CPP 48
  - PLANETS.CPP 91
  - PLOTEMP.C 220

- Save command
  - hot key 25, 26
- save file 631, 633
- saveregs pragma 518
- \_saveregs (keyword) 392, 396
  - uses for 396
- sbumpc, streambuf member function 557
- scalar data types *See* data types
- scaling factor
  - graphics 619
- scanf (function)
  - >> operator and 535
- scientific, ios data member 548
- scope 99, 371-373, *See also* variables; visibility
  - block 372
  - block statements and 441
  - C++ 373, 488-490
    - data members 147
    - functions 136
    - hiding 488
    - operator (::) 426, 452
    - rules 489
  - classes 372
    - names 456
  - enclosing 488
  - enumerations 372
    - C++ 419
  - extern keyword and 100
  - file 372
    - static storage class specifier and 375
  - formal parameters 408
  - function 372
    - prototype 372
  - function variables 90
  - global 372
  - global declarations and 94
  - global variables and 100
  - goto and 372
  - identifiers and 355
  - local
    - duration and 374
  - members 460-463
  - name spaces and 372
  - pointers 400
  - resolution operator (::) 136, 138, 158, 205
  - side effects and 100
  - storage class specifiers and 389-391
  - structures 372
  - unions 372
  - variables 372
  - visibility and 373
- screens *See also* graphics; text; windows
  - aspect ratio 619
  - attributes, controlling 609
  - cells
    - attributes 612
      - blinking 613
    - characters in 605
    - colors 612
  - clearing 619
  - colors 612, 622
  - coordinates 607
    - starting positions 606
  - cursor
    - changing 610
    - manipulating 608
  - LCD
    - IDE option *See* LCD displays, IDE option (/I)
    - installing Turbo C++ for 17
  - modes
    - controlling 609
    - defining 605
    - graphics 606, 614, 616
    - selecting 616
    - text 605, 611, 616
  - plasma
    - installing Turbo C++ for 17
  - resolution 606, *See also* graphics, pixels
  - streams 122
  - using two *See* monitors, dual
  - viewports *See* graphics
- scroll bars 28, 29
- Search Again command
  - hot key 26
- search and replace *See* searching
- search for text 631
- search.h (header file) 567
- searches
  - #include directive algorithm 509
- searching
  - include files 294
  - libraries 294
- seek\_dir, ios data member 548

- seekg, istream member function 553
- seekoff
  - filebuf member function 544
  - streambuf member function 557
  - strstreambuf member function 560
- seekp, ostream member function 555
- seekpos, streambuf member function 557
- \_seg (keyword) 392, 582, 583
  - \_segment keyword and 568
- segment-naming control
  - command-line compiler options 284
- segment:offset address notation 575
  - making far pointers from 586
- \_segment (keyword) 568
- segmented memory architecture 574
- segments 575, 578
  - component of a pointer 586
  - controlling 284
  - map of
    - ACBP field and 341
    - TLINK and 341
  - memory 574
  - pointers 392, 582, 583
  - registers 573, 574
  - uninitialized
    - TLINK and 340
- \_segname (keyword) 568
- selecting text 631
- selection
  - operators *See* operators, selection
  - statements *See* if statements; switch statements
- self *See* this (keyword)
- \_self (keyword) 568
- semicolon (for empty loops) 83
- semicolons 366, 442
- sequence
  - classes *See* classes, sequence
- setb, streambuf member function 557
- setbase (manipulator) 192, 533, 534
- setbkcolor (function)
  - CGA vs. EGA 625
- setbuf
  - filebuf member function 544
  - fstreambase member function 546
  - streambuf member function 557
  - strstreambuf member function 560
- setcursortype, conbuf member function 542
- \_setcursortype (function) 610
- setf
  - constants used with 547
- setf (function) 534
- setf, ios member function 550
- setfill (manipulator) 192, 533, 534
- setg, streambuf member function 557
- setiosflags (manipulator) 192, 533, 534
- set\_new\_handler (for new operator) 453
- setp, streambuf member function 558
- setprecision (manipulator) 192, 533, 534
- setstate, ios member function 550
- setw (manipulator) 192, 533, 534
- sgetc, streambuf member function 558
- sgetn, streambuf member function 558
- shift bits operators (<< and >>) 64, 425, 433
- short integers *See* integers, short
- shortcuts *See* hot keys
- showbase, ios data member 548
- showpoint, ios data member 548
- showpos, ios data member 548
- SI register 573
- side effects 100
  - macro calls and 508
- sign 356
  - defined 53
  - extending 359
  - conversions and 386
  - flag 574
- signature, function 162
- signed (keyword) 384
- .silent MAKE directive 323
- single quote character
  - characters and 72
  - displaying 59, 359
- sink (streams) 529
- size\_t (data type) 431, 481, 482
- sizeof (operator) 119, 430
  - arrays and 431
  - classes and 431
  - example 371
  - function-type expressions and 431
  - functions and 431
  - preprocessor directives and 431
  - unions and 416
- skipws, ios data member 548

- `__SMALL__` macro 519
- small code *See* memory models
- Smalltalk
  - C++ vs. 127
- Smart option
  - C++ Virtual Tables
    - command-line option 287, 289
  - `snnextc`, `streambuf` member function 558
  - software *See* programs
  - software license agreement 13
  - software requirements to run Turbo C++ 3
  - solar system
    - program 91
  - sounds
    - beep 59
  - source (streams) 529
  - source code 349
  - source files
    - .ASM
      - command-line compiler and 265
      - multiple *See* projects
  - source-level debugger *See* Turbo Debugger
  - Source Tracking options 256
  - SP register 573
  - special-purpose registers (iAPx86) 573
  - specifiers *See* type specifiers
  - speed
    - optimization 279
  - splicing lines 350, 363
  - `sputbackc`, `streambuf` member function 558
  - `sputc`, `streambuf` member function 558
  - `sputn`, `streambuf` member function 558
  - SS register 574
  - `_ss` (keyword) 392, 582
  - stack
    - overflow 278
    - pointers 573
    - segment 574
    - standard frame
      - generating 277
  - standalone utilities *See* MAKE (program manager); TLIB (librarian); TLINK (linker)
  - standard conversions *See* conversions
  - standard library files *See* libraries
  - standard stack frame
    - generating 277
  - start-up and exit
    - command-line compiler 266
    - IDE 20
  - startup code (TLINK) 335
  - startup modules for memory models 337
  - startup pragma 515
  - state, ios data member 547
  - state queries 626-628
  - statements 441-448, *See also* break statements;
    - if statements; switch statements, *See also* individual statement names
    - assembly language 441
    - block 441
      - marking start and end 365
    - default 444
    - defined 46
    - do while *See* loops, do while
    - expression 366, 442
    - for *See* loops, for
    - if *See* if statements
    - iteration *See* loops
    - jump *See* break statements; continue statements; goto statements; return statements
    - labeled 442
    - null 442
    - syntax 441
    - while *See* loops, while
  - static
    - duration 373
    - functions 375
    - members *See* data members, static; member functions, static
    - objects *See* objects, static
    - variables *See* variables, static
  - static (keyword) 101, 390
    - linkage and 375
  - static binding *See* C++, binding, early
  - `_status87` (function)
    - floating point exceptions and 600
  - status line 30
  - stdarg.h (header file)
    - user-defined functions and 406
  - `__STDC__` macro 521
    - `#define` and `#undef` directives and 506
  - stderr, functions of *See* streams
  - stdin, functions of *See* streams

- stdio, ios data member **548**
- stdio.h (header file)
  - stream.h vs. **200**
- stdout, functions of *See* streams
- Step Over command
  - debugging and **224**
  - hot key **25, 27**
- storage class
  - identifiers and **371**
  - specifiers **389**
    - functions and **376**
    - linkage and **375**
    - static
      - file scope and **375**
- stossc, streambuf member function **558**
- str
  - ostrstream member function **556**
  - strstream member function **560**
  - strstreambuf member function **560**
- strcmp (function)
  - example **86**
- stream.h
  - stdio.h vs. **200**
- streambuf **189**
- streambuf (class) **530, 556**
  - derived classes of **530**
- streams
  - binary
    - opening **123**
  - C++ **188-196**
  - cin, cout, and cerr **189**
  - classes and **529**
  - clearing **534**
  - cout
    - flushing **201**
  - data types **532**
  - defined **144, 529**
  - errors **537**
  - file buffers **195**
  - file class **529**
  - flushing **534**
  - formatted I/O **530**
  - library **188**
  - manipulators and *See* manipulators
  - memory buffer class **529, 530**
  - open function and **195**
  - output **531**
  - string class **529**
    - tied **550**
  - default **122**
  - defined **122**
  - functions of **122**
  - keyboards **122**
  - opening **122, 124**
  - preopened **122**
  - printers **122**
  - screens **122**
  - standard
    - table **122**
  - text
    - opening **123**
    - using
      - example program **123**
- strings
  - arrays and **108**
  - characters and **72**
  - clipping **621**
  - concatenating **184, 362**
  - continuing across line boundaries **363**
  - converting arguments to **508**
  - defined **71**
  - displaying **71, 72**
  - empty **362**
  - inserting terminal null into **534**
  - inspecting **230**
  - instructions
    - registers **573**
  - literal **71, 350, 362**
    - merging **275**
  - memory and **71**
  - null **362**
  - null character and **108**
  - pointers and **115**
  - puts and **72**
  - scanning
    - while loops and **445**
  - streams
    - C++ **538**
    - streams and **529**
- strlen (function)
  - example **86**
- stroked fonts *See* fonts
- strstrea.h (header file)
- string streams and **538**

- stringstream (class) 560
- stringstreambase (class) 558
- stringstreambuf (class) 559
- struct
  - default access 103
  - keyword
    - introduction 103
- struct (keyword) 410, *See also* structures
  - C++ and 411, 456
- structures 409-415
  - access
    - C++ 464
    - classes vs. 141
  - ANSI violations 282
  - bit fields *See* bit fields
  - Turbo C++ versus Microsoft C 569
  - C++ 455, *See also* classes
    - C vs. 456
  - classes vs. 129
  - complex 601
  - declaring 410
  - defined 111
  - functions and 411
  - incomplete declarations of 414
  - indeterminate arrays and 404
  - initializing 387
  - inspecting 230
  - member functions and 411
  - members
    - access 411, 427, 463
    - as pointers 411
    - C++ 411
    - comparing 434
    - declaring 410
    - defined 111
    - names 413
  - memory allocation 413
  - name spaces 372, 413
  - pointers and 116
  - tags 410
    - typedefs and 410
  - typedefs and 410
  - undefined 282
  - unions vs. 415
  - untagged 410
    - typedefs and 410
  - within structures 411
- word alignment
  - memory and 413
  - zero length 282
- subscripting operator *See* brackets
- subscripts for arrays 365, 426
  - overloading 484
- subtraction operator (-) 425, 432
- suppressing load operations 280
- swap (example) 119
- .swap MAKE directive 323
- switch statements 77-79, 444
  - break *See* break statements
  - case statement and
    - duplicate case constants 444
    - default label and 444
- switches *See* command-line compiler, options
- .SYM files 635, 636
  - default names 636
  - disk space and 637
  - smaller than expected 637
- symbolic
  - constants *See* macros
  - debugger *See* Turbo Debugger
- symbols
  - action *See* TLIB
  - duplicate
    - warning (TLINK) 339
- sync, filebuf member function 544
- sync, stringstreambuf member function 560
- sync\_with\_stdio, ios member function 550
- syntax
  - assembly language statements 441
  - classes 455
  - declarations 377, 378
  - declarator 398
  - delete operator 174
  - directives 502
  - errors
    - project files 255, 256
  - expressions 421
  - IDE command line 20
  - inline functions 180
  - MAKE 299
  - manipulators 534
  - new operator 173
  - statements 441
  - templates 490



- TLINK 332
- Syntax Highlighting 42
- Syntax highlighting 36
- syntax highlighting
  - IDE 37
- system
  - resources 121
- system control, graphics 615
- System menu (=) 23
- system requirements 3

## T

- \t (horizontal tab character) 59, 359
- T- TCC option (remove assembler options) 287
- /t TLINK option
  - default to .COM 343
  - generate .COM file 335, 343
- 'this' pointer in 'pascal' member functions 292
- tab (manipulator) 197
- tab characters 59
- Tab mode 634
- tables, virtual *See* virtual tables
- Tabs mode 631
- tags
  - enumerations 418
  - name spaces 419
  - structure *See* structures, tags
- TASM *See* Turbo Assembler
- taxonomy
  - defined 131
  - types 382
- TC and TCC *See* Turbo C++; command-line compiler; integrated environment
- TC.EXE *See* integrated environment
- TCC.EXE *See* command-line compiler
- TCCONFIG.TC *See* configuration files, IDE
- TCDEF.DPR files 35
- TCDEF.DSK files 35
- TCDEF.SYM 286, 516, 635, 636, *See also* .SYM files
- \_\_TCPLUSPLUS\_\_ macro 521
- /Td LINK options (target file) 343
- TDSTRIP
  - TLINK and 344
- technical support 7
- tellg, istream member function 553
- tellp, ostream member function 555
- TEML *See* The online document UTIL.DOC
- temperature-plotting program 220
- template function 492
- templates 490, *See also* syntax
  - angle brackets 495
  - arguments 494
  - class 493
  - compiler switches 497
  - eliminating pointers 496
  - function 490
    - implicit and explicit 492
    - instantiation 492
    - overriding 492
  - generation 290
  - macro 521
  - type-safe
    - generic lists 495
    - using switches 498
- \_\_TEMPLATES\_\_ macro 521
- temporary objects 451
- tentative
  - definitions 377
- testing software 242
- text *See also* editing
  - blocks *See also* editing, block operations, *See* editing, block operations
    - moving in and out of memory 608
  - capturing to memory 608
  - colors 612
  - entering
    - in dialog boxes 32
  - in graphics mode 620
  - information on current settings 628
  - justifying 621
  - manipulation
    - functions 608
    - onscreen 608
    - output and 608
  - mode types 611
  - output
    - header file 607
  - reading and writing 608
  - scrolling 608
  - strings
    - clipping 621
    - size 621

- writing to screen 608
- text files *See also* editing
  - opening 123
- textattr, conbuf member function 542
- textbackground, conbuf member function 542
- textcolor, conbuf member function 542
- textmode
  - conbuf member function 542
  - constream member function 543
- textmode (function) 607
- THELP *See* The online document UTIL.DOC
- 32-bit code 338
- this (keyword) 186
  - nonstatic member functions and 457
  - static member functions and 459
- thrash control
  - IDE (/s) and 22
- threshold size
  - far global variables
    - setting 276
- tie, ios member function 550
- tied streams 550
- Tile command
  - hot key 26
- time *See also* date
  - macro 521
- \_\_TIME\_\_ macro 521
  - #define and #undef directives and 506
- \_\_TINY\_\_ macro 519
- tiny memory model *See* memory models
- title bars 28
- TLIB (librarian)
  - classes and 148
- TLINK (linker)
  - ACBP field and 341
  - assembler code and 338
  - .COM files and 343, 344
  - command-line compiler and 337
  - debugging information 344
  - executable file map generated by 340
  - floating-point libraries 336
  - graphics library and 336
  - initialization modules 335
  - invoking 331
  - LIB environment variable and 562
  - libraries 336
  - LINK (Microsoft) versus 566
  - memory models and 334
  - Microsoft C and 565
  - numeric coprocessor libraries 336
  - options 338
    - case sensitivity (/c) 338
    - .COM files (/t) 335, 343
    - .COM files (/Td) 343
    - debugging information (/v) 344
    - duplicate symbols warning (/d) 339
    - executable files (/Td) 343
    - expanded memory (/ye) 344
    - extended dictionary (/e) 339
    - extended memory (/yx) 345
    - file extension 333, 335
    - libraries, ignoring (/n) 342
    - line numbers (/l) 340
    - map files (/m)
      - debugging 341
      - public symbols in 341
      - segments in 341
    - /n
      - (ignore default libraries) 342
    - overlays (/o) 342
    - /s
      - (map files) 340
    - source code line numbers (/l) 340
    - target files 343
    - /Td
      - (target files) 343
    - 32-bit assembler code and (/3) 338
    - tiny model .COM files (/t) 335
    - uninitialized trailing segments (/i) 340
    - /v
      - (debugging information) 344
    - /x
      - (map files) 340
    - /ye
      - (expanded memory) 344
    - /yx
      - (extended memory) 345
  - response files 333
    - example 334
  - starting 331
  - startup code 335
  - syntax 332
  - target file option (/Td) 343
  - using directly 566

- TLINK (linker)
  - segment limit 691
- TLINK (linker)
  - warnings
    - defined 642
    - list 642
- Toggle Breakpoint command
  - hot key 27
- tokens
  - continuing long lines of 508
  - kinds of 352
  - parsing 350
  - pasting 351, 507
  - replacement 503
  - replacing and merging 368
- Topic Search command
  - hot key 26
- Topic search in Help 631
- Trace Into command
  - debugging and 225
  - hot key 25, 27
- trailing segments, uninitialized 340
- Transfer *See also* projects, translators
  - command 23
  - dialog box
  - projects and 259
- transfer programs
  - list 259
- translation units 375
- translators *See* projects, translators
- trap flag 574
- TRIGRAPH *See* The online document UTIL.DOC
- trunc, ios data member 548
- truth table
  - bitwise operators 436
- Tstring TCC option (pass string to assembler) 287
- Turbo Assembler
  - Turbo C++ command-line compiler and 270
  - command-line compiler and 265
  - default 286
  - invoking 270
  - TLINK and 338
- Turbo C++ *See also* C++; C language; keywords C and 278
  - converting to from Microsoft C 561-569

- exiting 14
  - implementation data 3
  - installing 14-17
  - project files and 258
  - quitting 23
  - starting 14
  - starting up 20
- Turbo Debugger
  - described 279
- Turbo Editor Macro Language compiler *See* The online document UTIL.DOC
- Turbo Profiler 593
- \_\_TURBOC\_\_ macro 522
- TURBOC.CFG 271
- tutorials
  - C++ 199-215
  - debugging 217-250
- type-safe
  - lists 496
- type specifiers
  - elaborated 456
  - pure 381
- type taxonomy 382
- typecasting
  - defined 62
  - pointers 403
- typed constants *See* constants
- typedef (keyword) 109, 390
  - name space 372
  - structure tags and 410
  - structures and 410
- typedefs
  - untagged structures and 410
- typefaces used in these books 6
- types *See* data types
- typographic conventions 6

## U

- U MAKE option (undefine) 300
- U TCC option (undefine) 274, 505
- u TCC option (underscores) 278
- UINT\_MAX (constant) 433
- ULONG\_MAX (constant) 433
- unary operators 425, *See* operators, unary
  - minus (-) 425, 430
  - plus (+) 425, 430
  - syntax 428

- unbuffered, streambuf member function **558**
- unconditional breakpoints *See* watch expressions
- #undef directive **504**
  - global identifiers and **506**
- !undef MAKE directive **328**
- underbars *See* underscores
- underflow
  - filebuf member function **544**
  - strstreambuf member function **560**
- underscores **278**
  - generating **392**
  - generating automatically **278**
  - ignoring **392**
- undo **631**
- Undo command
  - hot key **26**
- unindent
  - block **630**
  - mode **631, 634**
- uninitialized data segment *See* data segment
- union (keyword)
  - C++ **456**
- unions **415**
  - anonymous
    - member functions and **416**
  - base classes and **464**
  - bit fields and *See* bit fields
  - C++ **417, 455**
    - C vs. **456**
  - classes and **417**
  - constructors and destructors and **469**
  - declarations **417**
  - initialization **387, 417**
  - inspecting **230**
  - members
    - access **427, 463**
    - name space **372**
    - sizeof and **416**
    - structures vs. **415**
- unitbuf, ios data member **548**
- units, translation *See* translation units
- UNIX
  - keywords
    - using **281**
  - porting Turbo C++ files to **281**
- unsetf (function) **534**

- unsetf, ios member function **550**
- unsigned (keyword) **384**
- untagged structures *See* structures, untagged
- uppercase, ios data member **548**
- user-defined formatting flags **551**
- User Screen
  - command **224**
  - debugging and **224**
  - hot key **26**
- user-specified library files **291**
- UTIL.DOC **622**
- utilities *See also* The online document  
UTIL.DOC

## V

- v
  - option (debugging information) **340**
  - TCC option (debugging information) **278**
- \v (vertical tab character) **59, 359**
- V and -Vn TCC options (C++ virtual tables) **287**
- /v TLINK option (debugging information) **344**
- Va TCC option (class argument compatibility) **292**
- Vb TCC option (virtual base class pointer compatibility) **292**
- Vc TCC option (derived class with pointer to inherited virtual base class member function) **292**
- Vm TCC options (C++ member pointers) **289**
- Vp TCC option ('this' pointer in 'pascal' member functions compatibility) **292**
- Vt TCC option (virtual table pointers) **293**
- Vv TCC option (pointers to virtual base class members) **293**
- value, passing by *See* parameters
- values
  - comparing **433**
- var, passing by *See* parameters
- varargs.h (header file) **567**
- variable argument list **278**
- variable number of arguments **367**
- variables *See also* scope
  - assignment statements **56**
  - automatic **101, See also** auto (keyword)
  - word-aligning **275**
- communal **275**

- declaring 55, 388
- declaring anywhere (C++) 202
- default values 56
- evaluating 232
- external 389
- global *See* global variables
  - far 276
- initializing 55, 388
- instances and objects and 134
- internal linkage 390
- local 99
- name space 372
- naming 57, 58, 242
- pointers and 112
- pseudo *See* pseudovariables
- register 280, *See also* registers, variables
- sharing 94
- signed and unsigned 53
- static 101
- volatile 393
- watching 237
- variant record types *See* unions
- vectors, interrupt *See* interrupts
- vertical tab 359
- vertical tab character (\v) 59
- vi option (C++ inline functions) 279
- video adapters *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics drivers; Video Graphics Array Adapter (VGA)
  - graphics, compatible with Turbo C++ 615
  - modes 605
  - output
    - directing 613
    - using 605-628
- Video Graphics Array Adapter (VGA) *See also* graphics drivers; video adapters
  - color control 625
- viewports *See* graphics
- virtual
  - base classes *See* classes, base, virtual
  - destructors *See* destructors, virtual
  - functions *See* member functions, virtual
- virtual (keyword) 163
  - constructors and destructors and 468
  - functions and 485
- virtual base class
  - hidden pointer to 292
  - pointers to 293
- virtual functions *See also* member functions, virtual
  - hidden members in derived classes with pointers to 292
  - specifying pure 486
- virtual table pointers
  - compatibility 293
- virtual tables
  - 32-bit pointers and 288
  - controlling 287
  - storing in the code segment 288
- visibility 373, *See also* scope
  - C++ 373
  - pointers 400
  - scope and 373
- visual page
  - defined 620
  - setting 619
- void (keyword) 383
  - defined 88
  - function pointers and 399
  - functions and 406
  - interrupt functions and 393
  - pointers 400
  - typecasting expressions and 383
- volatile (keyword) 391, 393
  - formal parameters and 408
- VROOMM 588, *See also* overlays

## W

- W MAKE option (save options) 300
- wxxx TCC options (warnings) 282
- warn pragma 519
- warning beep 59
- warnings *See also* errors
  - C++ 283
  - command-line options 282-284
  - Compile-time 640
  - defined 640
  - disabling 515
  - enabling and disabling 282
  - frequent errors 283
  - TLIB 642
  - messages 5

- options 282-284
- overriding 519
- portability 283
- pragma warn and 519
- TLINK
  - defined 642
  - TLINK (list) 642
- watch expressions 237, *See also* debugging
- Watch menu 237
- watches
  - deleting 239
  - editing 239
  - setting 237
- wchar\_t (wide character constants) 360
  - arrays and 388
- wherex, conbuf member function 542
- wherey, conbuf member function 542
- while loops *See* loops, while
- whitespace 350
  - comments and 352
  - comments as 350
  - extracting 534
- wide character constants (wchar\_t) 360
- width, ios member function 550
- WILDARGS.OBJ 525
- wildcards
  - expansion 525
    - by default 526
    - from the IDE 526
  - MAKE and 309
- window
  - conbuf member function 542
  - constream member function 543
- window (function)
  - default window and 606
  - example 611
- windows
  - active
    - erasing 608
  - Call Stack 240
  - controlling 609
  - creating 609
  - debugging 239
  - default type 606
  - defined 606

- Edit *See* Edit, window
- Help *See* Help
- Inspector 229
- managing
  - header file 607
- output in 609
- scrolling 608
- text
  - creating 611
  - default size 610
  - using IDE 26, 27, 28, 29, 30
- word
  - delete 630
  - mark 630
- word aligning
  - integers 275.
- word alignment 413
  - memory and
    - structures 413
- write (function) 192
- write, ostream member function 555
- write block 630
- writing a wrapper class 495
- ws (manipulator) 192, 534
- \_wscroll (global variable) 608
- wxxx TCC options (warnings) 282-284
- warn pragma and 519

## X

- x\_fill, ios data member 547
- x\_flags, ios data member 547
- x\_precision, ios data member 547
- x\_tie, ios data member 548
- x\_width, ios data member 548
- /x IDE option (extended memory) *See*
  - extended memory, IDE option (/x) and
- X TCC option (disable autodependency information) 279
- xalloc, ios member function 550
- \xH (display a string of hexadecimal digits) 359
- XOR operator (^) 425, 436
  - truth table 436

## Y

- Y
  - command-line compiler option (compiler generated code for overlays) *592*
  - TCC option (overlays) *279, 521*
- y TCC option (line numbers) *279*
- /ye TLINK option (expanded memory) *344*
- Yo option (overlays) *590*
- Yo TCC option (overlays) *279*
- /yx TLINK option (extended memory) *345*

## Z

- zero flag *574*
- zoom box *28, 29*
- Zoom command
  - hot key *25, 26*
- zV options (far virtual table segments) *285*
- zX options (code and data segments) *284, 285, 584*

# 3.0

# TURBO C<sup>®</sup>++

**Borland<sup>®</sup>**

Copyright © 1998 Borland International, Inc. All rights reserved. All Borland product names are trademarks of Borland International, Inc. Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Internet: <http://www.borland.com/> Offices in: Australia, Canada, Chile, France, Germany, Hong Kong, Japan, Latin America, The Netherlands, Singapore, Taiwan, and United Kingdom. Part #14MN-CPP01-30 • BOR 10560