

TURBO PASCAL[®]

6.0

USER'S
GUIDE

B O R L A N D



Turbo Pascal[®]

Version 6.0

User's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1983, 1990 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.

C O N T E N T S

Introduction	1	Data types	31
The Turbo Pascal manuals	2	Integer data types	31
Installing Turbo Pascal	2	Real data types	32
Customizing Turbo Pascal	4	Character and string data types	33
Laptop systems	4	Boolean data type	35
The README file	4	Pointer data type	36
Typefaces used in these books	5	Identifiers	37
How to contact Borland	6	Operators	38
Chapter 1 Learning the new IDE	7	Assignment operators	38
The components	8	Arithmetic operators	39
The menu bar and menus	8	Bitwise operators	39
Shortcuts	9	Relational operators	39
Turbo Pascal windows	11	Logical operators	40
Window management	13	Address operators	41
The status line	14	Set operators	41
Dialog boxes	15	String operators	41
Check boxes and radio buttons	16	Output	41
Input boxes and lists	16	The Writeln procedure	41
Editing	17	Input	43
Starting Turbo Pascal	18	Conditional statements	43
Creating your first program	18	The if statement	44
Analyzing your first program	19	The case statement	45
Saving your first program	20	Loops	45
Compiling your first program	20	The while loop	46
Running your first program	21	The repeat..until loop	46
Checking the files you've created	22	The for loop	48
Stepping up: your second program	22	Procedures and functions	49
Debugging your program	24	Program structure	49
Using the Watch window	25	Procedure and function structure	50
Fixing your second program	25	Sample program	51
Programming pizazz: your third program	26	Program comments	52
Chapter 2 Programming in Turbo Pascal	29	Chapter 3 Turbo Pascal units	55
The elements of programming	29	What is a unit?	55
		A unit's structure	56
		Interface section	57

Implementation section	57	Early binding vs. late binding	96
Initialization section	59	Object type compatibility	97
How are units used?	59	Polymorphic objects	99
Referencing unit declarations	60	Virtual methods	101
Implementation section uses clause ...	63	Range checking virtual method	
Circular unit references	63	calls	103
Sharing other declarations	65	Once virtual, always virtual	103
The standard units	66	An example of late binding	103
System	66	Procedure or method?	105
Dos	66	Object extensibility	108
Overlay	66	Static or virtual methods	109
Crt	66	Dynamic objects	110
Printer	67	Allocation and initialization with	
Graph	67	New	111
Turbo3 and Graph3	67	Disposing dynamic objects	112
Writing your own units	68	Destructors	112
Compiling units	68	An example of dynamic object	
An example	69	allocation	115
Units and large programs	70	Disposing of a complex data structure	
Units as overlays	71	on the heap	116
The TPUMOVER utility	71	Where to now?	118
Chapter 4 Object-oriented		Conclusion	119
programming	73	Chapter 5 Debugging Turbo Pascal	
Objects?	74	programs	121
Inheritance	75	Taxonomy of bugs	121
Objects: records that inherit	76	Compile-time errors	122
Instances of object types	79	Run-time errors	122
An object's fields	79	Logic errors	122
Good practice and bad practice	79	The integrated debugger	123
Methods	80	What the debugger can do	123
Code and data together	82	Tracing	124
Defining methods	82	Go to cursor	124
Method scope and the Self parameter .	83	Breaking	124
Object data fields and method formal		Watching	124
parameters	85	Evaluate/Modify	124
Objects exported by units	85	Navigating	125
Private section	87	In and out of the debugger	125
Programming in the active voice	87	Starting a debugging session	126
Encapsulation	88	Restarting a debugging session ...	126
Methods: no downside	89	Ending a debugging session	127
Extending objects	90	Tracing through your program	127
Inheriting static methods	93	Stepping through your program	129
Virtual methods and polymorphism ..	95	Using breakpoints	131

Using Ctrl-Break	133	Defining at the command line	165
Watching values	133	Defining in the IDE	165
Types of watch expressions	135	Predefined symbols	165
Format specifiers	136	The VER60 symbol	166
Typecasting	137	The MSDOS and CPU86 symbols .	166
Expressions	138	The CPU87 symbol	166
Editing and deleting watches	139	The IFxxx, ELSE, and ENDIF	
Evaluating and modifying	139	symbols	167
Modifying expressions	140	The IFDEF and IFNDEF directives ..	168
Navigation	142	The IFOPT directive	169
The call stack	142	Optimizing code	170
Finding procedures and functions .	143	Chapter 7 The IDE reference	173
Object-oriented debugging	144	Starting and exiting	174
Stepping and tracing method calls ...	144	Command-line options	174
Objects in the Evaluate window	144	The /C option	174
Expressions in the Find Procedure		The /D option	174
command	145	The /E option	175
General issues	145	The /G option	175
Writing programs for debugging	145	The /L option	175
Memory issues	146	The /N option	175
Outside the IDE	147	The /O option	176
Re-configuring Turbo Pascal	147	The /P option	176
Modifying your source code	148	The /S option	176
Turbo Debugger and the IDE	149	The /T option	176
Recursive routines	150	The /W option	176
Where debugging won't go	150	The /X option	176
Common pitfalls	151	Exiting Turbo Pascal	177
Error handling	152	≡ (System) menu	177
Input/output error checking	152	About	177
Range checking	153	Refresh Display	177
Other error-handling abilities	154	Clear Desktop	177
Chapter 6 Project management	157	File menu	178
Program organization	157	Open	178
Initialization	159	Using the File list box	179
The Build and Make options	159	New	179
The Make option	160	Save	179
The Build option	161	Save As	180
The Stand-alone MAKE utility	161	Save All	180
A quick example	162	Change Dir	180
Creating a makefile	163	Print	181
Using MAKE	163	Get Info	181
Conditional compilation	164	DOS Shell	182
The DEFINE and UNDEF directives .	164	Exit	182

Edit menu	182	Run-time Errors	200
Restore Line	183	Syntax Options	200
Cut	184	Numeric Processing	201
Copy	184	Debugging	201
Paste	184	Conditional Defines	202
Copy Example	184	Memory Sizes	202
Show Clipboard	184	Linker	203
Clear	185	Map File	203
Search menu	185	Link Buffer (memory)	203
Find	185	Debugger	203
Options	185	Debugging	204
Direction	186	Display swapping	204
Scope	186	Directories	205
Origin	187	Environment	206
Replace	187	Preferences	206
Search Again	188	Editor	208
Go to Line Number	188	Mouse	209
Find Procedure	188	Startup	209
Find Error	188	Colors	210
Run menu	189	Save Options	211
Run	189	Retrieve Options	211
Program Reset	190	Window menu	211
Go to Cursor	190	Size/Move	212
Trace Into	191	Zoom	212
Step Over	191	Tile	212
Parameters	191	Cascade	212
Compile menu	192	Next	212
Compile	192	Previous	212
Make	192	Close	213
Build	193	Watch	213
Destination	193	Register	213
Primary File	193	Output	213
Debug menu	194	Call Stack	214
Evaluate/Modify	194	User Screen	214
Watches	196	List	214
Add Watch	196	Help menu	214
Delete Watch	196	Contents	215
Edit Watch	196	Index	216
Remove All Watches	196	Topic Search	216
Toggle Breakpoint	196	Previous Topic	216
Breakpoints	197	Help on Help	217
Options menu	198	Chapter 8 The editor from A to Z	219
Compiler	199	The new and the old	219
Code Generation	199		

Editor reference	220	Compiler mode options	233
Jumping around	222	The make (/M) option	233
Block commands	222	The build all (/B) option	234
Other editing commands	224	The find error option	234
Search and replace	225	The link buffer option	235
Searching and searching again	225	The quiet option	235
Search and replace	225	Directory options	236
Pair matching	226	The EXE & TPU directory option	236
Directional and nondirectional		The include directories option	236
matching	227	The unit directories option	236
Nestable delimiters	227	The object files directories option	237
Chapter 9 The command-line		Debug options	237
compiler	229	The map file option	237
Compiler options	230	The standalone debugging option ...	238
Compiler directive options	231	The TPC.CFG file	239
The switch directive option	231	Compiling in protected mode	240
The conditional defines option	232	Index	241

T A B L E S

1.1: General hot keys	9	5.1: Watch expression values	138
1.2: Menu hot keys	10	6.1: Summary of compiler directives ...	164
1.3: Editing hot keys	10	6.2: Predefined conditional symbols ...	165
1.4: Window management hot keys	10	7.1: Format specifiers recognized in debugger expressions	195
1.5: Online help hot keys	11	8.1: Full summary of editor commands ..	220
1.6: Debugging/Running hot keys	11	8.2: Block commands in depth	223
1.7: Manipulating windows	14	8.3: Other editor commands in depth ...	224
2.1: Integer data types	32	8.4: Delimiter pairs	227
2.2: Real data types	33	9.1: Command-line options	230
2.3: Operator precedence	38		

F I G U R E S

1.1: A typical window	12	7.9: The Program Parameters dialog box .	192
1.2: A typical status line	15	7.10: The Evaluate/Modify dialog box ..	194
1.3: A typical dialog box	15	7.11: The Breakpoints dialog box	197
4.1: A partial taxonomy chart of insects ...	76	7.12: The Edit Breakpoint dialog box	198
4.2: Layout of program ListDemo's data structures	116	7.13: The Compiler Options dialog box ..	199
7.1: The Open a File dialog box	178	7.14: The Linker dialog box	203
7.2: The Save File As dialog box	180	7.15: The Debugger dialog box	204
7.3: The Change Dir dialog box	181	7.16: The Directories dialog box	205
7.4: The Get Info box	182	7.17: The Preferences dialog box	206
7.5: The Find dialog box	185	7.18: The Startup Options dialog box	210
7.6: The Replace dialog box	187	7.19: The Colors dialog box	210
7.7: The Go to Line Number dialog box .	188	8.1: Search for match to square bracket or parenthesis	228
7.8: The Find Procedure dialog box	188		

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

Turbo Pascal is designed to meet the needs of all types of users of IBM PS/2s, PCs, and compatibles. It is a structured, high-level language you can use to write programs for any type or size of application.

Turbo Pascal 6.0 builds on what is already the world's standard Pascal compiler. Fully compatible with code written using earlier versions of Turbo Pascal, this new version also includes

- a brand new, state-of-the-art integrated development environment (IDE), with
 - multiple overlapping windows
 - mouse support, menus, dialogs
 - multi-file editor that can edit files up to 1 Mb
 - enhanced debugging facilities
 - complete save and restore of desktop
- an object-oriented application framework, Turbo Vision, for use in your applications (it gives you the same tools we used to write the IDE)
- a full-featured inline assembler
- private fields and methods in object declarations
- extended syntax directive (**\$X**) that lets you treat functions like procedures (and ignore function results)
- 286 code generation (**\$G** directive)
- address references in typed constants
- far and near procedure directives
- link in initialized data (**\$L**) from object files
- new heap manager is faster and reduces fragmentation (*FreeMin* and *FreeList* have been replaced; refer to Chapter 16 in the *Programmer's Guide* for more information)

For more about compiler directives, refer to Chapter 21 in the Programmer's Guide.

- enhanced hypertext online help facilities, with complete cut-and-paste example code for every library procedure and function

The Turbo Pascal manuals

The four manuals in the Turbo Pascal documentation set serve four different purposes. Briefly, here's what each contains:

The *Users's Guide* (this book) contains information on how to install, learn and use Turbo Pascal's integrated environment and command-line compilers. It also includes information on the basics of programming in Turbo Pascal, as well as more advanced topics like debugging, object-oriented programming, and management of larger projects.

The *Programmer's Guide* is a reference guide to technical aspects of Turbo Pascal, describing in detail the definition of the language, the contents of the standard libraries, how they are implemented in Turbo Pascal, and use of Turbo Pascal with assembly language. This volume also contains explanations of all compiler directives and error messages used by Turbo Pascal.

The *Library Reference* contains an alphabetical reference to all the standard procedures and functions supported by the Turbo Pascal run-time library.

The *Turbo Vision Guide* tells you all about the Turbo Vision object-oriented application framework for building windowing applications. This volume contains step-by-step tutorials on how to put together a Turbo Vision application, reference material on all the tools provided in the library, and an alphabetical reference for all the objects, procedures, functions, and types in Turbo Vision.

Installing Turbo Pascal

Turbo Pascal comes with an automated installation program called INSTALL. You should use INSTALL to load Turbo Pascal onto your system, as it will ensure that you get all the files you need into the places that you need them. INSTALL will automatically create directories and copy files from the distribution disks to your hard disk. INSTALL's operation is

self-explanatory. If you have installed earlier versions of Turbo Pascal or Turbo C++, you are already familiar with the process.

If you don't already know how to use DOS commands, refer to your DOS reference manual before setting up Turbo Pascal on your system.

We assume you are already familiar with DOS commands. For example, you'll need the DISKCOPY command to make backup copies of your distribution disks. Make a complete working copy of the distribution disks when you receive them, then store the original disks away in a safe place.

If you are not familiar with Borland's No-Nonsense License Statement, read the agreement included with your Turbo Pascal package. Be sure to mail us your filled-in product registration card; this guarantees that you'll be among the first to hear about the hottest new upgrades and versions of Turbo Pascal.

If you intend to use Turbo Pascal on a floppy-disk-only system, please read the information in the README file about floppy disk installation first.

To install Turbo Pascal on a hard disk:

- Insert the installation diskette into drive A.
- Type the command
A:INSTALL
and press *Enter*.
- Press *Enter* at the installation screen.
- Follow the prompts.

When it is finished, INSTALL reminds you to read the README file, which contains last-minute details about this release. INSTALL also tells you how to configure your CONFIG.SYS and AUTOEXEC.BAT files to use Turbo Pascal.

Also, once you've installed Turbo Pascal, you'll have a chance to try out TPTOUR. TPTOUR is a guided tour of some of the highlights of the new Turbo Pascal integrated environment. TPTOUR is installed by default in your main Turbo Pascal directory.

After installing Turbo Pascal and trying out TPTOUR, you may be anxious to get up and running with the new IDE. If so, just get to the directory that holds your newly-installed Turbo Pascal programs and type TURBO, then press *Enter*. Otherwise, just keep reading the rest of this introduction for more important startup information.

Press Alt-X to leave the IDE.

Customizing Turbo Pascal

The new integrated environment allows you to do all customization (colors, options, preferences) without exiting the program to use external utilities. You can also specify some options at the command line when you start the integrated environment (see Chapter 7).

If you want the IDE to save and restore your desktop between sessions, go to the **Preferences** dialog box (**O**ptions | **E**nvironment) and turn Auto Save on for both **E**nvironment and **D**esktop.

Laptop systems

If you have a laptop computer with an LCD or plasma display, in addition to carrying out the procedures given in the previous sections, you need to set your screen parameters before using Turbo Pascal. The Turbo Pascal integrated environment version works best if you type `MODE BW80` at the DOS prompt before running Turbo Pascal.

This and other options are explained in Chapter 7.

Although you could create a batch file to take care of this, you can also easily customize Turbo Pascal for a black-and-white screen with the Startup Options dialog box in the IDE (**O**ptions | **E**nvironment | **S**tartup).

The README file

The README file contains last-minute information that may not be in these manuals. It also lists every file on the distribution disks, with a brief description of what each one contains.

Here's how to access the README file:

1. If you haven't installed Turbo Pascal yet, insert your Turbo Pascal installation disk into drive A.
2. Type `A:` and press *Enter*.
3. Type `README` and press *Enter*. Once you are in README, use the `↑` and `↓` keys to scroll through the file.
4. Press *Esc* to exit.

If you've already installed Turbo Pascal, you can open README in an edit window by following these steps:

1. Start Turbo Pascal from the directory in which you installed it by typing `TURBO` and pressing *Enter*.
2. Press *F3*. Type in README and press *Enter*. Turbo Pascal will open the README file in an edit window.
3. When you're done with the README file, press *Alt-F3* to close the editor window or *Alt-X* to leave the IDE.

Typefaces used in these books

All typefaces used in this manual were produced by Borland's Sprint: The Professional Word Processor, on a PostScript laser printer. Their uses are as follows:

- Monospace type** This typeface represents text as it appears on-screen or in a program. It is also used for anything you must type (such as `TURBO` to start up Turbo Pascal).
- []** 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.*
- Boldface** This typeface is used in text for Turbo Pascal reserved words, for compiler directives (**`{$I-}`**) and for command-line options (***(/A)***).
- Italics** Italics indicate 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.

How to contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type GO BOR from the main CompuServe menu and choose "Borland Programming Forum A (Turbo Pascal)" from the Borland main menu. Leave your questions or comments there for the support staff to process.

If you prefer, write a letter with your comments and send it to

Borland International
Technical Support Department—Turbo Pascal
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95067-0001, USA

See the README file included with your distribution disks for details on how to report a bug.

You can also telephone our Technical Support department between 6 am and 5 pm Pacific time at (408) 438-5300. Please have the following information handy before you call:

1. Product name and serial number on your original distribution disk. Please have your serial number ready, or we won't be able to process your call.
2. Product version number. The version number for Turbo Pascal is displayed when you first load the program and before you press any keys.
3. Computer brand, model, and the brands and model numbers of any additional hardware.
4. Operating system and version number. (The version number can be determined by typing VER at the DOS prompt.)
5. Contents of your AUTOEXEC.BAT file.
6. Contents of your CONFIG.SYS file.

Learning the new IDE

There's also a command-line version available, TPC.EXE.

Turbo Pascal is more than just a fast Pascal compiler; it is an efficient Pascal compiler with an easy-to-learn and easy-to-use integrated development environment (for short, we call it the IDE). With Turbo Pascal, you don't need to use a separate editor, compiler, linker, and debugger in order to create, debug, and run your Pascal programs. All these features are built into Turbo Pascal, and they are all accessible from the IDE.

You can begin building your first Turbo Pascal program using the compiler built into the IDE. By the end of this chapter, you'll have learned your way around the development environment, written and saved three small programs, and learned some basic programming skills.

Online context-sensitive help is only a keystroke (or a mouse click) away. You can get help at any point (except when *your* program has control) by pressing the shortcut *F1*. The **Help** menu (*Alt-H*) provides you with a table of contents to the help system, a detailed index, searching capabilities (*Ctrl-F1*), the ability to go back to other screens (*Alt-F1*), and help on Help (*F1* when you're already in help). Any help screen can contain one or more *keywords* (highlighted items) on which you can get more information.

If you want more detail about the IDE, look at Chapter 7, "The IDE reference."

The components

*We often abbreviate menu items. For example, to choose add a watch (Debug | Watch | Add Watch); we'll tell you to choose **D | W | Add Watch**.*

There are three visible components to the IDE: the menu bar at the top, the desktop, and the status line at the bottom. Many menu items also offer dialog boxes. Before we detail each menu item in the IDE, we'll describe these more generic components.

The menu bar and menus

The menu bar is your primary access to all the menu commands. The only time the menu bar is not visible is when you're viewing your program's output. You'll see a highlighted menu title when the menu bar is active; this is the currently *selected* menu.

If a menu command is followed by an ellipsis mark (...), choosing the command displays a dialog box. If the command is followed by an arrow (►), the command leads to another menu (a pop-up menu). A command without either an ellipsis mark or an arrow indicates that the action occurs once you choose it.

Here is how you choose menu commands using just the keyboard:

1. Press *F10*. This makes the menu bar active.
2. Use the arrow keys to select the menu you want to display. Then press *Enter*.

To cancel an action, press Esc.

As a shortcut for this step, you can just press the highlighted letter of the menu title. For example, from the menu bar, press *E* to quickly display the **E**dit menu. From anywhere, press *Alt* and the highlighted letter to display the menu you want.

3. Use the arrow keys again to select the command you want. Then press *Enter*.

Again, as a shortcut, you can just press the highlighted letter of a command to choose it once the menu is displayed.

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



You can also use a mouse to choose commands. The process is this:

Turbo Pascal uses only the left mouse button. You can, however, customize the right button and other mouse settings; see page 209.

1. Click the desired menu title to display the menu.
2. Click the desired command.

You can also 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. You can still select (highlight) an unavailable command in order to get online help about it.

Shortcuts

Turbo Pascal offers a number of quick ways to choose menu commands. For example, mouse users can combine the two-step process into one by dragging from the menu title down to the menu commands and releasing the mouse button when the command you want is selected.

In dialog boxes, just press the highlighted letter.

From the keyboard, you can use a number of shortcuts (or *hot keys*) to access the menu bar and choose commands. You can get to, or activate, main menu items by pressing *Alt* and the highlighted letter. Once you're in a menu, you can press an item's highlighted letter or the shortcut next to it. You can also click on shortcuts on the status line.

The following tables list the most-used Turbo Pascal hot keys:

Table 1.1
General hot keys

Key(s)	Menu item	Function
F1	Help	Displays a 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.
F6	Window Next	Cycles through all open windows.
F7	Run Trace Into	Runs your program in debug mode, tracing into procedures.
F8	Run Step Over	Runs your program in debug mode, stepping over procedure calls.
F9	Compile Make	Makes the current executable.
F10	(none)	Takes you to the menu bar.

Table 1.2
Menu hot keys

Key(s)	Menu item	Function
<i>Alt-Spacebar</i>	≡ menu	Takes you to the ≡ (System) menu
<i>Alt-C</i>	C ompile menu	Takes you to the Compile menu
<i>Alt-D</i>	D ebug menu	Takes you to the Debug menu
<i>Alt-E</i>	E dit menu	Takes you to the Edit menu
<i>Alt-F</i>	F ile menu	Takes you to the File menu
<i>Alt-H</i>	H elp menu	Takes you to the Help menu
<i>Alt-O</i>	O ptions menu	Takes you to the Options menu
<i>Alt-R</i>	R un menu	Takes you to the Run menu
<i>Alt-S</i>	S earch menu	Takes you to the Search menu
<i>Alt-W</i>	W indow menu	Takes you to the Window menu
<i>Alt-X</i>	F ile E xit	Exits Turbo Pascal to DOS

Table 1.3
Editing hot keys

Key(s)	Menu item	Function
<i>Ctrl-Del</i>	E dit C lear	Removes selected text from the window and doesn't put it in the Clipboard
<i>Ctrl-Ins</i>	E dit C opy	Copies selected text to Clipboard
<i>Shift-Del</i>	E dit C ut	Places selected text in the Clipboard, deletes selection
<i>Shift-Ins</i>	E dit P aste	Pastes text from the Clipboard into the active window
<i>Ctrl-L</i>	S earch S earch Again	Repeats last Find or Replace command
<i>F2</i>	F ile S ave	Saves the file in the active Edit window
<i>F3</i>	F ile O pen	Lets you open a file

Table 1.4
Window management hot keys

Key(s)	Menu item	Function
<i>Alt-#</i>	(none)	Displays a window, where # is the number of the window you want to view
<i>Alt-0</i>	W indow L ist	Displays a list of open windows
<i>Alt-F3</i>	W indow C lose	Closes the active window
<i>Alt-F5</i>	W indow U ser Screen	Displays User Screen
<i>Shift-F6</i>	W indow P revious	Cycles backward through active windows
<i>F5</i>	W indow Z oom	Zooms/unzooms the active window
<i>F6</i>	W indow N ext	Cycles forward through active windows
<i>Ctrl-F5</i>	W indow S ize/Move	Changes size or position of active window

Table 1.5
Online help hot keys

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

Table 1.6
Debugging/Running hot keys

Key(s)	Menu item	Function
<i>Alt-F9</i>	Compile Compile	Compiles last file in editor
<i>Ctrl-F2</i>	Run Program Reset	Resets running program
<i>Ctrl-F4</i>	Debug Evaluate/Modify	Evaluates an expression
<i>Ctrl-F7</i>	Debug Add Watch	Adds a watch expression
<i>Ctrl-F8</i>	Debug Toggle Breakpoint	Sets or clears conditional breakpoint
<i>Ctrl-F9</i>	Run Run	Runs program
<i>F4</i>	Run Go To Cursor	Runs program to cursor position
<i>F7</i>	Run Trace Into	Executes tracing into procedures
<i>F8</i>	Run Step Over	Executes skipping procedure calls
<i>F9</i>	Compile Make	Makes (compiles/links) program

Turbo Pascal windows

You can increase the number of windows that can potentially be opened by increasing the heap size using the Startup option (Options | Environment).

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

You can have any number of windows open in Turbo Pascal (memory and heap space allowing), but only one window can be *active* at any time. The active window is the one that you're currently working in. Any command you choose or text you type generally applies only to the active window. There are several types of windows, but most have these things in common:

- a title bar
- a close box
- scroll bars
- a resize corner

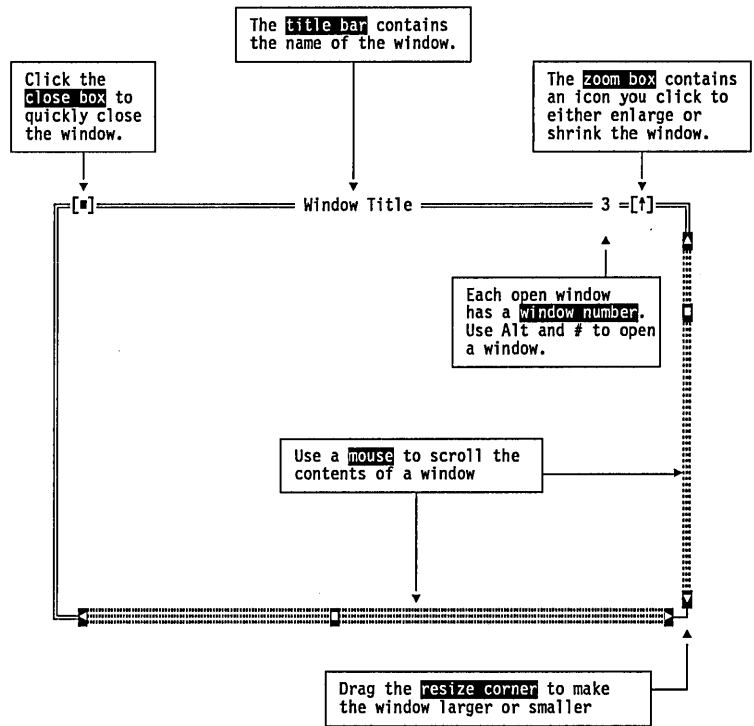
- a zoom box
- a window number

Turbo Pascal makes it easy to spot the active window by placing a double-lined border around it. The active window always has a close box, a zoom box, scroll bars, and a resize corner. If your windows are overlapping, the active window is always the one on top of all the others (the frontmost one).

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

This is what a typical window looks like:

Figure 1.1
A typical window



The *close box* of a window is the box in the upper left corner. You click this box to quickly close the window. (Or choose **Window | Close** or press *Alt-F3*.) The Help window is considered temporary; you can close it 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 (and vice versa). You can also drag the title bar to move the window around.

Each of the windows you open in Turbo Pascal have a *window number* in the upper right border. *Alt-0* (zero) gives you a list of all windows you have open. You can make a window active (bringing 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.

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**, or press *F5*.

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



Scroll bars let both mouse and keyboard users see how far into the file they've gone.



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.

The *resize box* is in the lower right corner of a window. You drag any corner to make the window larger or smaller. You can spot the resize corner by its single-line border instead of the double-line border used in the rest of the window. To resize using the keyboard, choose **Size/Move** from the **Window** menu, or press *Ctrl-F5*.

Window management

Table 1.7 gives you a quick rundown of how to handle windows in Turbo Pascal. Note that you don't need a mouse to perform these actions—a keyboard works just fine.

Table 1.7
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, or press <i>F3</i> .
Open other windows	Choose the desired window from the Window menu
Close a window	Choose Close from the Window menu (or press <i>Alt-F3</i>), or click the close box of the window.
Activate a window	<p>Click anywhere in the window, or</p> <p>Press <i>Alt</i> plus the window number in the upper right border of the window), or</p> <p>Choose Window List or press <i>Alt-0</i> and select the window from the list, or</p> <p>Choose Window Next or <i>F6</i> to make the next window active (next in the order you first opened them). Or press <i>Alt-F6</i> to cycle backward.</p>
Move the active window	<p>Drag its title bar, or press <i>Ctrl-F5</i> (Window Size/Move) and use the arrow keys to place the window where you want it, then press <i>Enter</i>.</p>
Resize the active window	<p>Drag the resize corner (or any other 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>. The shortcut is to press <i>Ctrl-F5</i> and then use <i>Shift</i> and the arrow keys.</p>
Zoom the active window	<p>Click the zoom box in the upper right corner of the window, or</p> <p>Double-click the window's title bar, or</p> <p>Choose Window Zoom, or press <i>F5</i>.</p>

The status line

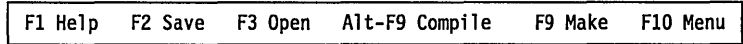
The status line appears at the bottom of the screen; it

- reminds you of basic keystrokes and shortcuts 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 1.2
A typical status line

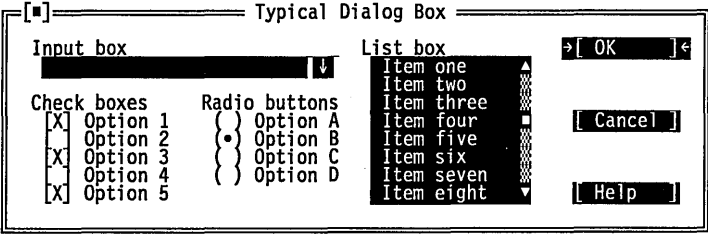


Dialog boxes

If a menu command has an ellipsis after it (...), the command opens a *dialog box*. A dialog box is 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: radio buttons, check boxes, action buttons, input boxes, and list boxes. Here's a typical dialog box that illustrates some of these items:

Figure 1.3
A typical dialog box



If you have a color monitor, Turbo Pascal will use different colors for various elements of the dialog box.

The Breakpoints Options dialog box is unique in that it doesn't have a Cancel button.

This dialog box has three standard action buttons: OK, Cancel, and Help. If you choose OK, the choices in the dialog box are made in Turbo Pascal; if you choose Cancel, nothing changes and no action is made, but 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, you can just click the button you want. When you're using the keyboard, you can press the highlighted letter of an item to activate it. For example, pressing *K* selects the OK button. Press *Tab* or *Shift-Tab* to move forward or back from one item to another in a dialog box. Each element highlights 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.

Check boxes and radio buttons

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. You check a check box (set it to on) by clicking it or its text, by pressing *Tab* until the check box (or its group) is highlighted and then pressing *Spacebar*, or by selecting the highlighted letter.

If selecting a check box in a group, use the arrow keys or highlighted letters to select the item you want, and then press *Spacebar*. On monochrome monitors, Turbo Pascal indicates the active check box or group of check boxes by placing a chevron symbol (*>>*) next to it. When you press *Tab*, the chevron moves to the next group of checkboxes or radio buttons.

Radio buttons are so called because they act just like the group of buttons on a car radio. There is always one—and only one—button pushed in at a time. Push one in, and the one that was in pops out.

Radio buttons differ from check boxes in that they present mutually exclusive choices. For this reason, radio buttons *always* come in groups, and exactly one (no more, no less) radio button can be on in any one group at any one time. To choose a radio button, click it or its text. From the keyboard, select 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.

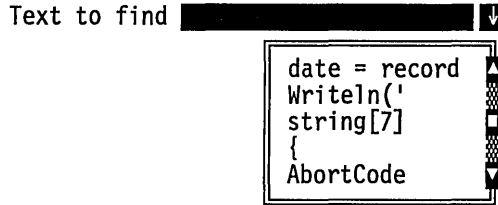
Input boxes and lists

You're probably all familiar with input boxes; this is where you type in text. Most basic text-editing keys work in the text box (for example, arrow keys, *Home*, *End*, and insert/overwrite toggles by *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, entering *^P^L* enters a *^L* into the input box. This is useful for search strings.

If an input box has a down-arrow icon to its right, there is a *history list* associated with that input box. Press *↓* to view the

history list and *Enter* to select an item from the list. The list will display any text you typed into the box the last few times you used it. If you want to reenter text that you already entered, press ↓ or click the ↓ icon. 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 seven times previously:



A final component of many dialog boxes is a *list box*. A list box lets you scroll through and select from variable-length lists 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 Pascal 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* or the arrow keys 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.

Editing

If you're a longtime user of Borland products, the following summary of new editing features can help you identify the areas that are different from older products.

Turbo Pascal's integrated editor now has

- mouse support
- support for large files (up to 1 Mb files; limited to 2 megabytes for all editors combined)
- *Shift* ↑ ↓ → ← for selecting text
- Edit windows that you can move, resize, or overlap
- multi-file capabilities let you open several files at once

- multiple windows let you have several views onto the same file or different files
- a sophisticated macro language, so you can create your own editor commands (see TEMC.DOC on your distribution disks)
- the ability to paste text or examples from the Help window
- an editable Clipboard that allows cutting, copying, and pasting in or between windows

Starting Turbo Pascal

If you have a low-density 5 1/4" floppy drive system, this procedure does not apply; read the README file for installation and operation details.

If you're using a floppy disk drive, put your Turbo Pascal system disk into Drive A and type the following command:

```
TURBO
```

Press *Enter* to run the program TURBO.EXE, which brings up the IDE.

If you're using a hard disk, get into the Turbo Pascal subdirectory you created with INSTALL (the default is C:\TP) and run TURBO.EXE by typing

```
TURBO
```

at the C:\TP> prompt. Now you're ready to write your first Turbo Pascal program.

Creating your first program

To access the examples given here, run INSTALL with the Unpack Examples option set to on.

When you load Turbo Pascal (type TURBO and press *Enter* at the DOS prompt), what you'll see is the main menu bar, a status line, an empty desktop, and a window with product version information (choosing the **About** command from the **≡**, or **S**ystem, menu at any time will bring up this information). When you press any key, the version information disappears, but the windowed environment remains.

Press *F10* to go the menu bar and then *F3* (a shortcut for **File** | **O**pen) to display the Open a File dialog box. You're in the input box, so go ahead and type in MYFIRST (you don't need the .PAS extension; it's assumed) and then press *Enter*. Now you can start

typing in the following program, pressing *Enter* at the end of each line:

Don't forget the semicolons and follow the last **end** with a period.

```
program MyFirst;
var
  A,B: Integer;
  Ratio: Real;
begin
  Write('Enter two numbers: ');
  Readln(A,B);
  Ratio := A / B;
  Writeln('The ratio is ',Ratio);
  Write('Press <Enter>...');
  Readln;
end.
```

Use the *Backspace* key to make deletions, and use the arrow keys to move around in the edit window. If you're unfamiliar with editing commands, Chapter 8 discusses all those available.

Analyzing your first program

While you can type in and run this program without ever knowing how it works, we've provided a brief explanation here. The first line you entered gives the program the name *MyFirst*. This is an optional statement, but it's a good practice to include it.

The next three lines declare some *variables*, with the word **var** signaling the start of variable declarations. *A* and *B* are declared to be of type *Integer*; that is, they can contain whole numbers, such as 52, -421, 0, 32,283, and so on. *Ratio* is declared to be of type *Real*, which means it can hold fractional numbers such as 423.328 and -0.032, in addition to all integer values.

The rest of the program contains the *statements* to be executed. The word **begin** signals the start of the program. The statements are separated by semicolons and contain instructions to write to the screen (*Write* and *Writeln*), to read from the keyboard (*Readln*), and to perform calculations (*Ratio := A / B*). The *Readln* at the end of the program will cause execution to pause (until you press *Enter*) so you can inspect the program's output. The program's execution starts with the first instruction after **begin** and continues until **end.** is encountered.

Saving your first program

After entering your first program, it's a good idea to save it to disk. To do this, choose the **Save** command from the **File** menu by pressing *F10*, then *F* to bring up the **File** menu and *S* to choose the **Save** command. An easier method would be to use the shortcut for **File | Save**, *F2*.

Compiling your first program

To compile your first program, go to the **Compile** option on the main menu. You can press *F10 C*, or *Alt-C* takes you right to it. *Alt-F9* is the quickest shortcut, initiating compilation right away.

Turbo Pascal compiles your program, changing it from Pascal (which you can read) to 8086 machine code for the microprocessor (which your PC can execute). You don't see the 8086 machine code; it's stored in memory (or on disk).

Like English, Pascal has rules of grammar you must follow. However, unlike English, Pascal's structure isn't lenient enough to allow for slang or poor syntax—the compiler must always understand what you mean. In Pascal, when you don't use the appropriate words or symbols in a statement or when you organize them incorrectly, you will get a compile-time (syntax) error.

What compile-time errors are you likely to get? Probably the most common error novice Pascal programmers will get is

Unknown identifier

or

';' expected

Pascal requires that you declare all variables, data types, constants, and subroutines—in short, all identifiers—before using them. If you refer to an undeclared identifier or if you misspell it, you'll get an error. Other common errors are unmatched **begin..end** pairs, assignment of incompatible data types (such as assigning reals to integers), parameter count and type mismatches in procedure and function calls, and so on.

When you start compiling, a box appears in the middle of the screen, giving information about the compilation taking place. If no errors occurred during compilation, the message "Compilation

successful: press any key" flashes across the box. The box remains visible until you press a key. See how fast that went?

If an error occurs during compilation, Turbo Pascal stops, positions the cursor at the point of error in the editor, and displays an error message at the top of the editor, as it does with compile-time error messages. (The first key you press will clear the error message, and *Ctrl-Q W* will bring it back until you change files or recompile. Make the correction, save the updated file, and compile it again.)

Running your first program

After you've fixed any typing errors that might have occurred, go to the main menu and choose **Run | Run** (or press *Ctrl-F9*). You're placed at the User screen, and the message

Enter two numbers:

appears on the screen. Type in any two integers (whole numbers), with a space between them, and press *Enter*. The following message will appear:

The ratio is

followed by the ratio of the first number to the second. On the next line the message "Press <Enter>..." will appear and the program will wait for you to press the *Enter* key. To review your program output, choose **Window | User Screen** (or press *Alt-F5*).

If an error occurs while your program is executing, you'll get a message on the screen that looks like this:

Run-time error <errnum> at <segment>:<offset>

where <errnum> is the appropriate error number (see Appendix A in the *Programmer's Guide*, "Error messages," for information on compiler and run-time error messages), and <segment>:<offset> is the memory address where the error occurred. (If you need this number later, look for it in the Output window.) You'll be positioned at the point of error in your program with a descriptive error message displayed on the editor status line. While the message is still on the editor status line, you can press *F1* to get help with that particular error. Any other keystroke clears the error message. If you need to find the error location again, choose **Search | Find Error**.

When your program has finished executing, you're returned to the place in your program where you started. You can now modify your program if you wish. If you choose the **Run | Run** command before you make any changes to your program, Turbo Pascal immediately executes it again, without recompiling.

Once you're back in the IDE after executing your program, you can view your program's output by choosing the **Run | User Screen** command (or by pressing *Alt-F5*). Choose it again to return to the Turbo Pascal environment.

Checking the files you've created

If you exit Turbo Pascal (choose **Exit** from the **File** menu), you can see a directory listing of the source (Pascal) file you've created. To exit, press *D* (for **DOS Shell**) in the **File** menu or, alternatively, press *X* (for **Exit**) and type the following command at the DOS prompt:

```
DIR MYFIRST.*
```

You'll get a listing that looks something like this:

```
MYFIRST  PAS    217  8-10-88  11:07a
```

The file MYFIRST.PAS contains the Pascal program you just wrote. If you saved the program while you were typing, you'll also see a backup file MYFIRST.BAK, which was created automatically by the editor.



You'll only see the executable file if you've changed your default **Destination** setting in the **Compile** menu to *Disk*. You would then produce a file called MYFIRST.EXE, which would contain the machine code that Turbo Pascal generated from your program. You could then execute that program by typing MYFIRST followed by *Enter* at the DOS system prompt.

Stepping up: your second program

Now you're going to write a second program that builds upon the first. If you exited from Turbo Pascal using the **DOS Shell** command from the **File** menu, you can return to the Turbo Pascal environment by typing **EXIT** at the DOS prompt. If you exited using **Exit** from the **File** menu, you would type

```
TURBO MYFIRST.PAS
```

at the prompt in order to return to the IDE. This will place you directly into the editor. Now, modify your MYFIRST.PAS program to look like this:

```
program MySecond;
var
  A,B: Integer;
  Ratio: Real;
begin
  repeat
    Write('Enter two numbers: ');
    Readln(A,B);
    Ratio := A/B;
    Writeln('The ratio is ',Ratio:8:2);
    Write('Press <Enter>...');
    Readln;
  until B = 0;
end.
```

You want to save this as a separate program, so go to the **File** menu, select **Save As**, and type in MYSECOND.PAS and press *Enter*.

Go ahead and compile and run your second program using *Ctrl-F9*. This tells Turbo Pascal to run your updated program. And since you've made changes to the program, Turbo Pascal automatically compiles the program before running it.

A major change has been made to the program: The statements have been enclosed in the **repeat..until** loop. This causes all the statements between **repeat** and **until** to be executed until the expression following **until** is True. A test is made to see if *B* has a value of zero or not. If *B* has a value of zero, then the loop should exit.

Run your program, try out some values, then enter 1 0 and press *Enter*. Your program does exit, but not quite in the way you intended: It exits with a *run-time error*. You're placed back in the editor, with the cursor in front of the line

```
Ratio := A/B;
```

and the message

```
Error 200: Division by zero
```

at the top of the edit window.

Debugging your program

If you've programmed before, you may recognize this error and how to fix it. But let's take this opportunity to show you how to use the *integrated debugger* that's built into Turbo Pascal 6.0.

Turbo Pascal's integrated debugger allows you to step through your code one line at a time. At the same time, you can watch your variables to see how their values change.

To start the debugging session, choose the **Run | Trace Into** command (or press *F7*). If your program needs to be recompiled, Turbo Pascal will do so. The first statement (**begin** in this case) in the main body of your program is highlighted; from now on we'll call this highlighted bar the *run bar*.

The first *F7* you pressed initiated the debugging session. Now press *F7* to begin executing the program. The debugger just executed the invisible startup code. The next executable line in this program is the *Write* statement on line 7.

Press *F7* again. Your screen blinks momentarily, then shows your program with the run bar on the second statement (*Readln*). What's happening here is that Turbo Pascal switches to the User screen (where your program is executed and its output displayed), executes your first statement (a *Write* statement), then goes back to the editing screen.

Press *F7* again. This time, the User screen comes up and stays there. That's because a *Readln* statement is waiting for you to enter two numbers. Type two integer numbers, separated by a space; be sure the second number isn't a zero. Now press *Enter*. You're back at the Edit window, with the run bar on the assignment statement on line 9.

Press *F7* and execute the assignment statement. Now the run bar is on the *Writeln* statement on line 10. Press *F7* twice. Now you're about to execute *Readln* on line 12. Press *F7*, inspect your program's output, and then press *Enter*.

The run bar is on the **until** clause. Press *F7* one more time, and you're back at the top of the **repeat** loop.

Instead of racing through one program statement after another, the integrated debugger lets you step through your code one line at a time. This is a powerful tool, and we go into a more detailed

discussion of debugging in Chapter 5. First, we'll give you a quick taste of debugging by tracking down that divide-by-zero error.

Using the Watch window

Let's take a look at the values of the variables you've declared. Press *Alt-D* to bring up the **Debug**. Choose the **Add Watch** command from the **Watches** menu (or press *Ctrl-F7*). Type *A* in the Watch Expression input box and press *Enter*. This puts *A* in the Watch window, along with its current value. Now use the **Add Watch** command to add *B* and *Ratio* to the Watch window. Finally, use it to add the expression *A/B* to the Watch window.

Choose **Run | Trace Into** (or press *F7*) to step through your program. This time, when you have to enter two numbers, enter 0 for the second number. When you press *Enter* and return to the IDE, look at the expression *A/B* in the Watch window (press *Alt* and the window # or *Alt-W W*). Instead of having a value after it, it has the phrase "Invalid floating-point operation"; that's because dividing by zero is undefined. Note, though, that having this expression in your Watch window doesn't cause the program to stop with an error. Instead, the error is reported to you and the debugger does not perform the division in the Watch window.

Now press *F7* again, assigning *A/B* to *Ratio*. At this point, your program does halt, and the error message "Division by zero" appears at the top of the Edit window again.

Fixing your second program

Now you probably have a good idea of what's wrong with your program: If you enter a value of zero for the second number (*B*), the program halts with a run-time error.

How do you fix it? If *B* has a value of zero, don't divide *B* into *A*. Edit your program so that it looks like this:

```
program MySecond;
var
  A,B: Integer;
  Ratio: Real;
begin
  repeat
    Write('Enter two numbers: ');
    Readln(A,B);
    if B = 0 then
      Writeln('The ratio is undefined')
    else
```

```

begin
  Ratio := A / B;
  Writeln('The ratio is ',Ratio:8:2);
end;
Write('Press <Enter>...');
Readln;
until B = 0;
end.

```

Now run your program (either by itself, or using the debugger). If you do use the debugger, note how the values in the Watch window change as you step through the program. When you're ready to stop, enter 0 for B. The program will pause after printing the message "The ratio is undefined. Press <Enter>...."

Now you have an idea just how powerful the debugger is. You can step through your program line by line; you can display the value of your program's variables and expressions, and you can watch the values change as your program runs.

Programming pizazz: your third program

You need to unzip GRAPH.TPU from BGI.ZIP in order to run this program.

For the last program, let's get a little fancy and dabble in graphics. This program assumes that you have a graphics adapter for your system, and that you are currently set up to use that adapter. If you are in doubt, try the program and see what happens. If an error message appears, then you probably don't have a graphics adapter (or you have one that's not supported by our *Graph* unit). In any case, pressing *Enter* once should get you back to the IDE.

Open the file (F3) MYTHIRD.PAS and enter this program:

```

program MyThird;
uses
  Graph;
const
  Start = 25;
  Finish = 175;
  Step = 2;
var
  GraphDriver: Integer;           { Stores graphics driver number }
  GraphMode: Integer;           { Stores graphics mode for the driver }
  ErrorCode: Integer;           { Reports an error condition }
  X1, Y1, X2, Y2: Integer;
begin
  GraphDriver := Detect;         { Try to autodetect Graphics card }

```

To run this program, you must be in the same directory as the BGI driver files (*.BGI).

```
InitGraph(GraphDriver, GraphMode, '');
ErrorCode := GraphResult;
if ErrorCode <> grOk then { Error? }
begin
  Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
  Writeln('You probably don''t have a graphics card!');
  Writeln('Program aborted...');
  Readln;
  Halt(1);
end;
Y1 := Start;
Y2 := Finish;
X1 := Start;
while X1 <= Finish do
begin
  X2 := (Start+Finish) - X1;
  Line(X1, Y1, X2, Y2);
  X1 := X1 + Step;
end;
X1 := Start;
X2 := Finish;
Y1 := Start;
while Y1 <= Finish do
begin
  Y2 := (Start+Finish) - Y1;
  Line(X1, Y1, X2, Y2);
  Y1 := Y1 + Step;
end;
OutText('Press <Enter> to quit:');
Readln;
CloseGraph;
end. { MyThird }
```

Save this program (*F2*) and then compile it (*Alt-F9*). If you have no errors during compilation, choose **Run | Run** (*Ctrl-F9*) to run it. This program produces a square with some wavy patterns along the edges. When execution is over, you'll be returned to your program.

The **uses** clause says that the program uses a unit named *Graph*. A *unit* is a library, or collection, of subroutines (procedures and functions) and other declarations. In this case, the unit *Graph* contains the routines you want to use: *InitGraph*, *Line*, *CloseGraph*, and more.

The section labeled **const** defines three numeric constants—*Start*, *Finish*, and *Step*—that affect the size, location, and appearance of

the square. By changing their values, you can change how the square looks.



Warning: Don't set *Step* to anything less than 1; if you do, the program will get stuck in what is known as an *infinite loop* (a loop that circles endlessly). If you've compiled to disk and are running the .EXE from DOS, you won't be able to exit except by pressing *Ctrl-Alt Del* or by turning your PC off. If you're running from inside the IDE, you can interrupt the program by pressing *Ctrl-Break*.

The variables *X1*, *Y1*, *X2*, and *Y2* hold the values of locations along opposite sides of the square. The square itself is drawn by drawing a straight line from *X1,Y1* to *X2,Y2*. The coordinates are then changed, and the next line drawn. The coordinates always start out in opposite corners: The first line drawn goes from (25,25) to (175,175).

The program itself consists primarily of two loops. The first loop draws a line from (25,25) to (175,175). It then moves the *X* (horizontal) coordinates by two, so that the next line goes from (27,25) to (173,175). This continues until the loop draws a line from (175,25) to (25,175).

The program then goes into its second loop, which pursues a similar course, changing the *Y* (vertical) coordinates by two each time. The routine *Line* is from the *Graph* unit and draws a line between the endpoints given.

The final *Readln* statement causes the program to wait for you to press a key before it goes back into text mode and returns to the IDE.

You might want to step through this program line-by-line using the integrated debugger and then watch it swap back and forth between the program's graphics mode and the IDE's text mode.

Programming in Turbo Pascal

The Pascal language was designed by Niklaus Wirth in the early 1970s to teach programming. Because of that, it's particularly well-suited as a first programming language. And if you've already programmed in other languages, you'll find it easy to pick up Pascal.

To get you started on the road to Pascal programming, this chapter will teach you the basic elements of the Pascal language and show you how to use them in your programs. However, because we don't cover everything about Pascal programming here, you might want to pick up a copy of the *Turbo Pascal Disk Tutor* (Borland Osborne/McGraw Hill), a complete book-plus-disk tutorial about programming in Pascal and using Turbo Pascal.

Before you work through this chapter, you might want to read Chapter 7, "The IDE reference," and Chapter 8, "The editor from A to Z," to learn about the environment and text editor in Turbo Pascal. If you haven't already installed Turbo Pascal as described in the introduction, you should do so now.

The elements of programming

Most programs are designed to solve a problem. They solve problems by manipulating information or data. As a programmer, you do the following:

- get the information into the program—*input*.
- have a place to keep it—*data*.
- give the right instructions to manipulate the data—*operations*.
- be able to get the data back out of the program to the user (you, usually)—*output*.

You can organize your instructions so that

- some are executed only when a specific condition (or set of conditions) is True—*conditional execution*.
- others are repeated a number of times—*loops*.
- others are broken off into chunks that can be executed at different locations in your program—*subroutines*.

These are the seven basic elements of programming: *input*, *data*, *operations*, *output*, *conditional execution*, *loops*, and *subroutines*. This list is not comprehensive, but it does describe those elements that programs (and programming languages) usually have in common.

Many programming languages, including Pascal, have additional features. And when you want to learn a new language quickly, you can find out how that language implements these seven elements, then build from there. Here's a brief description of each element:

Input

This means reading values in from the keyboard, from a disk, or from an I/O port.

Data

These are constants, variables, and structures that contain numbers (integer and real), text (characters and strings), or addresses (of variables and structures).

Operations

These assign one value to another, combine values (add, divide, and so forth), and compare values (equal, not equal, and so on).

Output

This means writing information to the screen, to a disk, or to an I/O port.

Conditional execution

This refers to executing a set of instructions if a specified condition is True (and skipping them or executing a different set

if it is False) or if a data item has a specified value or range of values.

Loops

These execute a set of instructions some fixed number of times, while some condition is True or until some condition is True.

Subroutines

These are separately named sets of instructions that can be executed anywhere in the program just by referencing the name.

Now we'll take a look at how to use these elements in Turbo Pascal.

Data types

When you write a program, you're working with information that generally falls into one of five basic types: *integers*, *real numbers*, *characters* and *strings*, *Boolean*, and *pointers*.

Integers are the whole numbers you learned to count with (1, 5, -21, and 752, for example).

Real numbers have fractional portions (3.14159) and exponents (2.579×10^{24}). These are also sometimes known as *floating-point numbers*.

Characters are any of the letters of the alphabet, symbols, and the numbers 0-9. They can be used individually (*a*, *Z*, *!*, *3*) or combined into character strings ('This is only a test.').

Boolean expressions have one of two possible values: True or False. They are used in conditional expressions, which we'll discuss later.

Pointers hold the address of some location in the computer's memory, which in turn holds information.

Integer data types

Standard Pascal defines the data type integer as consisting of the values ranging from $-MaxInt$ through 0 to $MaxInt$, where $MaxInt$ is the largest possible integer value allowed by the compiler you're using. Turbo Pascal supports type integer, defines $MaxInt$ as equal to 32,767, and allows the value -32,768 as well. A variable of type Integer occupies 2 bytes.

Turbo Pascal also defines a long integer constant, *MaxLongInt*, with a value of 2,147,483,647.

Turbo Pascal also supports four other integer data types, each of which has a different range of values. Table 2.1 shows all five integer types.

Table 2.1
Integer data types.

Type	Range	Size in Bytes
Byte	0..255	1
Shortint	-128..127	1
Integer	-32768..32767	2
Word	0..65535	2
Longint	-2147483648..2147483647	4

A final note: Turbo Pascal allows you to use hexadecimal (base 16) integer values. To specify a constant value as hexadecimal, place a dollar sign (\$) in front of it; for example, \$27 = 39 decimal.

Real data types

Standard Pascal defines the data type Real as representing floating-point values consisting of a significand (fractional portion) multiplied by an exponent (power of 10). The number of digits (known as *significant digits*) in the significand and the range of values of the exponent are compiler-dependent. Turbo Pascal defines the type real as being 6 bytes in size, with 11 significant digits and an exponent range of 10^{-38} to 10^{38} .

Turbo Pascal also supports the IEEE Standard 754 for binary floating-point arithmetic. This includes the data types Single, Double, Extended, and Comp. Single uses 4 bytes, with 7 significant digits and an exponent range of 10^{-45} to 10^{38} ; double uses 8 bytes, with 15 significant digits and an exponent range of 10^{-324} to 10^{308} ; and extended uses 10 bytes, with 19 significant digits and an exponent range of 10^{-4951} to 10^{4931} .

If you have an 8087 math coprocessor and enable the numeric support compiler directive or environment option (**{\$N+}**), Turbo Pascal generates the proper 8087 instructions to support these types and to perform all floating-point operations on the 8087.

If you don't have an 8087 chip, but you still want to use the IEEE Standard types, you can ask Turbo Pascal to *emulate* an 8087 chip, by enabling both the 8087 emulation and numeric processing compiler directives (**{\$E+}** and **{\$N+}**, respectively). Turbo Pascal

then links in a special math library that performs floating-point functions just like an 8087 chip.

Table 2.2
Real data types

Type	Range	Significant Digits	Size in Bytes
Real	$2.9 \times 10E-39$.. $1.7 \times 10E38$	11-12	6
Single	$1.5 \times 10E-45$.. $3.4 \times 10E38$	7- 8	4
Double	$5.0 \times 10E-324$.. $1.7 \times 10E308$	15-16	8
Extended	$1.9 \times 10E-4951$.. $1.1 \times 10E4932$	19-20	10
Comp*	$-2E+63+1$.. $2E+63-1$	19-20	8

* comp only holds integer values.

Get into the Turbo Pascal editor and enter the following program:

```

program DoRatio;
var
  A,B: Integer;
  Ratio: Real;
begin
  Write('Enter two numbers: ');
  Readln(A,B);
  Ratio := A div B;
  Writeln('The ratio is ',Ratio)
end.

```

Save this as DORATIO.PAS by selecting File | Save As from the main menu. Then press *Alt-R* to compile and run the program. Enter two values (such as 10 and 3) and note the result (3.000000). You probably expected an answer of 3.3333333333, but instead you received a 3. That's because you used the **div** operator, which performs integer division. Go back and change the **div** statement to read as follows:

```
Ratio := A / B;
```

Save the code (press *F2*), then compile and run. The result is now 3.3333333333, as you expected. Using the division operator (*/*) gives you the most precise result—a real number.

Character and string data types

You've learned how to store numbers in Pascal, now how about characters and strings? Pascal offers a predefined data type **Char** that is 1 byte in size and holds exactly one character. Character constants are represented by surrounding the character with single quotes (for example, 'A', 'e', '?', '2'). Note that '2' means the

character 2, while 2 means the *integer* 2 (and 2.0 means the *real* number 2).

Here's a modification of DORATIO that allows you to repeat it several times (this also uses a **repeat..until** loop, which we'll discuss a little later):

```
program DoRatio;
var
  A,B: Integer;
  Ratio: Real;
  Ans: Char;
begin
  repeat
    Write('Enter two numbers: ');
    Readln(A,B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio);
    Write('Do it again? (Y/N) ');
    Readln(Ans)
  until UpCase(Ans) = 'N'
end.
```

After calculating the ratio once, the program writes the message

```
Do it again? (Y/N)
```

and waits for you to type in a single character, followed by pressing *Enter*. If you type in a lowercase *n* or an uppercase *N*, the **until** condition is met and the loop ends; otherwise, the program goes back to the **repeat** statement and starts over again.

Appendix B in the Programmer's Guide lists the ASCII codes for all characters.

Note that *n* is *not* the same as *N*. This is because they have different ASCII code values. Characters are represented by the ASCII code: Each character has its own 8-bit number (characters take up 1 byte, remember).

Turbo Pascal gives you two additional ways of representing character constants: with a caret (^) or a number symbol (#). First, the characters with codes 0 through 31 are known as *control characters* (because historically they were used to control teletype operations). They are referred to by their abbreviations (CR for carriage return, LF for linefeed, ESC for escape, and so on) or by the word "Ctrl" followed by a corresponding letter (meaning the letter produced by adding 64 to the control code). For example, the control character with ASCII code 7 is known as BEL or *Ctrl-G*. Turbo Pascal lets you represent these characters using the caret (^), followed by the corresponding letter (or character). Thus, ^G

is a legal representation in your program of *Ctrl-G*, and you could write statements such as *Writeln(^G)*, causing your computer to beep at you. This method, however, only works for the control characters.

You can also represent *any* character using the number symbol (#), followed by the character's ASCII code. Thus, #7 would be the same as ^G, #65 would be the same as 'A', and #233 would represent one of the special IBM PC graphics characters.

Individual characters are nice, but what about strings of characters? After all, that's how you will most often use them. Standard Pascal does not support a separate string data type, but Turbo Pascal does. Take a look at this program:

```
program Hello;
var
  Name: string[30];
begin
  Write('What is your name? ');
  Readln(Name);
  Writeln('Hello, ',Name)
end.
```

This declares the variable *Name* to be of type **string**, with space set aside to hold 30 characters. One more byte is set aside internally by Turbo Pascal to hold the current length of the string. That way, no matter how long or short the name is you enter at the prompt, that is exactly how much is printed out in the *Writeln* statement. Unless, of course, you enter a name more than 30 characters long, in which case only the first 30 characters are used, and the rest are ignored.

When you declare a string variable, you can specify how many characters (up to 255) it can hold. Or you can declare a variable (or parameter) to be of type **String** with no length mentioned, in which case the default size of 255 characters is assumed.

Turbo Pascal offers a number of predefined procedures and functions to use with strings; they can be found in Chapter 1 in the *Library Reference*.

Boolean data type

Pascal's predefined data type **Boolean** has two possible values: **True** and **False**. You can declare a variable to be of type **Boolean**, then assign the variable either a **True** or **False** value or (more

importantly) an expression that resolves to one of those two values.

A *Boolean expression* is simply an expression that is either True or False. It is made up of relational expressions, Boolean operators, Boolean variables, and/or other Boolean expressions. For example, the following **while** statement contains a Boolean expression:

```
while (Index <= Limit) and not Done do ...]
```

The Boolean expression consists of everything between the keywords **while** and **do**, and presumes that *Done* is a variable (or possibly a function) of type Boolean.

Pointer data type

All the data types we've discussed until now hold just that—data. A *pointer* holds a different type of information—an address. A pointer is a variable that contains the address in memory (RAM) where some data is stored, rather than the data itself. In other words, it points to the data, like an address book or an index.

A pointer is usually (but not necessarily) specific to some other data type. Consider the following declarations:

```
type
  Buffer = string[255];
  BufPtr = ^Buffer;
var
  Buf1: Buffer;
  Buf2: BufPtr;
```

The data type *Buffer* is now just another name for **string[255]**, while the type *BufPtr* defines a pointer to a *Buffer*. The variable *Buf1* is of type *Buffer*; it takes up 256 bytes of memory. The variable *Buf2* is of type *BufPtr*; it contains a 32-bit address and takes up only 4 bytes of memory.

Where does *Buf2* point? Nowhere, currently. Before you can use *BufPtr*, you need to set aside (allocate) some memory and store its address in *Buf2*. You do that using the *New* procedure:

```
New(Buf2);
```

Since *Buf2* points to the type *Buffer*, this statement creates a 256-byte buffer somewhere in memory, then puts its address into *Buf2*.

How do you use the data pointed to by *Buf2*? Via the indirection operator `^`. For example, suppose you want to store a string in both *Buf1* and the buffer pointed to by *Buf2*. Here's what the statements would look like:

```
Buf1 := 'This string gets stored in Buf1.'  
Buf2^ := 'This string gets stored where Buf2 points.'
```

Note the difference between *Buf2* and *Buf2^*. *Buf2* refers to a 4-byte pointer variable; *Buf2^* refers to a 256-byte string variable whose address is stored in *Buf2*.

How do you free up the memory pointed to by *Buf2*? Using the *Dispose* procedure. *Dispose* makes the memory available for other uses. After you use *Dispose* on a pointer, it's good practice to assign the (predefined) value `nil` to that pointer. That lets you know that the pointer no longer points to anything:

```
Dispose (Buf2);  
Buf2 := nil;
```

Note that you assign `nil` to *Buf2*, not to *Buf2^*.

Identifiers

Up until now, we've given names to variables without worrying about what restrictions there might be. Let's talk about those restrictions now.

The names you give to constants, data types, variables, and functions are known as *identifiers*. Some of the identifiers used so far include

<i>Integer, Real, String</i>	Predefined data types
<i>Hello, DoSum, DoRatio</i>	Programs
<i>Name, A, B, Sum, Ratio</i>	User-defined variables
<i>Write, Writeln, Readln</i>	Predeclared procedures

Turbo Pascal has a few rules about identifiers; here's a quick summary:

- All identifiers must start with a letter or underscore (*a...z*, *A...Z*, or *_*). The rest of an identifier can consist of letters, underscores, and/or digits (*0...9*); no other characters are allowed.

- Identifiers are *case-insensitive*, which means that lowercase letters (*a...z*) are considered the same as uppercase letters (*A...Z*). For example, the identifiers *indx*, *Indx*, and *INDX* are identical.
- Identifiers can be of any length, but only the first 63 characters are significant.

Operators

Once you get your data into the program (and into your variables), you'll probably want to manipulate it somehow, using the operators available to you. There are eight operator types: assignment, arithmetic, bitwise, relational, logical, address, set, and string.

Most Pascal operators are *binary*, taking two operands; the rest are *unary*, taking only one operand. Binary operators use the usual algebraic form, for example, $a + b$. A unary operator always precedes its operand, for example, $-b$.

In more complex expressions, rules of precedence clarify the order in which operations are performed (see Table 2.3).

Table 2.3
Operator precedence

Operators	Precedence	Categories
@, not	First (high)	Unary operators
*, /, div, mod, and, shl, shr	Second	Multiplying operators
+, -, or, xor	Third	Adding operators
=, <>, <, >, <=, >=, in	Fourth (low)	Relational operators

Operations with equal precedence are normally performed from left to right, although the compiler may at times rearrange the operands to generate optimum code.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators.

Assignment operators

The most basic operation is *assignment* (that is, assigning a value to a variable), as in $Ratio := A / B$. In Pascal, the assignment symbol is a colon followed by an equal sign ($:=$). In the example given, the value of A / B on the right of the assignment symbol is assigned to the variable *Ratio* on the left.

Arithmetic operators

Pascal supports the usual set of binary arithmetic operators—they work with type integer and real values:

- Multiplication (*)
- Integer division (**div**)
- Real division (/)
- Modulus (**mod**)
- Addition (+)
- Subtraction (-)

Also, Turbo Pascal supports both *unary minus* ($a + (-b)$), which performs a *two's complement* evaluation, and *unary plus* ($a + (+b)$), which does nothing at all but is there for completeness.

Bitwise operators

For bit-level operations, Pascal has the following operators:

- **shl** (shift left): Shifts the bits left the indicated number of bits, filling at the right with 0's.
- **shr** (shift right): Shifts the bits right the indicated number of bits, filling at the left with 0's.
- **and**: Performs a logical **and** on each corresponding pair of bits, returning 1 if both bits are 1, and 0 otherwise.
- **or**: Performs a logical **or** on each corresponding pair of bits, returning 0 if both bits are 0, and 1 otherwise.
- **xor**: Performs a logical, exclusive **or** on each corresponding pair of bits, returning 1 if the two bits are different from one another, and 0 otherwise.
- **not**: Performs a logical complement on each bit, changing each 0 to a 1, and vice versa.

These allow you to perform very low-level operations on integer values.

Relational operators

Relational operators allow you to compare two values, yielding a Boolean result of True or False. Here are the relational operators in Pascal:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to
in	is a member of

Why would you want to know if something were True or False?
Enter the following program:

```

program TestGreater;
var
  A,B: Integer;
  Test: Boolean;
begin
  Write('Enter two numbers: ');
  Readln(A,B);
  Test := A > B;
  Writeln('A is greater than B', Test);
end.

```

This will print True if *A* is greater than *B* or False if *A* is less than or equal to *B*.

Logical operators

There are four logical operators—**and**, **xor**, **or**, and **not**—which are similar to but not identical with the bitwise operators. These logical operators work with logical values (True and False), allowing you to combine relational expressions, Boolean variables, and Boolean expressions.

They differ from the corresponding bitwise operators in this manner:

- Logical operators always produce a result of either True or False (a Boolean value), while the bitwise operators do bit-by-bit operations on type integer values.
- You cannot combine Boolean and integer-type expressions with these operators; in other words, the expression *Flag and Indx* is illegal if *Flag* is of type Boolean, and *Indx* is of type integer (or vice versa).
- The logical operators **and** and **or** will short-circuit by default; **xor** and **not** will not. Suppose you have the expression *exp1 and exp2*. If *exp1* is False, then the entire expression is False, so *exp2* will never be evaluated. Likewise, given the expression *exp1 or*

exp2, *exp2* will never be evaluated if *exp1* is True. You can force full Boolean expression using the **{\$B+}** compiler directive or the Complete Boolean Eval option (**Options | Compiler**).

Address operators

Pascal supports two special address operators: the *address-of* operator (@) and the *indirection* operator (^).

The @ operator returns the address of a given variable; if *Sum* is a variable of type integer, then @*Sum* is the address (memory location) of that variable. Likewise, if *ChrPtr* is a pointer to type char, then *ChrPtr*^ is the character to which *ChrPtr* points.

Set operators

Set operators perform according to the rules of set logic. The set operators and operations include

+	union
-	difference
*	intersection

String operators

The only string operation is the + operator, which is used to concatenate two strings.

Output

It may seem funny to talk about output before input, but a program that doesn't output information isn't of much use. That output usually takes the form of information written to the screen (words and pictures), to a storage device (floppy or hard disk), or to an I/O port (serial or printer ports).

The Writeln procedure

You've already used the most common output function in Pascal, the *Writeln* routine. The purpose of *Writeln* is to write information to the screen. Its format is both simple and flexible:

```
Writeln(item, item, ...);
```

Each *item* is something you want to print to the screen and can be a literal value, such as an integer or a real number (3, 42, -1732.3), a character ('a', 'Z'), a string ('Hello, world'), or a Boolean value (True). It can also be a named constant, a variable, a dereferenced pointer, or a function call, as long as it yields a value that is of type Integer, Real, Char, String, or Boolean. All the items are printed on one line, in the order given. The cursor is then moved to the start of the next line. If you wish to leave the cursor after the last item on the same line, then use the statement

```
Write(item,item,...);
```

When the items in a *Writeln* statement are printed, blanks are *not* automatically inserted; if you want spaces between items, you'll have to put them there yourself, like this:

```
Writeln(item, ' ', item, ' ',...);
```

For example, the following statements produce the indicated output:

```
A := 1; B := 2; C := 3;
Name := 'Frank';
  Writeln(A,B,C);                123
  Writeln(A, ' ', B, ' ', C);    . 1 2 3
  Writeln('Hi',Name);           HiFrank
  Writeln('Hi, ',Name, '.');     Hi, Frank.
```

You can also use *field-width specifiers* to define a field width for a given item. The format for this is

```
Writeln(item:width,...);
```

where *width* is an integer expression (literal, constant, variable, function call, or combination thereof) specifying the total width of the field in which *item* is written. For example, consider the following code and resulting output:

```
A := 10; B := 2; C := 100;
Writeln(A,B,C);                102100
Writeln(A:2,B:2,C:2);          10 2100
Writeln(A:3,B:3,C:3);          10 2100
Writeln(A,B:2,C:4);            10 2 100
```

Note that the item is padded with leading blanks on the left to make it equal to the field width. The actual value is right-justified.

What if the field width is less than what is needed? In the second *Writeln* statement given earlier, *C* has a field width of 2 but has a

value of 100 and needs a width of 3. As you can see by the output, Pascal simply expands the width to the minimum size needed.

This method works for all allowable items: integers, reals, characters, strings, and Booleans. However, real numbers printed with the field-width specifier (or with none at all) come out in exponential form:

```
X := 421.53;
Writeln(X);           4.2153000000E+02
Writeln(X:8);        4.2E+02
```

Because of this, Pascal allows you to append a second field-width specifier: *item:width:digits*. This second value forces the real number to be printed out in fixed-point format and tells how many digits to place after the decimal point:

```
X := 421.53;
Writeln(X:6:2);      421.53
Writeln(X:8:2);      421.53
Writeln(X:8:4);      421.5300
```

Input

Standard Pascal has two basic input functions that are used to read from data from the keyboard: *Read* and *Readln*. The general syntax is

```
Read(item,item,...);
```

or

```
Readln(item,item,...);
```

Each *item* is a variable of any integer, real, char, or string type. Numbers being input must be separated from other values by spaces or by pressing *Enter*.

Conditional statements

There are times when you want to execute some portion of your program when a given condition is True or False, or when a particular value of a given expression is reached. Let's look at how to do this in Pascal.

The if statement

Look again at the **if** statement in the previous examples; note that it can take the following generic format:

```
if expr
  then statement1
  else statement2
```

expr is any Boolean expression (resolving to True or False), and *statement1* and *statement2* are legal Pascal statements. If *expr* is True, then *statement1* is executed; otherwise, *statement2* is executed.

We must explain two important points about **if/then/else** statements:

1. **else statement2** is optional; in other words, this is a valid **if** statement:

```
if expr
  then statement1
```

In this case, *statement1* is executed only if *expr* is True. If *expr* is False, then *statement1* is skipped, and the program continues.

2. What if you want to execute more than one statement if a particular expression is True or False? You use a compound statement. A *compound statement* consists of the keyword **begin**, some number of statements separated by semicolons (;), and the keyword **end**.

The ratio example uses a single statement for the **if** clause

```
if B = 0.0 then
  Writeln('Division by zero is not allowed.')
```

and a compound statement for the **else** clause

```
else
begin
  Ratio = A / B;
  Writeln('The ratio is ',Ratio)
end;
```

You might also notice that the body of each program you've written is simply a compound statement followed by a period.

The case statement

else is an extension to standard Pascal.

This statement gives your program the power to choose from more than two alternatives without having to specify lots of **if** statements.

The **case** statement consists of an expression (the selector) and a list of statements, each preceded by a **case** label of the same type as the selector. It specifies that the one statement be executed whose **case** label is equal to the current value of the selector. If none of the **case** labels contain the value of the selector, then either no statement is executed or, optionally, the statements following the reserved word **else** are executed.

A **case** label consists of any number of constants or subranges, separated by commas and followed by a colon; for example,

```
case BirdSight of
  'C', 'c': Curlews := Curlews + 1;
  'H', 'h': Herons  := Herons + 1;
  'E', 'e': Egrets  := Egrets + 1;
  'T', 't': Terns   := Terns + 1;
end; { case }
```

A subrange is written as two constants separated by the subrange delimiter '..'. The constant type must match the selector type. The statement that follows the **case** label is executed if the selector's value equals one of the constants or if it lies within one of the subranges.

Loops

Just as there are statements (or groups of statements) that you want to execute conditionally, there are other statements that you may want to execute repeatedly. This kind of construct is known as a *loop*.

There are three basic kinds of loops: the **while** loop, the **repeat** loop, and the **for** loop. We'll cover them in that order.

The while loop

You can use the **while** loop to test for something at the beginning of your loop. Enter the following program:

```
program Hello;
var
  Count: Integer;
begin
  Count := 1;
  while Count <= 10 do
  begin
    Writeln('Hello and goodbye!');
    Inc(Count)
  end;
  Writeln('This is the end!')
end.
```

The first thing that happens when you run this program is that *Count* is set equal to 1, then you enter the **while** loop. This tests to see if *Count* is less than or equal to 10. *Count* is, so the loop's body (**begin..end**) is executed. This prints the message *Hello and goodbye!* to the screen, then increments *Count* by 1. *Count* is again tested, and the loop's body is executed once more. This continues as long as *Count* is less than or equal to 10 when it is tested. Once *Count* reaches 11, the loop stops, and the string *This is the end!* is printed on the screen.

The format of the **while** statement is

```
while expr do statement
```

where *expr* is a Boolean expression, and *statement* is either a single or a compound statement.

The **while** loop evaluates *expr*. If it's True, then *statement* is executed, and *expr* is evaluated again. If *expr* is False, the **while** loop is finished and the program continues.

The repeat..until loop

The second loop is the **repeat..until** loop, which we've seen in the program DORATIO.PAS:

```
program DoRatio;
var
  A,B: Integer;
```

```

Ratio: Real;
Ans: Char;
begin
  repeat
    Write('Enter two numbers: ');
    Readln(A,B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio);
    Write('Do it again? (Y/N) ');
    Readln(Ans)
  until UpCase(Ans) = 'N'
end.

```

As described before, this program repeats until you answer *n* or *N* to the question Do it again? (Y/N). In other words, everything between **repeat** and **until** is repeated until the expression following **until** is True.

Here's the generic format for the **repeat..until** loop:

```

repeat
  statement;
  statement;
  ...
  statement
until expr

```

There are three major differences between the **while** loop and the **repeat** loop. First, the statements in the **repeat** loop always execute at least once, because the test on *expr* is not made until after the **repeat** occurs. By contrast, the **while** loop will skip over its body if the expression is initially False.

Next, the **repeat** loop executes *until* the expression is True, where the **while** loop executes *while* the expression is True. This means that care must be taken in translating from one type of loop to the other. For example, here's the HELLO program rewritten using a **repeat** loop:

```

program Hello;
var
  Count: Integer;
begin
  Count := 1;
  repeat
    Writeln('Hello and goodbye!');
    Inc(Count)
  until Count > 10;

```

```
    Writeln('This is the end!')
end.
```

Note that the test is now *Count* > 10, where for the **while** loop it was *Count* <= 10.

Finally, the **repeat** loop can hold multiple statements without using a compound statement. Notice that you didn't have to use **begin..end** in the preceding program, but you did for the earlier version using a **while** loop.

Again, be careful to note that the **repeat** loop will always execute at least once. A **while** loop may or may not execute, depending on the value of the expression.

The for loop

The **for** loop is the one found in most major programming languages, including Pascal. However, the Pascal version is simultaneously limited and powerful.

Basically, the **for** loop executes a set of statements some fixed number of times while a variable (known as the *index variable*) steps through a range of values. To see how this works, modify the earlier HELLO program to read as follows:

```
program Hello;
var
  Count: Integer;
begin
  for Count := 1 to 10 do
    Writeln('Hello and goodbye!');
    Writeln('This is the end!')
end.
```

When you run this program, you can see that the loop works the same as the **while** and **repeat** loops already shown and, in fact, is precisely equivalent to the **while** loop. Here's the generic format of the **for** loop statement:

```
for index := expr1 to expr2 do statement
```

index is a variable of some scalar type (any integer type, char, Boolean, any enumerated type), *expr1* and *expr2* are expressions of some type compatible with *index*, and *statement* is a single or compound statement. *Index* is incremented by one after each time through the loop.

You can also decrement the index variable instead of incrementing it by replacing the keyword **to** with the keyword **downto**.

The **for** loop is equivalent to the following code:

```
index := expr1;
while index <= expr2 do
begin
    statement;
    Inc(index)
end;
```

The main drawback of the **for** loop is that it only allows you to increment or decrement by one. Its main advantages are conciseness and the ability to use char and enumerated types in the range of values.

Procedures and functions

You've learned how to execute code conditionally and iteratively. Now, what if you want to perform the same set of instructions on different sets of data or at different locations in your program? Well, you simply put those statements into a *subroutine*, which you can then call as needed.

In Pascal, there are two types of subroutines: *procedures* and *functions*. The main difference between the two is that a function returns a value and can be used in expressions, like this:

```
X := Sin(A);
```

while a procedure is called to perform one or more tasks:

```
Writeln('This is a test');
```

However, before you learn any more about procedures and functions, you need to understand Pascal program structure.

Program structure

In Standard Pascal, programs adhere to a rigid format:

```
program ProgName;
label
    labels;
```

```

const
    constant declarations;
type
    data type definitions;
var
    variable declarations;
procedures and functions;
begin
    main body of program
end.

```

You do not have to have all five declaration sections—**label**, **const**, **type**, **var**, and **procedures and functions**—in every program. But in standard Pascal, if they do appear, they must be in that order, and each section can appear only once. The declaration section is followed by any procedures and functions you might have, then finally the main body of the program, consisting of some number of statements.

Turbo Pascal gives you tremendous flexibility in your program structure. All it requires is that your program statement (if you have one) be first and that your main program body be last. Between those two, you can have as many declaration sections as you want, in any order you want, with procedures and functions freely mixed in. But identifiers must be defined before they are used, or else a compile-time error will occur.

Procedure and function structure

As mentioned earlier, procedures and functions—known collectively as *subprograms*—appear anywhere before the main body of the program. Procedures use this format:

```

procedure ProcName (parameters) ;
label
    labels;
const
    constant declarations;
type
    data type definitions;
var
    variable declarations;
procedures and functions;
begin
    main body of procedure;
end;

```

Functions look just like procedures except that a function declaration starts with a **function** header and ends with a data type for the return value of the function:

```
function FuncName(parameters): data type;
```

As you can see, there are only two differences between this and regular program structure: Procedures or functions start with a **procedure** or **function** header instead of a **program** header, and they end with a semicolon instead of a period. A procedure or function can have its own constants, data types, and variables, and even its own procedures and functions. What's more, all these items can only be used with the procedure or function in which they are declared.

Sample program

Here's a version of the DORATIO program that uses a procedure to get the two values, then uses a function to calculate the ratio:

```
program DoRatio;
var
  A,B: Integer;
  Ratio: Real;
procedure GetData(var X,Y: Integer);
begin
  Write('Enter two numbers: ');
  Readln(X,Y)
end;

function GetRatio(I,J: Integer): Real;
begin
  GetRatio := I/J
end;

begin
  GetData(A,B);
  Ratio := GetRatio(A,B);
  Writeln('The ratio is ',Ratio)
end.
```

This isn't exactly an improvement on the original program, being both larger and slower, but it does illustrate how procedures and functions work.

When you compile and run this program, execution starts with the first statement in the main body of the program: `GetData(A,B)`. This type of statement is known as a *procedure call*. Your program handles this call by executing the statements in *GetData*, replacing

X and Y (known as *formal parameters*) with A and B (known as *actual parameters*). The keyword **var** in front of X and Y in *GetData*'s procedure statement says that the actual parameters must be variables and that the variable values can be changed and passed back to the caller. So you can't pass literals, constants, expressions, and so on to *GetData*. Once *GetData* is finished, execution returns to the main body of the program and continues with the statement following the call to *GetData*.

That next statement is a function call to *GetRatio*. Note that there are some key differences here. First, *GetRatio* returns a value, which must then be used somehow; in this case, it's assigned to *Ratio*. Second, a value is assigned to *GetRatio* in its main body; this is how a function determines what value to return. Third, there is no **var** keyword in front of the formal parameters I and J. This means that the actual parameters could be any two integer expressions, such as *Ratio := GetRatio(A + B, 300)*; and that even if you change the values of the formal parameters in the **function** body, the new values will not be passed back to the caller. This, by the way, is *not* a distinction between procedures and functions; you can use both types of parameters with either type of subprogram.

Program comments

Sometimes you want to insert notes into your program to remind yourself (or inform someone else) of what certain variables mean, what certain functions or statements do, and so on. These notes are known as *comments*. Pascal, like most other programming languages, lets you put as many comments as you want into your program.

You can start a comment with the left curly brace (`{`), which signals to the compiler to ignore everything until after it sees the right curly brace (`}`).

Comments can even extend across multiple lines, like this:

```
{ This is a long
  comment, extending
  over several lines. }
```

Pascal also allows an alternative form of comment, beginning with a left parenthesis and an asterisk, (`*`), and ending with an asterisk and a right parenthesis, (`*`). This allows for a limited form

of comment nesting, because a comment beginning with `(*` ignores all curly braces, and vice versa.

Turbo Pascal units

In Chapter 1, you learned how to write standard Pascal programs. What about non-standard programming—more specifically, PC-style programming, with screen control, DOS calls, and graphics? To write such programs, you have to understand units or understand the PC hardware enough to do the work yourself. This chapter explains what a unit is, how you use it, what predefined units are available, how to go about writing your own units, and how to compile them.

What is a unit?

Turbo Pascal gives you access to a large number of predefined constants, data types, variables, procedures, and functions. Some are specific to Turbo Pascal; others are specific to the IBM PC (and compatibles) or to DOS. There are dozens of them, but you seldom use them all in a given program. Because of this, they are split into related groups called *units*. You can then use only the units your program needs.

A *unit* is a collection of constants, data types, variables, procedures, and functions. Each unit is almost like a separate Pascal program: It can have a main body that is called before your program starts and does whatever initialization is necessary. In short, a unit is a library of declarations you can pull into your program that allows your program to be split up and separately compiled.

All the declarations within a unit are usually related to one another. For example, the *Crt* unit contains all the declarations for screen-oriented routines on your PC.

Turbo Vision provides an entire suite of units; see the Turbo Vision Guide for details.

Turbo Pascal provides eight standard units for your use. Six of them—*System*, *Overlay*, *Graph*, *Dos*, *Crt*, and *Printer*—provide support for your regular Turbo Pascal programs; these are all stored in *TURBO.TPL*. The other two—*Turbo3* and *Graph3*—are designed to help maintain compatibility with programs and data files created under version 3.0 of Turbo Pascal. Some of these are explained more fully in Chapters 10 through 15 of the *Programmer's Guide*, but we'll look at each one here and explain its general function.

A unit's structure

A unit provides a set of capabilities through procedures and functions—with supporting constants, data types, and variables—but it hides how those capabilities are actually implemented by separating the unit into two sections: the *interface* and the *implementation*. When a program uses a unit, all the unit's declarations become available, as if they had been defined within the program itself.

A unit's structure is not unlike that of a program, but with some significant differences. Here's a unit, for example:

```
unit <identifier>;
interface
uses <list of units>; { Optional }
  { public declarations }
implementation
uses <list of units>; { Optional }
  { private declarations }
  { implementation of procedures and functions }
begin
  { initialization code }
end.
```

The unit header starts with the reserved word **unit**, followed by the unit's name (an identifier), much the way a program begins. The next item in a unit is the keyword **interface**. This signals the start of the interface section of the unit—the section visible to any other units or programs that use this unit.

A unit can use other units by specifying them in a **uses** clause. The **uses** clause can appear in two places. First, it can appear immediately after the keyword **interface**. In this case, any constants or data types declared in the interfaces of those units can be used in any of the declarations in this unit's interface section.

Second, it can appear immediately after the keyword **implementation**. In this case, any declarations from those units can be used only within the implementation section. This also allows for *circular unit references*; you'll learn how to use these later in this chapter.

Interface section

The interface portion—the “public” part—of a unit starts at the reserved word **interface**, which appears after the unit header and ends when the reserved word **implementation** is encountered. The interface determines what is “visible” to any program (or other unit) using that unit; any program using the unit has access to these “visible” items.

In the unit interface, you can declare constants, data types, variables, procedures, and functions. As with a program, these can be arranged in any order, and sections can repeat themselves (for example, **type ... var ... <procs> ... const ... type ... const ... var**).

The procedures and functions visible to any program using the unit are declared here, but their actual bodies—implementations—are found in the implementation section. **forward** declarations are neither necessary nor allowed. The bodies of all the regular procedures and functions are held in the implementation section after all the procedure and function headers have been listed in the interface section.

A **uses** clause may appear in the implementation. If present, **uses** must immediately follow the keyword **implementation**.

Implementation section

The implementation section—the “private” part—starts at the reserved word **implementation**. Everything declared in the interface portion is visible in the implementation: constants, types, variables, procedures, and functions. Furthermore, the implementation can have additional declarations of its own,

although these are not visible to any programs using the unit. The program doesn't know they exist and can't reference or call them. However, these hidden items can be (and usually are) used by the "visible" procedures and functions—those routines whose headers appear in the interface section.

A **uses** clause may appear in the implementation. If present, **uses** must immediately follow the keyword implementation.

If any procedures have been declared external, one or more **(\$L filename)** directive(s) should appear anywhere in the source file before the final **end** of the unit.

The normal procedures and functions declared in the interface—those that are not inline—must reappear in the implementation. The **procedure/function** header that appears in the implementation should either be identical to that which appears in the interface or should be in the short form. For the short form, type in the keyword (**procedure** or **function**), followed by the routine's name (identifier). The routine will then contain all its local declarations (labels, constants, types, variables, and nested procedures and functions), followed by the main body of the routine itself. Say the following declarations appear in the interface of your unit:

```
procedure ISwap(var V1,V2: Integer);
function IMax(V1,V2: Integer): Integer;
```

The implementation could look like this:

```
procedure ISwap;
var
  Temp: Integer;
begin
  Temp := V1; V1 := V2; V2 := Temp;
end; { of proc ISwap }
function IMax(V1,V2: Integer): Integer;
begin
  if V1 > V2 then
    IMax := V1
  else IMax := V2;
end; { of func IMax }
```

Routines local to the implementation (that is, not declared in the interface section) must have their complete **procedure/function** header intact.

Initialization section

The entire implementation portion of the unit is normally bracketed within the reserved words **implementation** and **end**. However, if you put the reserved word **begin** before **end**, with statements between the two, the resulting compound statement—looking very much like the main body of a program—becomes the **initialization** section of the unit.

The initialization section is where you initialize any data structures (variables) that the unit uses or makes available (through the interface) to the program using it. You can use it to open files for the program to use later. For example, the standard unit *Printer* uses its initialization section to make all the calls to open (for output) the text file *Lst*, which you can then use in your program's *Write* and *Writeln* statements.

When a program using that unit is executed, the unit's initialization section is called before the program's main body is run. If the program uses more than one unit, each unit's initialization section is called (in the order specified in the program's **uses** statement) before the program's main body is executed.

How are units used?

The units your program uses have already been compiled and stored as machine code, not Pascal source code; they are not Include files. Even the interface section is stored in the special binary symbol table format that Turbo Pascal uses. Furthermore, certain standard units are stored in a special file (TURBO.TPL) and are automatically loaded into memory along with Turbo Pascal itself.

As a result, using a unit or several units adds very little time (typically less than a second) to the length of your program's compilation. If the units are being loaded in from a separate disk file, a few additional seconds may be required because of the time it takes to read from the disk.

As stated earlier, to use a specific unit or collection of units, you must place a **uses** clause at the start of your program, followed by a list of the unit names you want to use, separated by commas:

```
program MyProg;
uses thisUnit,thatUnit,theOtherUnit;
```

When the compiler sees this **uses** clause, it adds the interface information in each unit to the symbol table and links the machine code that is the implementation to the program itself.

The ordering of units in the **uses** clause is not important. If *thisUnit* uses *thatUnit* or vice versa, you can declare them in either order, and the compiler will determine which unit must be linked into MyProg first. In fact, if *thisUnit* uses *thatUnit* but MyProg doesn't need to directly call any of the routines in *thatUnit*, you can "hide" the routines in *thatUnit* by omitting it from the **uses** clause:

```
unit thisUnit;
uses thatUnit;
...
program MyProg;
uses thisUnit, theOtherUnit;
...
```

In this example, *thisUnit* can call any of the routines in *thatUnit*, and MyProg can call any of the routines in *thisUnit* or *theOtherUnit*. MyProg cannot, however, call any of the routines in *thatUnit* because *thatUnit* does not appear in MyProg's **uses** clause.

If you don't put a **uses** clause in your program, Turbo Pascal links in the *System* standard unit anyway. This unit provides some of the standard Pascal routines as well as a number of Turbo Pascal-specific routines.

Referencing unit declarations

Once you include a unit in your program, all the constants, data types, variables, procedures, and functions declared in that unit's interface become available to you. For example, suppose the following unit existed:

```
unit MyStuff;
interface
const
  MyValue = 915;
type
  MyStars = (Deneb,Antares,Betelgeuse);
var
  MyWord: string[20];
```

```

procedure SetMyWord(Star: MyStars);
function TheAnswer: Integer;
implementation
...
end.

```

What you see here is the unit's interface, the portion that is visible to (and used by) your program. Given this, you might write the following program:

```

program TestStuff;
uses MyStuff;
var
  I: Integer;
  AStar: MyStars;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.

```

Now that you have included the statement **uses** *MyStuff* in your program, you can refer to all the identifiers declared in the interface section in the interface of *MyStuff* (*MyWord*, *MyValue*, and so on). However, consider the following situation:

```

program TestStuff;
uses MyStuff;
const
  MyValue = 22;
var
  I: Integer;
  AStar: MyStars;
function TheAnswer: Integer;
begin
  TheAnswer := -1
end;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.

```


This program redefines some of the identifiers declared in *MyStuff*. It will compile and run, but will use its own definitions for *MyValue* and *TheAnswer*, since those were declared more recently than the ones in *MyStuff*.

You're probably wondering whether there's some way in this situation to still refer to the identifiers in *MyStuff*? Yes, preface each one with the identifier *MyStuff* and a period (.). For example, here's yet another version of the earlier program:

```
program TestStuff;  
uses MyStuff;  
const  
    MyValue = 22;  
var  
    I: Integer;  
    AStar: MyStars;  
function TheAnswer: Integer;  
begin  
    TheAnswer := -1;  
end;  
begin  
    Writeln(MyStuff.MyValue);  
    AStar := Deneb;  
    SetMyWord(AStar);  
    Writeln(MyWord);  
    I := MyStuff.TheAnswer;  
    Writeln(I);  
end.
```

This program will give you the same answers as the first one, even though you've redefined *MyValue* and *TheAnswer*. Indeed, it would have been perfectly legal (although rather wordy) to write the first program as follows:

```
program TestStuff;  
uses MyStuff;  
var  
    I: Integer;  
    AStar: MyStuff.MyStars;  
begin  
    Writeln(MyStuff.MyValue);  
    AStar := MyStuff.Deneb;  
    MyStuff.SetMyWord(AStar);  
    Writeln(MyStuff.MyWord);  
    I := MyStuff.TheAnswer;  
    Writeln(I);  
end.
```

Note that you can preface any identifier—constant, data type, variable, or subprogram—with the unit name.

Implementation section uses clause

As of version 5.0, Turbo Pascal allows you to place a **uses** clause in a unit's implementation section. If present, the **uses** clause must immediately follow the **implementation** keyword, just like a **uses** clause in the interface section must immediately follow the **interface** keyword.

A **uses** clause in the implementation section allows you to further hide the inner details of a unit, since units used in the **implementation** section are not visible to users of the unit. More importantly, however, it also enables you to construct mutually dependent units.

Since units in Turbo Pascal need not be strictly hierarchical, you can make circular unit references. The next section provides an example that demonstrates the usefulness of circular references.

Circular unit references

The following program demonstrates how two units can “use” each other. The main program, *Circular*, uses a unit named *Display*. *Display* contains one routine in its interface section, *WriteXY*, which takes three parameters: an (x, y) coordinate pair and a text message to display. If the (x, y) coordinates are onscreen, *WriteXY* moves the cursor to (x, y) and displays the message there; otherwise, it calls a simple error-handling routine.

So far, there's nothing fancy here—*WriteXY* is taking the place of *Write*. Here's where the circular unit reference enters in: How is the error-handling routine going to display its error message? By using *WriteXY* again. Thus you have *WriteXY*, which calls the error-handling routine *ShowError*, which in turn calls *WriteXY* to put an error message onscreen. If your head is spinning in circles, let's look at the source code to this example, so you can see that it's really not that tricky.

The main program, *Circular*, clears the screen and makes three calls to *WriteXY*:

```
program Circular;  
  { Display text using WriteXY }  
  
uses  
  Crt, Display;
```

```

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(100, 100, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.

```

Look at the (x, y) coordinates of the second call to *WriteXY*. It's hard to display text at (100, 100) on an 80x25 line screen. Next, let's see how *WriteXY* works. Here's the source to the *Display* unit, which contains the *WriteXY* procedure. If the (x, y) coordinates are valid, it displays the message; otherwise, *WriteXY* displays an error message:

```

unit Display;
{ Contains a simple video display routine }

interface

procedure WriteXY(X, Y: Integer; Message: String);

implementation
uses
  Crt, Error;

procedure WriteXY(X, Y: Integer; Message: String);
begin
  if (X in [1..80]) and (Y in [1..25]) then
    begin
      GoToXY(X, Y);
      Write(Message);
    end
  else
    ShowError('Invalid WriteXY coordinates');
  end;
end.

```

The *ShowError* procedure called by *WriteXY* is declared in the following code in the *Error* unit. *ShowError* always displays its error message on the 25th line of the screen:

```

unit Error;
{ Contains a simple error-reporting routine }

interface

procedure ShowError(ErrMsg: String);

implementation
uses
  Display;

```

```

procedure ShowError(ErrMsg: String);
begin
  WriteXY(1, 25, 'Error: ' + ErrMsg);
end;

end.

```

Notice that the **uses** clause in the **implementation** sections of both *Display* and *Error* refer to each other. These two units can refer to each other in their **implementation** sections because Turbo Pascal can compile complete **interface** sections for both. In other words, the Turbo Pascal compiler will accept a reference to partially compiled unit *A* in the **implementation** section of unit *B*, as long as both *A* and *B*'s **interface** sections do not depend upon each other (and thus follow Pascal's strict rules for declaration order).

Sharing other declarations

What if you want to modify *WriteXY* and *ShowError* to take an additional parameter that specifies a rectangular window onscreen:

```

procedure WriteXY(SomeWindow: WindRec; X, Y: Integer;
  Message: String);

procedure ShowError(SomeWindow: WindRec; ErrMsg: String);

```

Remember these two procedures are in separate units. Even if you declared *WindData* in the **interface** of one, there would be no legal way to make that declaration available to the **interface** of the other. The solution is to declare a third module that contains only the definition of the window record:

```

unit WindData;
interface
type
  WindRec = record
    X1, Y1, X2, Y2: Integer;
    ForeColor, BackColor: Byte;
    Active: Boolean;
  end;
implementation
end.

```

In addition to modifying the code to *WriteXY* and *ShowError* to make use of the new parameter, the **interface** sections of both the *Display* and *Error* units can now "use" *WindData*. This approach is legal because unit *WindData* has no dependencies in its **uses** clause, and units *Display* and *Error* refer to each other only in their respective **implementation** sections.

The standard units

The file TURBO.TPL contains all the standard units except *Graph* and the compatibility units (*Graph3* and *Turbo3*): *System*, *Overlay*, *Crt*, *Dos*, and *Printer*. These are units loaded into memory with Turbo Pascal; they're always readily available to you. You will normally keep the file TURBO.TPL in the same directory as TURBO.EXE (or TPC.EXE).

System

System contains all the standard and built-in procedures and functions of Turbo Pascal. Every Turbo Pascal routine that is *not* part of standard Pascal and that is *not* in one of the other units is in *System*. This unit is always linked into every program. The details of the *System* unit are described in Chapter 10 of the *Programmer's Guide*, "The System unit."

Dos

Dos defines numerous Pascal procedures and functions that are equivalent to the most commonly used DOS calls, such as *Exec*, *GetTime*, *SetTime*, *DiskSize*, and so on. It also defines two low-level routines, *MsDos* and *Intr*, which allow you to directly invoke any MS-DOS call or system interrupt. *Registers* is the data type for the parameter to *MsDos* and *Intr*. Some other constants and data types are also defined.

The *Dos* unit is discussed in detail in Chapter 11 of the *Programmer's Guide*, "The Dos unit."

Overlay

Overlay provides support for Turbo Pascal's powerful overlay system. Overlays are discussed in detail in Chapter 13 of the *Programmer's Guide*, "The Overlay unit."

Crt

Crt provides a set of PC-specific declarations for input and output: constants, variables, and routines. You can use these to manipulate your text screen (do windowing, direct cursor addressing, text color and background). You can also do "raw" input from the keyboard and control the PC's sound chip. *Crt* is

described in detail in Chapter 15 of the *Programmer's Guide*, "The Crt unit."

Printer

Printer declares the text-file variable *Lst* and connects it to a device driver that allows you to send standard Pascal output to the printer using *Write* and *Writeln*. For example, once you include *Printer* in your program, you could do the following:

```
Write(Lst, 'The sum of ', A:4, ' and ', B:4, ' is ');  
C := A + B;  
Writeln(Lst, C:8);
```

Graph

The *Graph* unit is not built into TURBO.TPL, but instead resides on the same disk with the .BGI and .CHR support files. Place GRAPH.TPU in the current directory or use the unit directory to specify the full path to GRAPH.TPU. (If you have a hard disk and you used the INSTALL program, your system is already set up so you can use *Graph*.)

Graph supplies a set of fast, powerful graphics routines that allow you to make full use of the graphics capabilities of your PC. It implements the device-independent Borland graphics handler, allowing support of CGA, EGA, Hercules, AT &T 400, MCGA, 3270 PC, and VGA and 8514 graphics.

Further explanations of *Graph* and the Borland Graphic Interface (BGI) may be found in Chapter 12 of the *Programmer's Guide*, "The Graph unit and the BGI."

Turbo3 and Graph3

These units are provided for backward compatibility only. *Turbo3* contains two variables and several procedures no longer supported by Turbo Pascal. *Graph3* supports the full set of graphics routines—basic, advanced, and turtlegraphics—from version 3.0. Full information on these units is included in the online file TURBO3.INT.

Now that you've been introduced to units, let's see about writing your own.

Writing your own units

Say you've written a unit called *IntLib*, stored it in a file called INTLIB.PAS, and compiled it to disk; the resulting code file will be called INTLIB.TPU. To use it in your program, you must include a **uses** statement to tell the compiler you're using that unit. Your program might look like this:

```
program MyProg;  
uses IntLib;
```

Note that Turbo Pascal expects the unit code file to have the same name (up to eight characters) of the unit itself. If your unit name is *MyUtilities*, then Turbo is going to look for a file called MYUTILIT.PAS.

Compiling units

You compile a unit exactly the way you'd compile a program: Write it using the editor and select the **Compile | Compile** command (or press *Alt-F9*). However, instead of creating an .EXE file, Turbo Pascal will create a .TPU (Turbo Pascal Unit) file. You can then leave this file as is or merge it into TURBO.TPL using TPUMOVER.EXE.

In any case, you probably want to copy your .TPU files (along with their source) to the unit directory you specified in the **Unit Directories** input box (**Options | Directories**). That way, you can reference those files without having to have them in the current directory or in TURBO.TPL. (The **Unit Directories** input box lets you give multiple directories for the compiler to search for in unit files.)

You can only have one unit in a given source file; compilation stops when the final **end** statement is encountered.

To locate a unit specified in a **uses** clause, the compiler first checks the resident units—those units loaded into memory at startup from the TURBO.TPL file. If the unit is not among the resident units, the compiler assumes it must be on disk. The name of the file is assumed to be the unit name with extension .TPU. It is first searched for in the current directory, and then in the directories specified with the **O | D | Unit Directories** menu command or in a */U* directive on the TPC command line. For instance, the construct

```
uses Memory;
```

where *Memory* is not a resident unit, causes the compiler to look for the file MEMORY.TPU in the current directory, and then in each of the unit directories.

When the **Compile | Make** and **Compile | Build** commands compile the units specified in a **uses** clause, the source files are searched for in the same way as the .TPU files, and the name of a given unit's source file is assumed to be the unit name with extension .PAS.

An example

Okay, now let's write a small unit. We'll call it *IntLib* and put in two simple integer routines—a procedure and a function:

```
unit IntLib;
interface
procedure ISwap(var I,J: Integer);
function IMax(I,J: Integer): Integer;
implementation
procedure ISwap;
var
  Temp: Integer;
begin
  Temp := I; I := J; J := Temp;
end; { of proc ISwap }
function IMax;
begin
  if I > J then
    IMax := I
  else IMax := J;
end; { of func IMax }
end. { of unit IntLib }
```

Type this in, save it as the file INTLIB.PAS, then compile it to disk. The resulting unit code file is INTLIB.TPU. Move it to your unit directory (whatever that might happen to be) or leave it in the same directory as the program that follows. This next program uses the unit *IntLib*:

```
program IntTest;
uses IntLib;
var
  A,B: Integer;
begin
  Write('Enter two integer values: ');
  Readln(A,B);
```



```

ISwap(A,B);
Writeln('A = ', A, ' B = ', B);
Writeln('The max is ', IMax(A,B));
end. { of program IntTest }

```

Congratulations! You've just created your first unit and written a program that uses it!

Units and large programs

Up until now, you've probably thought of units only as libraries—collections of useful routines to be shared by several programs. Another function of a unit, however, is to break up a large program into modules.

Two aspects of Turbo Pascal make this modular functionality of units work: (1) its tremendous speed in compiling and linking and (2) its ability to manage several code files simultaneously, such as a program and several units.

Typically, a large program is divided into units that group procedures by their function. For instance, an editor application could be divided into initialization, printing, reading and writing files, formatting, and so on. Also, there could be a "global" unit—one used by all other units, as well as the main program—that defines global constants, data types, variables, procedures, and functions.

The skeleton of a large program might look like this:

```

program Editor;
uses
  Dos,Crt,Printer           { Standard units from TURBO.TPL }
  EditGlobals,             { User-written units }
  EditInit,
  EditPrint,
  EditRead,EditWrite,
  EditFormat;

  { Program's declarations, procedures, and functions }
begin { main program }
end. { of program Editor }

```

Note that the units in this program could either be in TURBO.TPL or in their own individual .TPU files. If the latter is true, then Turbo Pascal will manage your project for you. This means when you recompile program *Editor* using the compiler's built-in make facility, Turbo Pascal will compare the dates of each .PAS and

.TPU file and recompile modules whose source has been modified.

Another reason to use units in large programs has to do with code segment limitations. The 8086 (and related) processors limit the size of a given chunk, or segment, of code to 64K. This means that the main program and any given segment cannot exceed a 64K size. Turbo Pascal handles this by making each unit a separate code segment. Your upper limit is the amount of memory the machine and operating system can support—640K on most PCs. Without units, you're limited to 64K of code for your program. (See Chapter 6, "Project management," for more information about how to deal with large programs.)

Units as overlays

Sometimes, even the ability to have multiple units loaded isn't enough to solve your memory problems. You might not have 640K to work with, or you may need to have large amounts of data in memory at the same time. In other words, you just can't fit your entire program into memory at once.

Turbo Pascal offers a solution: *overlays*. An overlay is a chunk of program that is loaded into memory when needed, and unloaded when not. This allows you to bring in sections of a program only when you need them.

Overlays in Turbo Pascal are based on units: The smallest chunk of code that can be loaded or unloaded is an entire unit. You can define complex sets of overlays, specifying which units can or cannot be in memory at the same time. Best of all, with Turbo Pascal's intelligent overlay manager, you don't have to worry about loading or unloading the overlays yourself—it's all done automatically.

You'll learn more about overlays and how to set them up and use them in Chapter 13 of the *Programmer's Guide*, "The Overlay unit."

The TPUMOVER utility

Suppose you want to add a well-designed and thoroughly debugged unit to the library of standard units (TURBO.TPL) so that it's automatically loaded into memory when you run the compiler. Is there any way to add to TURBO.TPL? Yes, by using the TPUMOVER.EXE utility.

*You can find out more about
TPUMOVER in the file
UTILS.DOC in ONLINE.ZIP on
your distribution disks.*

You can also use TPUMOVER to remove units from the Turbo Pascal standard unit library file, reducing its size and the amount of memory it takes up when loaded.

As you've seen, it's really quite simple to write your own units. A well-designed, well-implemented unit simplifies program development; you solve the problems only once, not for each new program. Best of all, a unit provides a clean, simple mechanism for writing very large programs.

Object-oriented programming

Object-oriented programming (OOP) is a method of programming that closely mimics the way all of us get things done. It is a natural evolution from earlier innovations to programming language design: It is more structured than previous attempts at structured programming; and it is more modular and abstract than previous attempts at data abstraction and detail hiding. Three main properties characterize an object-oriented programming language:

- *Encapsulation*: Combining a record with the procedures and functions that manipulate it to form a new data type—an object.
- *Inheritance*: Defining an object and then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors' code and data.
- *Polymorphism*: Giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

Turbo Pascal's language extensions give you the full power of object-oriented programming: more structure and modularity, more abstraction, and reusability built right into the language. All these features add up to code that is more structured, extensible, and easy to maintain.

The challenge of object-oriented programming is that it requires you to set aside habits and ways of thinking about programming that have been standard for many years. Once you do that,

however, OOP is a simple, straightforward, superior tool for solving many of the problems that plague traditional programs.

A note to you who have done object-oriented programming in other languages: Put aside your previous impressions of OOP and learn Turbo Pascal's object-oriented features on their own terms. OOP is not one single way of programming; it is a continuum of ideas. In its object philosophy, Turbo Pascal is more like C++ than Smalltalk. Smalltalk is an interpreter, while from the beginning, Turbo Pascal has been a pure native code compiler. Native code compilers do things differently (and far more quickly) than interpreters. Turbo Pascal was designed to be a production development tool, not a research tool.

And a note to you who haven't any notion at all what OOP is about: That's just as well. Too much hype, too much confusion, and too many people talking about something they don't understand have greatly muddied the waters in recent years. Strive to forget what people have told you about OOP. The best way (in fact, the *only* way) to learn anything useful about OOP is to do what you're about to do: Sit down and try it yourself.

Objects?

Yes, objects. Look around you...there's one: the apple you brought in for lunch. Suppose you were going to describe an apple in software terms. The first thing you might be tempted to do is pull it apart: Let *S* represent the area of the skin; let *J* represent the fluid volume of juice it contains; let *F* represent the weight of fruit inside; let *D* represent the number of seeds....

Don't think that way. Think like a painter. You see an apple, and you paint an apple. The picture of an apple is not an apple; it's just a symbol on a flat surface. But it hasn't been abstracted into seven numbers, all standing alone and independent in a data segment somewhere. Its components remain together, in their essential relationships to one another.

Objects model the characteristics and behavior of the elements of the world we live in. They are the ultimate data abstraction so far.

An apple can be pulled apart, but once it's been pulled apart it's not an apple anymore. The relationships of the parts to the whole and to one another are plainer when everything is kept together

Objects keep all their characteristics and behavior together.

in one wrapper. This is called *encapsulation*, and it's very important. We'll return to encapsulation in a little while.

Equally important, objects can *inherit* characteristics and behavior from what are called *ancestor objects*. This is an intuitive leap; inheritance is perhaps the single biggest difference between object-oriented Turbo Pascal and Standard Pascal programming today.

Inheritance

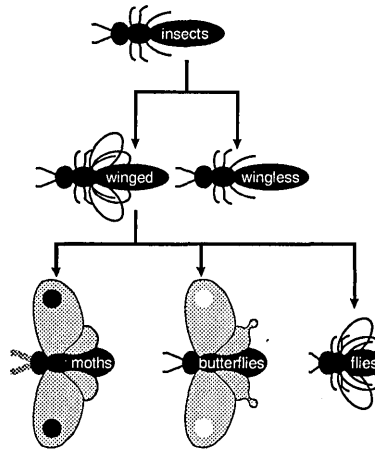
The goal of science is to describe the workings of the universe. Much of the work of science, in furthering that goal, is simply the creation of family trees. When entomologists return from the Amazon with a previously unknown insect in a jar, their fundamental concern is working out where that insect fits into the giant chart upon which the scientific names of all other insects are gathered. There are similar charts of plants, fish, mammals, reptiles, chemical elements, subatomic particles, and external galaxies. They all look like family trees: a single overall category at the top, with an increasing number of categories beneath that single category, fanning out to the limits of diversity.

Within the category *insect*, for example, there are two divisions: insects with visible wings, and insects with hidden wings or no wings at all. Under winged insects is a larger number of categories: moths, butterflies, flies, and so on. Each category has numerous subcategories, and beneath those subcategories are even more subcategories (see Figure 4.1).

This classification process is called *taxonomy*. It's a good starting metaphor for the inheritance mechanism of object-oriented programming.

The questions a scientist asks in trying to classify a 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. A scientist begins at the top of a specimen's family tree and starts 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. Eventually the scientist gets to the point of counting hairs on the third segment of the insect's hind legs—specific indeed (and a good reason, perhaps, not to be an entomologist).

Figure 4.1
A partial taxonomy chart of
insects



The important point to remember is that 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 of insect called *flies* inherits that characteristic from its order.

As you'll learn shortly, object-oriented programming is the process of building family trees for data structures. One of the important things object-oriented programming adds to traditional languages like Pascal is a mechanism by which data types inherit characteristics from simpler, more general types. This mechanism is inheritance.

Objects: records that inherit

In Pascal terms, an object is very much like a record, which is a wrapper for joining several related elements of data together under one name. In a graphics environment, you might gather together the *X* and *Y* coordinates of a position on the graphics screen and call it a record type named *Location*:

```
Location = record
  X, Y: Integer;
end;
```

Location here is a *record type*; that is, it's a template that the compiler uses to create record variables. A variable of type *Location* is an instance of type *Location*. The term *instance* is used now and then in Pascal circles, but it is used all the time by OOP people, and you'll do well to start thinking in terms of types and instances of those types.

With type *Location* you have it both ways: When you need to think of the *X* and *Y* coordinates separately, you can think of them separately as fields *X* and *Y* of the record. On the other hand, when you need to think of the *X* and *Y* coordinates working together to pin down a place on the screen, you can think of them collectively as *Location*.

Suppose you want to display a point of light at a position described on the screen by a *Location* record. In Pascal you might add a Boolean field indicating whether there is an illuminated pixel at a given location, and make it a new record type:

```
Point = record
  X, Y: Integer;
  Visible: Boolean;
end;
```

You might also be a little more clever and retain record type *Location* by creating a field of type *Location* within type *Point*:

```
Point = record
  Position: Location;
  Visible: Boolean;
end;
```

This works, and Pascal programmers do it all the time. One thing this method doesn't do is force you to think about the nature of what you're manipulating in your software. You need to ask questions like, "How does a point on the screen differ from a location on the screen?" The answer is this: A point is a location that lights up. Think back on the first part of that statement: A *point* is a *location*....

There you have it!

Implicit in the definition of a point is a location for that point. (Pixels exist only onscreen, after all.) Object-oriented programming recognizes that special relationship. Because all points must contain a location, type *Point* is a descendant type of type *Location*. *Point* inherits everything that *Location* has, and adds whatever is new about *Point* to make *Point* what it must be.

This process by which one type inherits the characteristics of another type is called *inheritance*. The inheritor is called a *descendant type*; the type that the descendant type inherits from is an *ancestor type*.

The familiar Pascal record types cannot inherit. Turbo Pascal, however, extends the Pascal language to support inheritance. One of these extensions is a new category of data structure, related to records but far more powerful. Data types in this new category are defined with a new reserved word: **object**. An object type can be defined as a complete, stand-alone type in the fashion of Pascal records, or it can be defined as a descendant of an existing object type by placing the name of the ancestor type in parentheses after the reserved word **object**.

In the graphics example you just looked at, the two related object types would be defined this way:

```
type
  Location = object
    X, Y: Integer;
  end;
  Point = object (Location)
    Visible: Boolean;
  end;
```

Note the use of parentheses here to denote inheritance.

Here, *Location* is the ancestor type, and *Point* is the descendant type. As you'll see a little later, the process can continue indefinitely: You can define descendants of type *Point*, and descendants of *Point's* descendant type, and so on. A large part of designing an object-oriented application lies in building this *object hierarchy* expressing the family tree of the objects in the application.

All the types eventually inheriting from *Location* are called *Location's* descendant types, but *Point* is one of *Location's* immediate descendants. Conversely, *Location* is *Point's* immediate ancestor. An object type (just like a DOS subdirectory) can have any number of immediate descendants, but only one immediate ancestor.

Objects are closely related to records, as these definitions show. The new reserved word **object** is the most obvious difference, but there are numerous other differences, some of them quite subtle, as you'll see later.

For example, the *X* and *Y* fields of *Location* are not explicitly written into type *Point*, but *Point* has them anyway, by virtue of

inheritance. You can speak about *Point's* X value, just as you can speak about *Location's* X value.

Instances of object types

Instances of object types are declared just as any variables are declared in Pascal, either as static variables or as pointer referents allocated on the heap:

```
type
  PointPtr = ^Point;

var
  StatPoint: Point;    { Ready to go! }
  DynaPoint: PointPtr; { Must allocate with New before use }
```

An object's fields

You access an object's data fields just as you access the fields of an ordinary record, either through the **with** statement or by *dotting*. For example,

```
MyPoint.Visible := False;

with MyPoint do
begin
  X := 341;
  Y := 42;
end;
```

Don't forget: An object's inherited fields are not treated specially simply because they are inherited.

You just have to remember at first (eventually it comes naturally) that inherited fields are just as accessible as fields declared within a given object type. For example, even though X and Y are not part of *Point's* declaration (they are inherited from type *Location*), you can specify them just as though they were declared within *Point*:

```
MyPoint.X := 17;
```

Good practice and bad practice

Turbo Pascal actually lets you make an object's fields and methods private; for more on this, refer to page 87.

Even though you can access an object's fields directly, it's not an especially good idea to do so. Object-oriented programming principles require that an object's fields be left alone as much as possible. This restriction might seem arbitrary and rigid at first, but it's part of the big picture of OOP that is being built in this chapter. In time you'll see the sense behind this new definition of good programming practice, though there's some ground to cover

before it all comes together. For now, take it on faith: Avoid accessing object data fields directly.

So—how are object fields accessed? What sets them and reads them?

An object's data fields are what an object knows; its methods are what an object does.

The answer is that an object's *methods* are used to access an object's data fields whenever possible. A *method* is a procedure or function declared *within* an object and tightly bonded to that object.

Methods

Methods are one of object-oriented programming's most striking attributes, and they take some getting used to. Start by harkening back to that fond old necessity of structured programming, initializing data structures. Consider the task of initializing a record with this definition:

```
Location = record
  X, Y: Integer;
end;
```

Most programmers would use a **with** statement to assign initial values to the X and Y fields:

```
var
  MyLocation: Location;
with MyLocation do
begin
  X := 17;
  Y := 42;
end;
```

This works well, but it's tightly bound to one specific record instance, *MyLocation*. If more than one *Location* record needs to be initialized, you'll need more **with** statements that do essentially the same thing. The natural next step is to build an initialization procedure that generalizes the **with** statement to encompass any instance of a *Location* type passed as a parameter:

```

procedure InitLocation(var Target: Location; NewX, NewY: Integer);
begin
  with Target do
    begin
      X := NewX;
      Y := NewY;
    end;
  end;

```

This does the job, all right—but if you’re getting the feeling that it’s a little more fooling around than it ought to be, you’re feeling the same thing that object-oriented programming’s early proponents felt.

It’s a feeling that implies that, well, you’ve designed procedure *InitLocation* specifically to serve type *Location*. Why, then, must you keep specifying what record type and instance *InitLocation* acts upon? There should be some way of welding together the record type and the code that serves it into one seamless whole.

Now there is. It’s called a *method*. A method is a procedure or function welded so tightly to a given type that the method is surrounded by an invisible **with** statement, making instances of that type accessible from within the method. The type definition includes the header of the method. The full definition of the method is qualified with the name of the type. Object type and object method are the two faces of this new species of structure called an object:

```

type
  Location = object
    X, Y: Integer;
    procedure Init(NewX, NewY: Integer);
  end;

procedure Location.Init(NewX, NewY: Integer);
begin
  X := NewX; { The X field of a Location object }
  Y := NewY; { The Y field of a Location object }
end;

```

Now, to initialize an instance of type *Location*, you simply call its method as though the method were a field of a record, which in one very real sense it is:

```

var
  MyLocation: Location;

MyLocation.Init(17, 42); { Easy, no? }

```

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 exists 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 procedure with the wrong data or the wrong procedure with the right data. Matching the two is the programmer's job, and while Pascal's strong typing does help, at best it can only say what *doesn't* go together.

Pascal says nothing, anywhere, about what *does* go together. If it's not in a comment or in your head, you take your chances.

By bundling code and data declarations together, an object helps keep them in sync. Typically, to get the value of one of an object's fields, you call a method belonging to that object that returns the value of the desired field. To set the value of a field, you call a method that assigns a new value to that field.

See "Private section" on page 87 for details on how to do this.

Like many aspects of object-oriented programming, respect for encapsulated data is a discipline you should always observe. It's better to access an object's data by using the methods it provides, instead of reading the data directly. Turbo Pascal lets you enforce encapsulation through the use of a **private** declaration in an object's declaration.

Defining methods

The process of defining an object's methods is reminiscent of Turbo Pascal units. Inside an object, a method is defined by the header of the function or procedure acting as a method:

```
type
  Location = object
    X, Y: Integer;
    procedure Init(InitX, InitY: Integer);
    function GetX: Integer;
    function GetY: Integer;
end;
```

All data fields must be declared before the first method declaration.

As with procedure and function declarations in a unit's **interface** section, method declarations within an object tell *what* a method does, but not *how*.

The *how* is defined *outside* the object definition, in a separate procedure or function declaration. When methods are fully defined outside the object, the method name must be preceded by the name of the object type that owns the method, followed by a period:

```
procedure Location.Init(InitX, InitY: Integer);  
begin  
  X := InitX;  
  Y := InitY;  
end;  
  
function Location.GetX: Integer;  
begin  
  GetX := X;  
end;  
  
function Location.GetY: Integer;  
begin  
  GetY := Y;  
end;
```

Method definition follows the intuitive dotting method of specifying a record field. In addition to having a definition of *Location.GetX*, it would be completely legal to define a procedure named *GetX* without the identifier *Location* preceding it. However, the "outside" *GetX* would have no connection to the object type *Location* and would probably confuse the sense of the program as well.

Method scope and the Self parameter

Notice that nowhere in the previous methods is there an explicit `with object do ...` construct. The data fields of an object are freely available to that object's methods. Although they are separated in the source code, the method bodies and the object's data fields really share the same scope.

This is why one of *Location*'s methods can contain the statement `GetY := Y` without any qualifier to *Y*. It's because *Y belongs to the object that called the method*. When an object calls a method, there is

an implicit statement to the effect `with myself do` method linking the object and its method in scope.

This implicit `with` statement is accomplished by the passing of an invisible parameter to the method each time any method is called. This parameter is called *Self*, and is actually a full 32-bit pointer to the object instance making the method call. The *GetY* method belonging to *Location* is roughly equivalent to the following:

This example is not fully correct syntactically; it's here simply to give you a fuller appreciation for the special link between an object and its methods.

```
function Location.GetY(var Self: Location): Integer;
begin
  GetY := Self.Y;
end;
```

Is it important for you to be aware of *Self*? Ordinarily, no: Turbo Pascal's generated code handles it all automatically in virtually all cases. There are a few circumstances, however, when you might have to intervene inside a method and make explicit use of the *Self* parameter.

Explicit use of Self is legal, but you should avoid situations that require it.

Self is actually an automatically declared identifier, and if you happen to find yourself in the midst of an identifier conflict within a method, you can resolve it by using the *Self* identifier as a qualifier to any data field belonging to the method's object:

```
type
  MouseStat = record
    Active: Boolean;
    X, Y: Integer;
    LButton, RButton: Boolean;
    Visible: Boolean;
  end;

procedure Location.GoToMouse(MousePos: MouseStat);
begin
  Hide;
  with MousePos do
    begin
      Self.X := X;
      Self.Y := Y;
    end;
  Show;
end;
```

Methods implemented as externals in assembly language must take Self into account when they access method parameters on the stack. For more details on method call stack frames, see Chapter 18 in the Programmer's Guide.

Object data fields and method formal parameters

This example is necessarily simple, and the use of *Self* could be avoided simply by abandoning the use of the **with** statement inside *Location.GoToMouse*. You might find yourself in a situation inside a complex method where the use of **with** statements simplifies the logic enough to make *Self* worthwhile. The *Self* parameter is part of the physical stack frame for all method calls.

One consequence of the fact that methods and their objects share the same scope is that a method's formal parameters cannot be identical to any of the object's data fields. This is not some new restriction imposed by object-oriented programming, but rather the same old scoping rule that Pascal has always had. It's the same as not allowing the formal parameters of a procedure to be identical to the procedure's local variables:

```
procedure CrunchIt (Crunchee: MyDataRec, Crunchby, ErrorCode:
Integer);
var
  A, B: Char;
  ErrorCode: Integer; { This declaration causes an error! }
begin
  ...
```

A procedure's local variables and its formal parameters share the same scope and thus cannot be identical. You'll get "Error 4: Duplicate identifier" if you try to compile something like this; the same error occurs if you attempt to give a method a formal parameter identical to any field in the object that owns the method.

The circumstances are a little different, since having procedure headers inside a data structure is a wrinkle new to Turbo Pascal, but the guiding principles of Pascal scoping have not changed at all.

Objects exported by units

It makes good sense to define objects in units, with the object type declaration in the interface section of the unit and the procedure bodies of the object type's methods defined in the implementation section.

Exported means "defined within the interface section of a unit."

Units can have their own private object type definitions in the implementation section, and such types are subject to the same restrictions as any types defined in a unit implementation section. An object type defined in the interface section of a unit can have descendant object types defined in the implementation section of the unit. In a case where unit *B* uses unit *A*, unit *B* can also define descendant types of any object type exported by unit *A*.

The object types and methods described earlier can be defined within a unit as shown in POINTS.PAS on your disk. To make use of the object types and methods defined in unit *Points*, you simply use the unit in your own program, and declare an instance of type *Point* in the **var** section of your program:

```
program MakePoints;
uses Graph, Points;

var
  APoint: Point;
  ...
```

To create and show the point represented by *APoint*, you simply call *APoint's* methods, using the dot syntax:

```
APoint.Init(151, 82);    { Initial X,Y at 151,82 }
APoint.Show;            { APoint turns itself on }
APoint.MoveTo(163, 101); { APoint moves to 163,101 }
APoint.Hide;            { APoint turns itself off }
```

Objects can also be typed constants.

Objects, being very similar to records, can also be used inside **with** statements. In that case, naming the object that owns the method isn't necessary:

```
with APoint do
begin
  Init(151, 82);    { Initial X,Y at 151,82 }
  Show;            { APoint turns itself on }
  MoveTo(163, 101); { APoint moves to 163,101 }
  Hide;            { APoint turns itself off }
end;
```

Just as with records, objects can be passed to procedures as parameters and (as you'll see later on) can also be allocated on the heap.

Private section In some circumstances you may have parts of an object declaration that you don't want to export. For example, you may want to provide objects for other programmers to use without letting them manipulate the object's data directly. To make it easy for you, Turbo Pascal allows you to specify private fields and methods within objects.

Private fields and methods are accessible only within the unit in which the object is declared. In the previous example, if the type *Point* had private fields, for example, they could only be accessed by code within the *Points* unit. Even though other parts of *Point* would be exported, the parts declared as private would be inaccessible.

Private fields and methods are declared just after regular fields and methods, following the optional **private** reserved word. Thus, the full syntax for an object declaration is

```
type
  NewObject = object (ancestor)
    fields; { these are public }
    methods; { these are public }
  private
    fields; { these are private }
    methods; { these are private }
end;
```

Programming in the active voice

Most of what's been said about objects so far has been from a comfortable, Turbo Pascal-ish perspective, since that's most likely where you are coming from. This is about to change, as you move on to OOP concepts with fewer precedents in standard Pascal programming. Object-oriented programming has its own particular mindset, due in part to OOP's origins in the (somewhat insular) research community, but also because the concept is truly and radically different.

Object-oriented languages were once called "actor languages" with this metaphor in mind.

One often amusing outgrowth of this is that OOP fanatics anthropomorphize their objects. Data structures are no longer passive buckets that you toss values into. In the new view of things, an object is looked upon as an actor on a stage, with a set of lines (methods) memorized. When you (the director) give the word, the actor recites from the script.

It can be helpful to think of the statement `APoint.MoveTo(242,118)` as giving an order to object `APoint`, saying "Move yourself to location 242,118." The object is the central concept here. Both the list of methods and the list of data fields contained by the object serve the object. Neither code nor data is boss.

Objects aren't being described as actors on a stage just to be cute. The object-oriented programming paradigm tries very hard to model the components of a problem as components, and not as logical abstractions. The odds and ends that fill our lives, from toasters to telephones to terry towels, all have characteristics (data) and behaviors (methods). A toaster's characteristics might include the voltage it requires, the number of slices it can toast at once, the setting of the light/dark lever, its color, its brand, and so on. Its behaviors include accepting slices of bread, toasting slices of bread, and popping toasted slices back up again.

If you wanted to write a kitchen simulation program, what better way to do it than to model the various appliances as objects, with their characteristics and behaviors encoded into data fields and methods? It's been done, in fact; the very first object-oriented language (Simula-67) was created as a language for writing such simulations.

This is the reason that object-oriented programming is so firmly linked in conventional wisdom to graphics-oriented environments. Objects should be simulations, and what better way to simulate an object than to draw a picture of it? Objects in Turbo Pascal should model components of the problem you're trying to solve. Keep that in mind as you further explore Turbo Pascal's object-oriented extensions.

Encapsulation

*Declaring fields as **private** allows you to enforce access to those fields only through methods.*

The welding of code and data together into objects is called *encapsulation*. If you're thorough, you can provide enough methods so that a user of the object never has to access its fields directly. Like Smalltalk and other programming languages, Turbo Pascal lets you enforce encapsulation through the use of a **private** directive. In this example, we won't specify a **private** section for fields and methods, but instead we will restrict ourselves to using methods in order to access the data we want.

Location and *Point* are written such that it is completely unnecessary to access any of their internal data fields directly:

```

type
  Location = object
    X, Y: Integer;
    procedure Init(InitX, InitY: Integer);
    function GetX: Integer;
    function GetY: Integer;
  end;

  Point = object(Location)
    Visible: Boolean;
    procedure Init(InitX, InitY: Integer);
    procedure Show;
    procedure Hide;
    function IsVisible: Boolean;
    procedure MoveTo(NewX, NewY: Integer);
  end;

```

There are only three data fields here: *X*, *Y*, and *Visible*. The *MoveTo* method loads new values into *X* and *Y*, and the *GetX* and *GetY* methods return the values of *X* and *Y*. This leaves no further need to access *X* or *Y* directly. *Show* and *Hide* toggle the Boolean *Visible* between True and False, and the *IsVisible* function returns *Visible*'s current state.

Assuming an instance of type *Point* called *APoint*, you would use this suite of methods to manipulate *APoint*'s data fields indirectly, like this:

```

with APoint do
  begin
    Init(0, 0);           { Init new point at 0,0 }
    Show;                { Make the point visible }
  end;

```

Note that the object's fields are not accessed at all except by the object's methods.

Methods: no downside

Adding these methods bulks up *Point* a little in source form, but the Turbo Pascal smart linker strips out any method code that is never called in a program. You therefore shouldn't hang back from giving an object type a method that might or might not be used in every program that uses the object type. Unused methods cost you nothing in performance or .EXE file size—if they're not used, they're simply not there.



There are powerful advantages to being able to completely decouple *Point* from global references. If nothing outside the object “knows” the representation of its internal data, the programmer who controls the object can alter the details of the internal data representation—as long as the method headers remain the same.

Within some object, data might be represented as an array, but later on (perhaps as the scope of the application grows and its data volume expands), a binary tree might be recognized as a more efficient representation. If the object is completely encapsulated, a change in data representation from an array to a binary tree *does not alter the object’s use at all*. The interface to the object remains completely the same, allowing the programmer to fine-tune an object’s performance without breaking any code that uses the object.

Extending objects

People who first encounter Pascal often take for granted the flexibility of the standard procedure *WriteLn*, which allows a single procedure to handle parameters of many different types:

```
WriteLn(CharVar);      { Outputs a character value }
WriteLn(IntegerVar);  { Outputs an integer value }
WriteLn(RealVar);     { Outputs a floating-point value }
```

Unfortunately, standard Pascal has no provision for letting you create equally flexible procedures of your own.

Object-oriented programming solves this problem through inheritance: When a descendant type is defined, the methods of the ancestor type are inherited, but they can also be overridden if desired. To override an inherited method, simply define a new method with the same name as the inherited method, but with a different body and (if necessary) a different set of parameters.

A simple example should make both the process and the implications clear. Let’s define a descendant type to *Point* that draws a circle instead of a point on the screen:

```

type
  Circle = object (Point)
    Radius: Integer;
    procedure Init (InitX, InitY: Integer; InitRadius: Integer);
    procedure Show;
    procedure Hide;
    procedure Expand (ExpandBy: Integer);
    procedure MoveTo (NewX, NewY: Integer);
    procedure Contract (ContractBy: Integer);
  end;

procedure Circle.Init (InitX, InitY: Integer; InitRadius: Integer);
begin
  Point.Init (InitX, InitY);
  Radius := InitRadius;
end;

procedure Circle.Show;
begin
  Visible := True;
  Graph.Circle (X, Y, Radius);
end;

procedure Circle.Hide;
var
  TempColor: Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor (GetBkColor);
  Visible := False;
  Graph.Circle (X, Y, Radius);
  Graph.SetColor (TempColor);
end;

procedure Circle.Expand (ExpandBy: Integer);
begin
  Hide;
  Radius := Radius + ExpandBy;
  if Radius < 0 then Radius := 0;
  Show;
end;

procedure Circle.Contract (ContractBy: Integer);
begin
  Expand (-ContractBy);
end;

```

```

procedure Circle.MoveTo(NewX, NewY: Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

```

A circle, in a sense, is a fat point: It has everything a point has (an X,Y location, a visible/invisible state) plus a radius. Object type *Circle* appears to have only the single field *Radius*, but don't forget about all the fields that *Circle* inherits by being a descendant type of *Point*. *Circle* has X, Y, and *Visible* as well, even if you don't see them in the type definition for *Circle*.

Since *Circle* defines a new field, *Radius*, initializing it requires a new *Init* method that initializes *Radius* as well as the inherited fields. Rather than directly assigning values to inherited fields like X, Y and *Visible*, why not reuse *Point*'s initialization method (illustrated by *Circle.Init*'s first statement). The syntax for calling an inherited method is *Ancestor.Method*, where *Ancestor* is the type identifier of an ancestral object type, and *Method* is a method identifier of that type.

Note that calling the method you override is not merely good style; it's entirely possible that *Point.Init* (or *Location.Init* for that matter) performs some important, hidden initialization. By calling the overridden method, you ensure that the descendant object type includes its ancestor's functionality. In addition, any changes made to the ancestor's method automatically affects all its descendants.

After calling *Point.Init*, *Circle.Init* can then perform its own initialization, which in this case consists only of assigning *Radius* the value passed in *InitRadius*.

Instead of drawing and hiding your circle point by point, you can make use of the BGI *Circle* procedure. If you do, *Circle* also needs new *Show* and *Hide* methods that override those of *Point*. These rewritten *Show* and *Hide* methods appear in the example starting on page 90.

Dotting resolves the potential problems stemming from the name of the object type being the same as that of the BGI routine that draws the object type on the screen. *Graph.Circle* is also a completely unambiguous way of telling Turbo Pascal that you're

referencing the *Circle* routine in GRAPH.TPU and not the *Circle* object type.

Important!



Whereas methods can be overridden, data fields cannot. Once you define a data field in an object hierarchy, no descendant type can define a data field with precisely the same identifier.

Inheriting static methods

One additional *Point* method is overridden in the earlier definition of *Circle*: *MoveTo*. If you're sharp, you might be looking at *MoveTo* and wondering why *MoveTo* doesn't use the *Radius* field, and why it doesn't make any BGI or other library calls specific to drawing circles. After all, the *GetX* and *GetY* methods are inherited all the way from *Location* without modification. *Circle.MoveTo* is also completely identical to *Point.MoveTo*. Nothing was changed other than to copy the routine and give it *Circle's* qualifier in front of the *MoveTo* identifier.

This example demonstrates a problem with objects and methods set up in this fashion. All the methods shown so far in connection with the *Location*, *Point*, and *Circle* object types are static methods.



The term *static* was chosen to describe methods that are not *virtual*. (You will learn about virtual methods shortly.) Virtual methods are in fact the solution to this problem, but in order to understand the solution you must first understand the problem.

The symptoms of the problem are these: Unless a copy of the *MoveTo* method is placed in *Circle's* scope to override *Point's MoveTo*, the method does not work correctly when it is called from an object of type *Circle*. If *Circle* invokes *Point's MoveTo* method, what is moved on the screen is a point rather than a circle. Only when *Circle* calls a copy of the *MoveTo* method defined in its own scope are circles hidden and drawn by the nested calls to *Show* and *Hide*.

Why so? It has to do with the way the compiler resolves method calls. When the compiler compiles *Point's* methods, it first encounters *Point.Show* and *Point.Hide* and compiles code for both into the code segment. A little later down the file it encounters *Point.MoveTo*, which calls both *Point.Show* and *Point.Hide*. As with any procedure call, the compiler replaces the source code references to *Point.Show* and *Point.Hide* with the addresses of their generated code in the code segment. Thus, when the code for

Point.MoveTo is called, it in turn calls the code for *Point.Show* and *Point.Hide* and everything's in phase.

So far, this scenario is all classic Turbo Pascal and would have been true (except for the nomenclature) since version 1.0. Things change, however, when you get into inheritance. When *Circle* inherits a method from *Point*, *Circle* uses the method exactly as it was compiled.

Look again at what *Circle* would inherit if it inherited *Point.MoveTo*:

```
procedure Point.MoveTo(NewX, NewY: Integer);  
begin  
  Hide;           { Calls Point.Hide }  
  X := NewX;  
  Y := NewY;  
  Show;          { Calls Point.Show }  
end;
```

The comments were added to drive home the fact that when *Circle* calls *Point.MoveTo*, it also calls *Point.Show* and *Point.Hide*, not *Circle.Show* and *Circle.Hide*. *Point.Show* draws a point, not a circle. As long as *Point.MoveTo* calls *Point.Show* and *Point.Hide*, *Point.MoveTo* can't be inherited. Instead, it must be overridden by a second copy of itself that calls the copies of *Show* and *Hide* defined within its scope; that is, *Circle.Show* and *Circle.Hide*.

The compiler's logic in resolving method calls works like this: When a method is called, the compiler first looks for a method of that name defined within the object type. The *Circle* type defines methods named *Init*, *Show*, *Hide*, *Expand*, *Contract*, and *MoveTo*. If a *Circle* method were to call one of those five methods, the compiler would replace the call with the address of one of *Circle's* own methods.

If no method by a name is defined within an object type, the compiler goes up to the immediate ancestor type, and looks within that type for a method of the name called. If a method by that name is found, the address of the ancestor's method replaces the name in the descendant's method's source code. If no method by that name is found, the compiler continues up to the next ancestor, looking for the named method. If the compiler hits the very first (top) object type, it issues an error message indicating that no such method is defined.

But when a static inherited method is found and used, you must remember that the method called is the method exactly as it was defined *and compiled* for the ancestor type. If the ancestor's method calls other methods, the methods called are the ancestor's methods, even if the descendant has methods that override the ancestor's methods.

Virtual methods and polymorphism

The methods discussed so far are static methods. They are static for the same reason that static variables are static: The compiler allocates them and resolves all references to them *at compile time*. As you've seen, objects and static methods can be powerful tools for organizing a program's complexity.

Sometimes, however, they are not the best way to handle methods.

Problems like the one described in the previous section are due to the compile-time resolution of method references. The way out is to be dynamic—and resolve such references at run time. Certain special mechanisms must be in place for this to be possible, but Turbo Pascal provides those mechanisms in its support of virtual methods.

Important!



Virtual methods implement an extremely powerful tool for generalization called polymorphism. *Polymorphism* is Greek for “many shapes,” and it is just that: A way of giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself.

The simple hierarchy of graphic figures already described provides a good example of polymorphism in action, implemented through virtual methods.

Each object type in our 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 objects to represent other figures such as lines, squares, arcs, and so on, you could write a method 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. That much they all have in common.

What is different for each object type is the *way* it must show itself to the screen. A point is drawn with a point-plotting routine that needs nothing more than an X,Y location and perhaps a color value. A circle needs an entirely separate 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 more complex drawing algorithm to take them into account.

Any graphic figure can be shown, but the mechanism by which each is shown is specific to each figure. One word, "Show," is used to show (literally) many shapes.

That's a good example of what polymorphism is, and virtual methods are how it is done in Turbo Pascal.

Early binding vs. late binding

The difference between a static method call and a virtual method call is the difference between a decision made now and a decision delayed. When you code a static method call, you are in essence telling the compiler, "You know what I want. Go call it." Making a virtual method call, on the other hand, is like telling the compiler, "You don't know what I want—yet. When the time comes, ask the instance."

Think of this metaphor in terms of the *MoveTo* problem mentioned in the previous section. A call to *Circle.MoveTo* can only go to one place: the closest implementation of *MoveTo* up the object hierarchy. In that case, *Circle.MoveTo* would still call *Point's* definition of *MoveTo*, since *Point* is the closest up the hierarchy from *Circle*. Assuming that no descendent type defined its own *MoveTo* to override *Point's* *MoveTo*, any descendent type of *Point* would still call the same implementation of *MoveTo*. The decision can be made at compile time and that's all that needs to be done.

When *MoveTo* calls *Show*, however, it's a different story. Every figure type has its own implementation of *Show*, so which implementation of *Show* is called by *MoveTo* should depend entirely on what object instance originally called *MoveTo*. This is why the call to the *Show* method within the implementation of *MoveTo* must be a delayed decision: When the code for *MoveTo* is compiled, no decision as to which *Show* to call can be made. The information isn't available at compile time, so the decision has to be deferred until run time, when the object instance calling *MoveTo* can be queried.

The process by which static method calls are resolved unambiguously to a single method by the compiler at compile time is *early binding*. In early binding, the caller and the callee are connected (bound) at the earliest opportunity, that is, at compile time. With *late binding*, the caller and the callee cannot be bound at compile time, so a mechanism is put into place to bind the two later on, when the call is actually made.

The nature of the mechanism is interesting and subtle, and you'll see how it works a little later.

Object type compatibility

Inheritance somewhat changes Turbo Pascal's type compatibility rules. In addition to everything else, a descendant type inherits type compatibility with all its ancestor types. This extended type compatibility takes three forms:

- between object instances
- between pointers to object instances
- between formal and actual parameters

In all three forms, however, it is critical to remember that type compatibility extends *only* from descendant to ancestor. In other words, descendant types can be freely used in place of ancestor types, but not vice versa.

Consider these declarations:

```
type
  LocationPtr = ^Location;
  PointPtr = ^Point;
  CirclePtr = ^Circle;

var
  ALocation: Location;
  APoint: Point;
  ACircle: Circle;
  PLocation: LocationPtr;
  PPoint: PointPtr;
  PCircle: CirclePtr;
```

With these declarations, the following assignments are legal:

```
ALocation := APoint;
APoint := ACircle;
ALocation := ACircle;
```

An ancestor object can be assigned an instance of any of its descendant types.

The reverse assignments are not legal.

This is a concept new to Pascal, and it might be a little hard to remember, at first, which way the type compatibility goes. Think of it this way: *The source must be able to completely fill the destination.* Descendant types contain everything their ancestor types contain by virtue of inheritance. Therefore a descendant type is either exactly the same size or (usually) larger than its ancestors, but never smaller. Assigning an ancestor object to a descendant object could leave some of the descendant's fields undefined after the assignment, which is dangerous and therefore illegal.

In an assignment statement, only the fields that the two types have in common are copied from the source to the destination. In the assignment statement

```
Alocation := ACircle;
```

only the *X* and *Y* fields of *ACircle* are copied to *Alocation*, since *X* and *Y* are all that types *Circle* and *Location* have in common.

Type compatibility also operates between pointers to object types, under the same rule as for instances of object types: Pointers to descendants can be assigned to pointers to ancestors. These pointer assignments are also legal:

```
PPoint := PCircle;  
PLocation := PPoint;  
PLocation := PCircle;
```

Again, the reverse assignments are not legal.

A formal parameter (either value or **var**) of a given object type can take as an actual parameter an object of its own, or any descendant type. Given this procedure header,

```
procedure DragIt (Target: Point);
```

actual parameters could legally be of type *Point* or *Circle*, but not type *Location*. *Target* could also be a **var** parameter; the same type compatibility rule applies.

Warning!
▶▶▶▶▶

However, keep in mind that there's a drastic difference between a value parameter and a **var** parameter: A **var** parameter is a pointer to the actual object passed as a parameter, whereas a value parameter is only a *copy* of the actual parameter. That copy, moreover, only includes the fields and methods included in the formal value parameter's type. This means the actual parameter is literally translated to the type of the formal parameter. A **var**

parameter is more similar to a typecast, in that the actual parameter remains unaltered.

Similarly, if a formal parameter is a pointer to an object type, the actual parameter can be a pointer to that object type or a pointer to any of that object's descendant types. Given this procedure header,

```
procedure Figure.Add(NewFigure: PointPtr);
```

actual parameters could legally be of type *PointPtr* or *CirclePtr*, but not type *LocationPtr*.

Polymorphic objects

In reading the previous section, you might have asked yourself: If any descendant type of a parameter's type can be passed in the parameter, how does the user of the parameter know which object type it is receiving? In fact, the user does not know, not directly. The exact type of the actual parameter is unknown at compile time. It could be any one of the object types descended from the **var** parameter type and is thus called a *polymorphic object*.

Now, exactly what are polymorphic objects good for? Primarily, this: *Polymorphic objects allow the processing of objects whose type is not known at compile time.* This whole notion is so new to the Pascal way of thinking that an example might not occur to you immediately. (You'll be surprised, in time, at how natural it begins to seem. That's when you'll truly be an object-oriented programmer.)

Suppose you've written a graphics drawing toolbox that supports numerous types of figures: points, circles, squares, rectangles, curves, and so on. As part of the toolbox, you want to write a routine that drags a graphics figure around the screen with the mouse pointer.

The old way would have been to write a separate drag procedure for each type of graphics figure supported by the toolbox. You would have had to write *DragCircle*, *DragSquare*, *DragRectangle*, and so on. Even if the strong typing of Pascal allowed it (and don't forget, there are always ways to circumvent strong typing), the differences between the types of graphics figures would seem to prevent a truly general dragging routine from being written.

After all, a circle has a radius but no sides, a square has one length of side, a rectangle two different lengths of side, and curves, arrgh....

At this point, clever Turbo Pascal hackers will step forth and say, do it this way: Pass the graphics figure record to procedure *DragIt* as the referent of a generic pointer. Inside *DragIt*, examine a tag field at a fixed offset inside the graphics figure record to determine what sort of figure it is, and then branch using a **case** statement:

```
case FigureIDTag of
  Point      : DragPoint;
  Circle     : DragCircle;
  Square     : DragSquare;
  Rectangle  : DragRectangle;
  Curve     : DragCurve;
  ...
```

Well, placing seventeen small suitcases inside one enormous suitcase is a slight step forward, but what's the real problem with this way of doing things?

What if the user of the toolbox defines some new graphics figure type?

What indeed? What if the user designs traffic signs and wants to work with octagons for stop signs? The toolbox does not have an Octagon type, so *DragIt* would not have an Octagon label in its **case** statement, and would therefore refuse to drag the new Octagon figure. If it were presented to *DragIt*, Octagon would fall out in the **case** statement's **else** clause as an "unrecognized figure."

Plainly, building a toolbox of routines for sale without source code suffers from this problem: The toolbox can only work on data types that it "knows," that is, that are defined by the designers of the toolbox. The user of the toolbox is powerless to extend the function of the toolbox in directions unanticipated by the toolbox designers. What the user buys is what the user gets. Period.

The way out is to use Turbo Pascal's extended type compatibility rules for objects and design your application to use polymorphic objects and virtual methods. If a toolbox *DragIt* procedure is set up to work with polymorphic objects, it works with any objects defined within the toolbox—and any descendant objects that you define yourself. If the toolbox object types use virtual methods, the toolbox objects and routines can work with your custom graphics figures *on the figures' own terms*. A virtual method you define today is callable by a toolbox .TPU unit file that was

written and compiled a year ago. Object-oriented programming makes it possible, and virtual methods are the key.

Understanding how virtual methods make such polymorphic method calls possible requires a little background on how virtual methods are declared and used.

Virtual methods

A method is made virtual by following its declaration in the object type with the new reserved word **virtual**. Remember that if you declare a method in an ancestor type **virtual**, all methods of the same name in any descendant must also be declared **virtual** to avoid a compiler error.

Here are the graphics shape objects you've been seeing, properly virtualized:

```
type
  Location = object
    X, Y: Integer;
    procedure Init(InitX, InitY: Integer);
    function GetX: Integer;
    function GetY: Integer;
  end;

  Point = object(Location)
    Visible: Boolean;
    constructor Init(InitX, InitY: Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible: Boolean;
    procedure MoveTo(NewX, NewY: Integer);
  end;

  Circle = object(Point)
    Radius: Integer;
    constructor Init(InitX, InitY: Integer; InitRadius: Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy: Integer); virtual;
    procedure Contract(ContractBy: Integer); virtual;
  end;
```

Notice first of all that the *MoveTo* method shown in the last iteration of type *Circle* is gone from *Circle*'s type definition. *Circle* no longer needs to override *Point*'s *MoveTo* method with an unmodified copy compiled within its own scope. Instead, *MoveTo* can now be inherited from *Point*, with all *MoveTo*'s nested method

calls going to *Circle's* methods rather than *Point's*, as happens in an all-static object hierarchy.

We suggest the use of the identifier *Init* for object constructors.

Also, notice the new reserved word **constructor** replacing the reserved word **procedure** for *Point.Init* and *Circle.Init*. A constructor is a special type of procedure that does some of the setup work for the machinery of virtual methods.

Warning! Every object type that has virtual methods must have a constructor.

- ▣ The constructor must be called before any virtual method is called. Calling a virtual method without previously calling the constructor can cause system lockup, and the compiler has no way to check the order in which methods are called.
- ▣ Each individual instance of an object must be initialized by a separate constructor call. It is not sufficient to initialize one instance of an object and then assign that instance to additional instances. The additional instances, while they might contain correct data, are not initialized by the assignment statements, and lock up the system if their virtual methods are called. For example:

```
var
  QCircle, RCircle: Circle;           { create two instances of Circle }
begin
  QCircle.Init(600,100,30);           { call constructor for QCircle }
  RCircle := QCircle;                 { RCircle is not valid! }
end.
```

What do constructors construct? Every object type has something called a *virtual method table* (VMT) in the data segment. The VMT contains the object type's size and, for each of its virtual methods, a pointer to the code implementing that method. What the constructor does is establish a link between the instance calling the constructor and the object type's VMT.

That's important to remember: There is only one virtual method table for each object type. Individual instances of an object type (that is, variables of that type) contain a link to the VMT—they do not contain the VMT itself. The constructor sets the value of that link to the VMT—which is why you can launch execution into nowhere by calling a virtual method before calling the constructor.

Range checking virtual method calls

The default state of *\$R* is inactive, (*\$R-*).

During program development, you might wish to take advantage of a safety net that Turbo Pascal places beneath virtual method calls. If the **\$R** toggle is in its active state, (**\$R+**), all virtual method calls are checked for the initialization status of the instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs.

Once you've shaken out a program and are certain that no method calls from uninitialized instances are present, you can speed your code up somewhat by setting the **\$R** toggle to its inactive state, (**\$R-**). Method calls from uninitialized instances will no longer be checked for, and will probably lock up your system if they're found.

Once virtual, always virtual

Notice that both *Point* and *Circle* have methods named *Show* and *Hide*. All method headers for *Show* and *Hide* are tagged as virtual methods with the reserved word **virtual**. Once an ancestor object type tags a method as **virtual**, all its descendant types that implement a method of that name must tag that method **virtual** as well. In other words, a static method can never override a virtual method. If you try, a compiler error results.

You should also keep in mind that the method heading cannot change in *any* way downward in an object hierarchy once the method is made virtual. You might think of each definition of a virtual method as a gateway to *all* of them. For this reason, the headers for all implementations of the same virtual method must be identical, right down to the number and type of parameters. This is not the case for static methods; a static method overriding another can have different numbers and types of parameters as necessary.

An example of late binding

To show how to use polymorphic objects with late binding in a Turbo Pascal program, let's return to the graphics figures unit described on page 86. The goal is to create a unit that exports several graphics figure objects (like *Point* and *Circle*) and a generalized means of dragging any of them around the screen. The unit, named *Figures*, is a simple implementation of the graphics toolbox discussed earlier. To demonstrate *Figures*, let's build a simple program that defines a new figure object type

unknown to *Figures* and then uses virtual methods to drag that new figure type around the screen.

Think about how graphics figures are alike and how they differ. The differences are obvious, and all involve shapes and angles and curves drawn onscreen. In the simple graphics program we'll describe, figures displayed onscreen share these attributes:

- They have a location, given as X,Y . The point within a figure considered to lie at this X,Y position is called the figure's *anchor point*.
- They can be either visible or invisible, specified by a Boolean value of True (visible) or False (invisible).

If you recall the earlier examples, these are precisely the characteristics of the *Location* and *Point* object types. *Point*, in fact, represents a sort of "grandparent" type from which all graphics figure objects are descended.

The rationale demonstrates an important principle of object-oriented programming: In defining a hierarchy of object types, gather all common attributes into a single type and allow the hierarchy of types to inherit all common elements from that type.

A note about abstract
objects



Type *Point* acts as a template from which its descendant object types can take elements common to all types in the hierarchy. In this example, no object of type *Point* is ever actually drawn to the screen, though no harm would come of doing so. (Calling *Point.Show* would obviously display a point on the screen.) An object type specifically designed to provide inheritable characteristics for its descendants is called an *abstract* object type. The point of an abstract type is to have descendants, not instances.

Go back to page 101 and read *Point* over once more, this time as a compendium of all the things that graphics figures have in common. *Point* inherits X and Y from the even earlier *Location* type, but *Point* contains X and Y nonetheless, and can bequeath them to its descendant types. Note that none of *Point*'s methods address the shape of a figure, but all figures can be visible or invisible and can be moved around on the screen.

Point also has an important function as a "broadcasting station" for changes to the object hierarchy *as a whole*. If some new feature is devised that applies to all graphics figures (color support, for example), it can be added to all object types descended from *Point* simply by adding the new features to *Point*. The new features are

instantly callable from any of *Point*'s descendant types. A method for moving a figure to the current position of the mouse pointer, for example, could be added to *Point* without changing any figure-specific methods, since such a method would only affect the two fields *X* and *Y*.

Obviously, if the new feature must be implemented differently for different figures, there must be a whole family of figure-specific virtual methods added to the hierarchy, each method overriding the one belonging to its immediate ancestor. Color, for example, would require minor changes to *Show* and *Hide* up and down the line, since the syntax of many GRAPH.TPU drawing routines depends on how drawing color is specified.

Procedure or method?

A major goal in designing the FIGURES.PAS unit is to allow users of the unit to extend the object types defined in the unit—and still make use of all the unit'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.

There are two ways to go about it. The way that might first occur to traditional Pascal programmers is to have FIGURES.PAS export a procedure that takes a polymorphic object as a **var** parameter, and then drags that object around the screen. Such a procedure is shown here:

```
procedure DragIt (var AnyFigure: Point; DragBy: Integer);
var
  DeltaX,DeltaY: Integer;
  FigureX,FigureY: Integer;
begin
  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) do
  begin                    { Apply delta to figure X,Y: }
    FigureX := FigureX + (DeltaX * DragBy);
    FigureY := FigureY + (DeltaY * DragBy);
    { And tell the figure to move }
    AnyFigure.MoveTo(FigureX, FigureY);
  end;
end;
```

DragIt calls an additional procedure, *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 keypad.)

What's important to notice about *DragIt* is that any object of type *Point* or any type descended from *Point* can be passed in the *AnyFigure* **var** parameter. Instances of *Point* or *Circle*, or any type defined in the future that inherits from *Point* or *Circle*, can be passed without complication in *AnyFigure*.

How does *DragIt*'s code know what object type is actually being passed? It doesn't—and that's OK. *DragIt* only references identifiers defined in type *Point*. By inheritance, those identifiers are also defined in any descendant of type *Point*. The methods *GetX*, *GetY*, *Show*, and *MoveTo* are just as truly present in type *Circle* as in type *Point*, and would be present in any future type defined as a descendant of either.

GetX, *GetY*, and *MoveTo* are static methods, which means that *DragIt* knows the procedure address of each at compile time. *Show*, on the other hand, is a virtual method. There is a different implementation of *Show* for both *Point* and *Circle*—and *DragIt* does not know at compile time which implementation is to be called. In brief, when *DragIt* is called, *DragIt* looks up the address of the correct implementation of *Show* in the VMT of the instance passed in *AnyFigure*. If the instance is a *Circle*, *DragIt* calls *Circle.Show*. If the instance is a *Point*, *DragIt* calls *Point.Show*. The decision as to which implementation of *Show* is called is not made until run time, and not, in fact, until the moment in the program when *DragIt* must call virtual method *Show*.

Now, *DragIt* works quite well, and if it is exported by the toolbox unit, it can drag any descendant type of *Point* around the screen, whether that type existed when the toolbox was compiled or not. But you have to think a little further: If any object can be dragged around the screen, why not make dragging a feature of the graphics objects themselves?

In other words, why not make *DragIt* a method?

Make it a method!

Indeed. Why pass an object to a procedure to drag the object around the screen? That's old-school thinking. If a procedure can be written to drag any graphics figure object around the screen,

then the graphics figure objects ought to be able to drag themselves around the screen.

In other words, procedure *DragIt* really ought to be method *Drag*.

Adding a new method to an existing object hierarchy involves a little thought. How far up the hierarchy should the method be placed? Think about the utility provided by the method 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. Metaphorically, you might think of a *Drag* method as *MoveTo* with an internal power source. In terms of inheritability, it sits right beside *MoveTo*—any object to which *MoveTo* is appropriate should also inherit *Drag*. *Drag* should thus be added to our abstract object type, *Point*, so that all *Point*'s descendants can share it.

Does *Drag* need to be **virtual**? The litmus test for making any method virtual is whether the functionality of the method is expected to change somewhere down the hierarchy tree. *Drag* is a closed-ended sort of feature. It only manipulates the *X,Y* position of a figure, and one doesn't imagine that it would become more than that. Therefore, it probably doesn't need to be virtual.

Still, you should use caution in any such decision: If you don't make *Drag* virtual, you lock out all opportunities for users of *FIGURES.PAS* to alter it in their efforts to extend *FIGURES.PAS*. You might not be able to imagine the circumstances under which a user might want to rewrite *Drag*. That doesn't for a moment mean that such circumstances will not arise.

For example, *Drag* has a joker in it that tips the balance in favor of its being virtual: It deals with *event handling*, that is, the interception of input from devices like the keyboard and mouse, which occur at unpredictable times yet must be handled when they occur. Event handling is a messy business, and often very hardware-specific. If your user has some input device that does not meld well with *Drag* as you present it, the user will be helpless to rewrite *Drag*. Don't burn any bridges: Make *Drag* virtual.

The process of converting *DragIt* to a method and adding the method to *Point* is almost trivial. Within the *Point* object definition, *Drag* is just another method header:

```

Point = object(Location)
  Visible: Boolean;
  constructor Init(InitX, InitY: Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible: Boolean;
  procedure MoveTo(NewX, NewY: Integer);
  procedure Drag(DragBy: Integer); virtual;
end;

```

The position of *Drag*'s method header in the *Point* object definition is unimportant. Remember, methods can be declared in any order, but data fields *must* be defined before the first method declaration.

The complete source code for FIGURES.PAS, including Drag implemented as a virtual method, is available on your disk.

Changing the procedure *DragIt* to the method *Drag* is almost entirely a matter of applying *Point*'s scope to *DragIt*. In the *DragIt* procedure, you had to specify *AnyFigure.Show*, *AnyFigure.GetX*, and so on. *Drag* is now a part of *Point*, so you no longer have to qualify method names. *AnyFigure.GetX* is now simply *GetX*, and so on. And of course, the *AnyFigure var* parameter is banished from the parameter line. The implied *Self* parameter now tells you which object instance is calling *Drag*.

By now, you should be thinking in terms of building functionality into objects in the form of methods rather than building procedures and passing objects to them as parameters. Ultimately you'll come to design programs in terms of activities that objects can do, rather than as collections of procedure calls that act upon passive data.

It's a whole new world.

Object extensibility

The important thing to notice about toolbox units like FIGURES.PAS is that the object types and methods defined in the unit can be distributed to users in linkable .TPU form only, without source code. (Only a listing of the interface portion of the unit need be released.) Using polymorphic objects and virtual methods, the users of the .TPU file can still add features to it to suit their needs.

This novel notion of taking someone else's program code and adding functionality to it *without benefit of source code* is called *extensibility*. Extensibility is a natural outgrowth of inheritance: You inherit everything that all your ancestor types have, and then

you add what new capability you need. Late binding lets the new meld with the old at run time, so the extension of the existing code is seamless and costs you no more in performance than a quick trip through the virtual method table.

FIGDEMO.PAS (on your disk) makes use of the *Figures* unit, and extends it by creating a new graphics figure object, *Arc*, as a descendant type of *Circle*. The object *Arc* could have been written long after FIGURES.PAS was compiled, and yet an object of type *Arc* can make use of inherited methods like *MoveTo* or *Drag* without any special considerations. Late binding and *Arc*'s virtual methods allows the *Drag* method to call *Arc*'s *Show* and *Hide* methods even though those methods might have been written long after *Point.Drag* itself was compiled.

Static or virtual methods

In general, you should make methods virtual. Use static methods only when you want to optimize for speed and memory efficiency. The tradeoff, as you've seen, is in extensibility.

Let's say you are declaring an object named *Ancestor*, and within *Ancestor* you are declaring a method named *Action*. How do you decide whether *Action* should be virtual or static? Here's the rule of thumb: Make *Action* virtual if there is a possibility that some future descendant of *Ancestor* will override *Action*, and you want that future code to be accessible to *Ancestor*.

Now apply this rule to the graphics objects you've seen in this chapter. In this case, *Point* is the ancestor object type, and you must decide whether to make its methods static or virtual. Let's consider its *Show*, *Hide*, and *MoveTo* methods. Since each different type of figure has its own means of displaying and erasing itself, *Show* and *Hide* are overridden by each descendant figure. Moving a graphics figure, however, seems to be the same for all descendants: Call *Hide* to erase the figure, change its *X,Y* coordinates, and then call *Show* to redisplay the figure in its new location. Since this *MoveTo* algorithm can be applied to any figure with a single anchor point at *X,Y*, it's reasonable to make *Point.MoveTo* a static method that is inherited by all descendants of *Point*; but *Show* and *Hide* are overridden and must be virtual so that *Point.MoveTo* can call its descendants' *Show* and *Hide* methods.

On the other hand, remember that if an object has any virtual methods, a VMT is created for that object type in the data segment

and every object instance has a link to the VMT. Every call to a virtual method must pass through the VMT, while static methods are called directly. Though the VMT lookup is very efficient, calling a method that is static is still a little faster than calling a virtual one. And if there are no virtual methods in your object, then there is no VMT in the data segment and—more significantly—no link to the VMT in every object instance.

The added speed and memory efficiency of static methods must be balanced against the flexibility that virtual methods allow: extension of existing code long after that code is compiled. Keep in mind that users of your object type might think of ways to use it that you never dreamed of, which is, after all, the whole point.

Dynamic objects

All the object examples shown so far have had static instances of object types that were named in a **var** declaration and allocated in the data segment and on the stack.

*The use of the word **static** here does not relate in any way to static methods.*

```
var
  ACircle: Circle;
```

Objects can be allocated on the heap and manipulated with pointers, just as the closely related record types have always been in Pascal. Turbo Pascal includes some powerful extensions to make dynamic allocation and deallocation of objects easier and more efficient.

Objects can be allocated as pointer referents with the *New* procedure:

```
var
  PCircle: ^Circle;

New(PCircle);
```

As with record types, *New* allocates enough space on the heap to contain an instance of the pointer's base type, and returns the address of that space in the pointer.

If the dynamic object contains virtual methods, it must then be initialized with a constructor call before any calls are made to its methods:

```
PCircle^.Init(600, 100, 30);
```

Method calls can then be made normally, using the pointer name and the reference symbol `^` (a caret) in place of the instance name that would be used in a call to a statically allocated object:

```
OldXPosition := PCircle^.GetX;
```

Allocation and initialization with New

Turbo Pascal extends the syntax of *New* to allow a more compact and convenient means of allocating space for an object on the heap and initializing the object with one operation. *New* can now be invoked with two parameters: the pointer name as the first parameter, and the constructor invocation as the second parameter:

```
New(PCircle, Init(600, 100, 30));
```

When you use this extended syntax for *New*, the constructor *Init* actually performs the dynamic allocation, using special entry code generated as part of a constructor's compilation. The instance name cannot precede *Init*, since at the time *New* is called, the instance being initialized with *Init* does not yet exist. The compiler identifies the correct *Init* method to call through the type of the pointer passed as the first parameter.

New has also been extended to allow it to act as a function returning a pointer value. The parameter passed to *New* is the *type* of the pointer to the object rather than the pointer variable itself:

```
type
  ArcPtr = ^Arc;

var
  PArc: ArcPtr;

PArc := New(ArcPtr);
```

Note that with version, the function-form extension to *New* applies to *all* data types, not only to object types:

```
type
  CharPtr = ^Char; { Char is not an object type... }

var
  PChar: CharPtr;

PChar := New(CharPtr);
```

The function form of *New*, like the procedure form, can also take the object type's constructor as a second parameter:

```
PArc := New(ArcPtr, Init(600, 100, 25, 0, 90));
```

A parallel extension to *Dispose* has been defined for Turbo Pascal, as fully explained in the following sections.

Fail helps you do error recovery in constructors; see the section "Constructor error recovery" in Chapter 17 of the Programmer's Guide.

Disposing dynamic objects

Just like traditional Pascal records, objects allocated on the heap can be deallocated with *Dispose* when they are no longer needed:

```
Dispose(PCircle);
```

There can be more to getting rid of an unneeded dynamic object than just releasing its heap space, however. An object can contain pointers to dynamic structures or objects that need to be released or "cleaned up" in a particular order, especially when elaborate dynamic data structures are involved. Whatever needs to be done to clean up a dynamic object in an orderly fashion should be gathered together in a single method so that the object can be eliminated with one method call:

```
MyComplexObject.Done;
```

The *Done* method should encapsulate all the details of cleaning up its object and all the data structures and objects nested within it.

It is legal and often useful to define multiple cleanup methods for a given object type. Complex objects might need to be cleaned up in different ways depending on how they were allocated or used, or depending on what mode or state the object was in when it was cleaned up.

We suggest the identifier Done for cleanup methods that "close up shop" once an object is no longer needed.

Destructors

Turbo Pascal provides a special type of method called a *destructor* for cleaning up and disposing of dynamically allocated objects. A destructor combines the heap deallocation step with whatever other tasks are necessary for a given object type. As with any method, multiple destructors can be defined for a single object type.

A destructor is defined with all the object's other methods in the object type definition:

```
Point = object (Location)
  Visible: Boolean;
  Next: PointPtr;
  constructor Init (InitX, InitY: Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible: Boolean;
  procedure MoveTo (NewX, NewY: Integer);
  procedure Drag (DragBy: Integer); virtual;
end;
```

Destructors can be inherited, and they can be either static or virtual. Because different shutdown tasks are usually required for different object types, it is a good idea *always* to make destructors virtual, so that in every case the correct destructor is executed for its object type.

Keep in mind that the reserved word **destructor** is not needed for every cleanup method, even if the object type definition contains virtual methods. Destructors really operate only on dynamically allocated objects. In cleaning up a dynamically allocated object, the destructor performs a special service: It guarantees that the correct number of bytes of heap memory are always released. There is, however, no harm in using destructors with statically allocated objects; in fact, by not giving an object type a destructor, you prevent objects of that type from getting the full benefit of Turbo Pascal's dynamic memory management.

Destructors really come into their own when polymorphic objects must be cleaned up and their heap allocation released. A polymorphic object is an object that has been assigned to an ancestor type by virtue of Turbo Pascal's extended type compatibility rules. In the running example of graphics figures, an instance of object type *Circle* assigned to a variable of type *Point* is an example of a polymorphic object. These rules govern pointers to objects as well; a pointer to *Circle* can be freely assigned to a pointer to type *Point*, and the referent of that pointer is also a polymorphic object.

The term *polymorphic* is appropriate because the code using the object doesn't know at compile time precisely what type of object is on the end of the string—only that the object is one of a hierarchy of objects descended from the specified type.

The size of object types differ, obviously. So when it comes time to clean up a polymorphic object allocated on the heap, how does *Dispose* know how many bytes of heap space to release? No information on the size of the object can be gleaned from a polymorphic object at compile time.

The destructor solves the problem by going to the place where the information is stored: in the instance variable's VMT. In every object type's VMT is the size in bytes of the object type. The VMT for any object is available through the invisible *Self* parameter passed to the method on any method call. A destructor is just a special kind of method, and it receives a copy of *Self* on the stack when an object calls it. So while an object might be polymorphic at *compile time*, it is never polymorphic at run time, thanks to late binding.

To perform this late-bound memory deallocation, the destructor must be called as part of the extended syntax for the *Dispose* procedure:

```
Dispose (PPoint, Done);
```

(Calling a destructor outside of a *Dispose* call does no automatic deallocation at all.) What happens here is that the destructor of the object pointed to by *PPoint* is executed as a normal method call. As the last thing it does, however, the destructor looks up the size of its instance type in the instance's VMT, and passes the size to *Dispose*. *Dispose* completes the shutdown by deallocating the correct number of bytes of heap space that had previously belonged to *PPoint*[^]. The number of bytes released is correct whether *PPoint* points to an instance of type *Point* or to one of *Point*'s descendant types like *Circle* or *Arc*.

Note that the destructor method itself can be empty and still perform this service:

```
destructor AnObject.Done;  
begin  
end;
```

What performs the useful work in this destructor is not the method body but the epilog code generated by the compiler in response to the reserved word **destructor**. In this, it is similar to a unit that exports nothing, but performs some "invisible" service by executing an initialization section before program startup. The action is all behind the scenes.

An example of dynamic object allocation

The final example program provides some practice in the use of objects allocated on the heap, including the use of destructors for object deallocation. The program shows how a linked list of graphics objects might be created on the heap and cleaned up using destructor calls when they are no longer required.

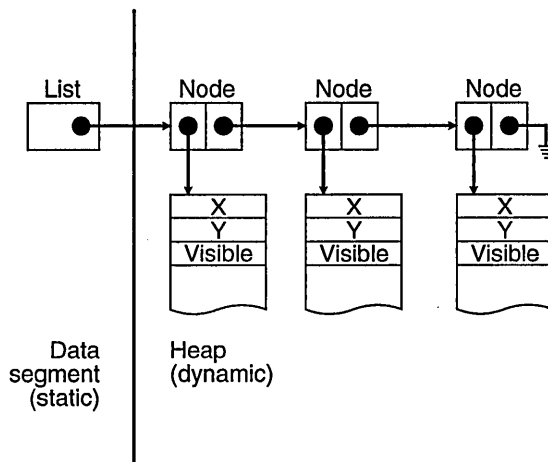
Building a linked list of objects requires that each object contain a pointer to the next object in the list. Type *Point* contains no such pointer. The easy way out would be to add a pointer to *Point*, and in doing so ensure that all *Point*'s descendant types also inherit the pointer. However, adding anything to *Point* requires that you have the source code for *Point*, and as said earlier, one advantage of object-oriented programming is the ability to extend existing objects without necessarily being able to recompile them.

The solution that requires no changes to *Point* creates a new object type not descended from *Point*. Type *List* is a very simple object 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 record type, *Node*, provides that service. *Node* is even simpler than *List*, in that it is not an object, has no methods, and contains no data except a pointer to type *Point* and a pointer to the next node in the list.

List has a method that allows it to add new figures to its linked list of *Node* records by inserting a new instance of *Node* immediately after itself, as a referent to its *Nodes* pointer field. The *Add* method takes a pointer to a *Point* object, rather than a *Point* object itself. Because of Turbo Pascal's extended type compatibility, pointers to any type descended from *Point* can also be passed in the *Item* parameter 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 descendants. The number of bytes of free heap 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, are cleaned up and removed from the heap with a single destructor call to the static *List* object, *AList*.

Figure 4.2
Layout of program
ListDemo's data structures



Disposing of a complex data structure on the heap

This destructor, *List.Done*, is worth a close look. Shutting down a *List* object involves disposing of three different kinds of structures: the polymorphic graphics figure objects in the list, the *Node* records that hold the list together, and (if it is allocated on the heap) the *List* object that heads up the list. The whole process is invoked by a single call to *AList*'s destructor:

```
AList.Done;
```

The code for the destructor merits examination:

```

destructor List.Done;
var
  N: NodePtr;
begin
  while Nodes <> nil do
  begin
    N := Nodes;
    Dispose(N^.Item, Done);
    Nodes := N^.Next;
    Dispose(N);
  end;
end;

```

The list is cleaned up from the list head by the "hand-over-hand" algorithm, metaphorically similar to pulling in the string of a kite: Two pointers, the *Nodes* pointer within *AList* and a working pointer *N*, alternate their grasp on the list while the first item in

the list is disposed of. A dispose call deallocates storage for the first *Point* object in the list (*Item*[^]); then *Nodes* is advanced to the next *Node* record in the list by the statement *Nodes := N[^].Next*; the *Node* record itself is deallocated; and the process repeats until the list is gone.

The important thing to note in the destructor *Done* is the way the *Point* objects in the list are deallocated:

```
Dispose (N^.Item, Done);
```

Here, *N[^].Item* is the first *Point* object in the list, and the *Done* method called is its destructor. Keep in mind that the actual type of *N[^].Item[^]* is not necessarily *Point*, but could as well be any descendant type of *Point*. The object being cleaned up is a polymorphic object, and no assumptions can be made about its actual size or exact type at compile time. In the earlier call to *Dispose*, once *Done* has executed all the statements it contains, the “invisible” epilog code in *Done* looks up the size of the object instance being cleaned up in the object’s VMT. *Done* passes that size to *Dispose*, which then releases the exact amount of heap space the polymorphic object actually occupied.

Remember that polymorphic objects must be cleaned up this way, through a destructor call passed to *Dispose*, if the correct amount of heap space is to be reliably released.

In the example program, *AList* is declared as a static variable in the data segment. *AList* could as easily have been itself allocated on the heap, and anchored to reality by a pointer of type *ListPtr*. If the head of the list had been a dynamic object too, disposing of the structure would have been done by a destructor call executed within *Dispose*:

```
var
  PList: ListPtr;
...
Dispose (PList, Done);
```

Here, *Dispose* calls the destructor method *Done* to clean up the structure on the heap. Then, once *Done* is finished, *Dispose* deallocates storage for *PList*’s referent, removing the head of the list from the heap as well.

LISTDEMO.PAS (on your disk) uses the same FIGURES.PAS unit described on page 108. It implements an *Arc* type as a descendant of *Point*, creates a *List* object heading up a linked list of three polymorphic objects compatible with *Point*, and then disposes of the

whole dynamic data structure with a single destructor call to *AList.Done*.

Where to now?

As with any aspect of computer programming, you don't get better at object-oriented programming by reading about it; you get better at it by doing it. Most people, on first exposure to object-oriented programming, are heard to mutter "I don't get it" under their breath. The "Aha!" comes later, when in the midst of putting their own objects in place, the whole concept comes together in the sort of perfect moment we used to call an epiphany. Like the face of woman emerging from a Rorschach inkblot, what was obscure before at once becomes obvious, and from then on it's easy.

The best thing to do for your first object-oriented project is to take the *FIGURES.PAS* unit (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 objects with 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 combination figures 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've had your "Aha!" start building object-oriented concepts into your everyday programming chores. Take some existing utilities you use every day and rethink them in object oriented terms. Take another look at your hodgepodge of procedure libraries and try to see the objects in them—then rewrite the procedures in object form. You'll find that libraries of objects 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 an object 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

Object-oriented programming 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. (It's the difference between having ten thousand insects classified in a taxonomy chart, and ten thousand insects all buzzing around your ears.) Far more than structured programming, object-orientation 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 the whole thing begins to sound almost too good to be true. Impossible, you think?

Hey, this is Turbo Pascal. "Impossible" is undefined.

Debugging Turbo Pascal programs

Turbo Pascal's superb development environment includes automatic project management, program modularity, high-speed compilation, and easy-to-use overlays. Yet with all that, your program can still have *bugs*, or errors, that keep it from working correctly.

Turbo Pascal gives you the tools you need to *debug* your program, which means to find and remove all the errors to get it running. Turbo Pascal also makes it easy to locate and fix compiler and run-time errors. And it lets you enable or disable automatic error checking at run time.

Turbo Pascal comes with a powerful, flexible source-level debugger that allows you to execute your program one line at a time, viewing expressions and modifying variables as you go. This debugger is built into the Turbo Pascal IDE; you can edit, compile, and debug without ever leaving Turbo Pascal. And for big or complex programs that require the full range of debugging support from machine language to evaluating Pascal expressions, Turbo Pascal fully supports Borland's standalone debugger, Turbo Debugger.

Taxonomy of bugs

There are three basic types of program bugs: compile-time errors, run-time errors, and logic errors.

Compile-time

errors

A compile-time, or *syntax*, error occurs when you violate a rule of Pascal syntax: leave out a semicolon, forget to declare a variable, pass the wrong number of parameters to a procedure, assign a real value to an integer variable. What it really means is that you're writing statements that don't follow the rules of Pascal.

Turbo Pascal won't compile your program (generate machine code) until all your syntax errors are gone. If Turbo Pascal finds a syntax error while it is compiling your program, it stops compiling, goes into your source code, locates the error, positions the cursor there, and displays an error message in the Edit window. Once you've corrected it, you can start compiling again.

For more about error messages, refer to Appendix A in the Programmer's Guide.

If you're using the command-line version (TPC.EXE), Turbo Pascal will print out the offending statement, along with the line number and the error message. You can then go into whatever editor you're using, find the line, fix the problem, and recompile.

Run-time errors

A run-time, or *semantic*, error happens when you compile a syntactically legal program that does something illegal when it executes, such as opening a nonexistent file for input or dividing by 0. In that case, Turbo Pascal halts your program and prints an error message to the screen that looks like this:

```
Run-time error ## at seg:ofs
```

If you're running in the IDE, Turbo Pascal automatically finds the location of the run-time error, pulling in the appropriate source file.

See Chapter 9, "The command-line compiler," for a complete explanation of using TPC.EXE to find run-time errors.

If you ran your program from the DOS prompt, you'll be returned to DOS. You can load TURBO.EXE and use **Search | Find Error** to locate the position in your source (make sure **Destination** is set to **Disk**). You can also use the command-line compiler (TPC.EXE) **/F** option to find the error.

Logic errors

Logic errors mean that your program does what you *told* it to do instead of what you *want* it to do. A variable may not have been initialized; calculations may turn out wrong; pictures drawn

onscreen don't look right; or the program might just skip doing what you think it should.

These can be the hardest errors to find, but they are the ones that the integrated debugger helps you with the most.

The integrated debugger

Some run-time and logic errors are obscure and hard to track down. Others can be buried by subtle interactions between sections of a large program. In these cases, what you'd really like to do is to execute your program interactively, watching the values of certain variables or expressions. You'd like your program to stop when it reaches a certain place so that you can see just how it got there. You'd like to stop and change the values of some variables while the program is executing, to force a certain behavior or see how the program responds. And you'd like to do this in a setting where you can quickly edit, recompile, and run your program again.

Turbo Pascal's integrated debugger has all the capabilities just described and more. It is an integral part of the Turbo Pascal IDE: Two of the main menu items (**R**un and **D**ebug) are devoted to its use; likewise, several hot keys are used for debugger commands. For more about the IDE and hot keys, refer to Chapter 7, "The IDE reference," or try TPTOUR or online help.

What the debugger can do

The integrated debugger performs in an uncomplicated manner. There are no special instructions in your code, no increase in the size of your .EXE file, and no need to recompile to create a standalone .EXE once you've finished debugging.

If your program is divided into a number of units, the source code for each is automatically loaded into the editor as you trace execution.

If you use overlays, the debugger handles them automatically within the IDE, smoothly switching back and forth between the compiler, the editor, and the debugger.

Here's an overview of the debugger's features:

Tracing

F7

Run|Trace Into You can execute one line in your program, then pause to see the results. When procedures or functions within your program are called, you have the option of executing the call as a single step, or of tracing through that routine line by line.

You can also trace your program's *output* line by line. You can have it swap screens as needed, or use dual monitors. You can also bring up the output screen in a separate window.

Go to cursor

F4

Run|Go to Cursor You can move the cursor to a specific line in your program, then tell the debugger to execute your program until it reaches that line. This makes it easy to skip over loops and other tedious sections of code; it also lets you go right to the spot where you want to start debugging.

Breaking

Debug|Breakpoints You can mark lines in your program as *breakpoints*. When you run your program and it comes to a breakpoint, it stops and displays the source code with the breakpoint in the execution bar. You can then examine variables, start tracing, or run the program until another breakpoint is encountered. You can attach a condition to a breakpoint. You can also break at any point during program execution by pressing *Ctrl-Break*. This has the effect of stopping at the next source line, as if a breakpoint had been set there.

Watching

Debug|Watches You can set up a number of *watches* in the Watch window. Each one can be a variable, data structure, or expression. The watches change to reflect their current values as you step through your program.

Evaluate/Modify

Ctrl

F4

Debug|Evaluate/Modify You can bring up the Evaluate and Modify box, which lets you interactively examine the value of variables, data structures, and expressions. You can change the value of any variable, including strings, pointers, elements of an array, and fields of a record. This provides an easy mechanism for testing how your code reacts to certain sets of values or conditions.

Navigating You can quickly locate procedure or function declarations, even if your program is broken up into many modules (**Search | Find Procedure**). During a trace, you can quickly scroll back through the procedure or function call(s) that led to where you are and examine the parameters for each call (**Window | Call Stack**).

In and out of the debugger

Before you start debugging, you should understand that the basic unit of execution in the debugger is a *line*, not a statement. More accurately, the *smallest* unit of execution is a line. If you have several Pascal statements on a single line, they will all be executed together with a single press of *F7*. If, on the other hand, you have a single statement spread out over several lines, then the entire statement will be executed by pressing *F7* once. All the execution commands are based on lines, including single-stepping and breakpoints; the line about to be executed is always shown in the execution bar.

A symbol table is a small internal database of all the identifiers that are used—constants, types, variables, procedures, and line-number information.

Before you start debugging a program, the compiler must be able to generate the necessary symbol table and line-number information for your programs. The debugging compiler directives **\$L+** and **\$D+** that do this are on by default; they correspond to the menu items **Options | Compiler | Local Symbols** and **Options | Compiler | Debug Information**, respectively. Also checked by default is the **Options | Debugger | Integrated** option, which generates debugging information in the executable file.

You can turn these switches off to conserve memory or disk space during compilation.

{\$D+} generates line-number tables that map object code to source positions. **{\$L+}** generates local debug information, which means it creates a list of the identifiers local to each procedure or function, so that the debugger can “remember” them while you’re debugging. When you use the compiler directives, separate them by a comma and *no spaces*, and precede *only the first directive* by a **\$**; for example, **{\$D+,L+}**.

When you step through your program, Turbo Pascal will sometimes swap to the User screen, execute your code, then swap back to the integrated environment to await your next command. You can control when this screen swap occurs with the **Options | Debugger | Display Swapping** setting, which has three possible values:

- When the **Smart** option is on (it is by default), the IDE only swaps to the User screen when a program line accesses video RAM or when a subroutine is stepped over.
- When the **Always** option is on, the User screen is swapped with each step.
- When the **None** is on, no display swapping ever occurs, and the IDE remains visible at all times. If the program writes to the screen or if user input is required, the text will overwrite the IDE screen. You can have Turbo Pascal repaint its windows by choosing **≡ | Refresh Display**.

Starting a debugging session

The quickest way to start debugging is to load in your program and choose **Run | Trace Into (F7)**. Your program gets compiled, and when it's finished, the editor will display the main body of your program, with the execution bar on the initial **begin**. You can continue to trace from there (using **F7** and **F8**), or you can use the other methods we describe here.

If you know where in the program you want to start debugging, you can have your program execute until it reaches that spot, then have it pause there. To do this, just bring up that section of code in the editor and move the cursor to the line where you want to stop. You can then do one of two things:

- You can choose **Run | Go to Cursor** (or press **F4**), which will execute your program until it reaches that point, then pause.
- You can set a breakpoint there (choose **Debug | Toggle Breakpoint** or press **Ctrl-F8**), and then run your program (choose **Run | Run** or press **Ctrl-F9**); it will now stop *every* time it reaches that line. You can set several breakpoints, in which case your program will stop whenever it comes to any of the breakpoints.

Restarting a debugging session

If you're in the middle of debugging a program and want to start all over again, choose the **Program Reset** command from the **Run** menu. This reinitializes the debugging system so the next step command will take you to the first line in the main body of your program. At the same time, it closes any files your program may have opened, clears the stack of any nested subroutine calls, and releases any heap space being used. It does *not* reinitialize or otherwise modify any variables (Turbo Pascal never initializes variables automatically); typed constants, however, are restored to their original values.

Turbo Pascal will also offer a restart if you make any changes to the program itself while debugging. For example, if you modify any part of the program, then press any execution command (*F7*, *F8*, *F4*, *Ctrl-F9*, and so on), you'll get a box with the message "Source modified, rebuild? (Y/N)." If you press *Y*, Turbo Pascal will re-make your program and start debugging from the beginning. If you press *N*, Turbo Pascal assumes you know what you're doing and continues the debug session in progress. (Any source code changes you made will *not* affect program execution until you recompile. If you added or deleted lines, the execution bar will *not* compensate for these changes and may appear to highlight the wrong line.)

Ending a debugging session

While you're debugging a program, Turbo Pascal keeps track of where you are and what you're doing. And since you can load and even edit different files while you're debugging, Turbo Pascal does not interpret loading a different file into the editor as "ending" a debugging session. So, if you want to run or debug a different program, let Turbo Pascal know by choosing the **Run | Program Reset** command (*Ctrl-F2*).

Tracing through your program

The simplest debugging technique is *single-step tracing*, which traces into procedures and functions. Load the following program (RANGE.PAS) in Turbo Pascal:

```
{ $D+,L+ }      { To be sure complete debug information is generated }
{ $R- }         { To be sure range checking is off }
program RangeTest;
var
  List: array[1..10] of Integer;
  Indx: Integer;
begin
  for Indx := 1 to 10 do
    List[Indx] := Indx;
  Indx := 0;
  while (Indx < 11) do
    begin
      Indx := Indx + 1;
      if List[Indx] > 0 then
        List[Indx] := -List[Indx];
    end;
  for Indx := 1 to 10 do
    Writeln(List[Indx]);
end.
```

The execution bar indicates the next line of the program to be run.

To start debugging, press *F7*. You're asking Turbo Pascal to execute the first line in the main body of your program. Note that the execution bar is on the **begin** on line 7. Since you haven't compiled your program yet, Turbo Pascal does it automatically, and then prepares to single-step your program.

Press *F7* a few more times. The execution bar moves to `List [Indx] := Indx;`, and appears to stay there. What's happening is that this line is executing in a loop.

Choose the **Debug | Watches | Add Watch** command (*Ctrl-F7*) to display the Add Watch box. You're going to monitor the values within your program by setting a watch; a *watch* is a variable, data structure, or expression.

What appears in the Add Watch box depends on where your cursor is positioned when you press *Ctrl-F7*. If you position the cursor on the first letter of any alphanumeric string, within it, or immediately following it, the string will be copied to the Add Watch box and highlighted. So, if the cursor is positioned on *Indx*, *Indx* will appear in the box. To change what's in the box, start typing and the original expression and the highlight will disappear.

Once the Add Watch box is displayed, regardless of its contents, you can add more to it by pressing the \rightarrow key (which copies more text from the editor). Place *List* in the box by using the \rightarrow and press *Enter*. A line like the following will appear in the Watch window at the bottom of your screen:

```
• List: (1,2,0,0,0,0,0,0,0)
```

Now, press *Ctrl-F7* again and type *Indx* and press *Enter*. *Indx* is listed first in the Watch window, making it look something like this:

```
• Indx: 3  
  List: (1,2,0,0,0,0,0,0,0)
```

Now press *F7* again, and you'll see the values of *Indx* and *List* change in the Watch window, reflecting what's happening in your program.

As you enter the **while** loop, you'll again see the values of *Indx* and *List* change, step by step. Note that the change in the Watch window reflects the actions of each line after you press *F7*.

Keep pressing *F7* until you're at the top of the **while** loop, with *Indx* equal to 10. This time through the loop, press *F7*, slowly

watching how the values in the Watch window change. When you execute the statement

```
List[Indx] := -List[Indx];
```

the value of *Indx* changes to -11. If you continue to press *F7*, you'll find that what you have is an infinite loop.

This program *will* compile and run. And run. And run. It gets stuck in an infinite loop because the **while** loop executes 11 times, not 10, and the variable *Indx* has a value of 11 the last time through the loop. Since the array *List* only has 10 elements in it, *List[11]* points to some memory location outside of *List*. Because of the way variables are allocated, *List[11]* happens to occupy the same space in memory as the variable *Indx*. This means that when *Indx* = 11, the statement

```
List[Indx] := -List[Indx];
```

is equivalent to

```
Indx := -Indx
```

Since *Indx* equals 11, this statement sets *Indx* to -11, which starts the program through the loop again. That loop now changes additional bytes elsewhere, at the locations corresponding to *List[-11..0]*. And because *Indx* never ends the loop at a value greater than or equal to 11, the loop never ends.

The important point is that, in just a few minutes and using only two keystrokes (*F7* and *Ctrl-F7*), you quickly and easily tracked down a subtle, nasty bug.

Stepping through your program

Trace Into is one debugging technique; Step Over (*F8*) is yet another, one that "steps over" subroutine calls. Both Step Over and Trace Into have a special meaning at the **begin** statement of the main program if your program uses units with initialization code. In this case, *F7* will step into each unit's initialization code, allowing you to see how each one of your units is set up. *F8* will step over the initialization code, leaving the execution bar on the next executable line after the **begin**.

Consider the following (incomplete) sample program:

```
{SD+,L+}  
program TestSort;  
const
```

```

    NLMMax = 100;
type
    NumList = array[1..NLMMax] of Integer;
var
    List: NumList;
    I, Count: Word;

procedure Sort(var L: NumList; C: Word);
begin
    { sort the list }
end; { of proc Sort }

begin
    Randomize;
    Count := NLMMax;
    for I := 1 to Count do
        List[I] := Random(1000);
    Sort(List, Count);
    for I := 1 to Count do
        Write(List[I]:8);
    Readln;
end. { of program TestSort }

```

Suppose you're debugging the *Sort* procedure. You want to trace your call to *Sort*, including checking the values within *List* before calling it. However, it gets tedious stepping through that first **for** loop 100 times as it initializes *List*. There must be a way you can get the loop to execute without having to single-step each line.

In fact, there are a few ways. First, you could put it in a separate procedure and press *F8* when you get to it, but that's a bit drastic. Second, you could set a *breakpoint* within your program, which is a place in your program where you want execution to run *to*, then stop *at*. Finally, you could move the cursor to the line calling *Sort* and choose the **Run | Go to Cursor** command (*F4*). Your program will execute until it gets to the line containing the cursor. The execution bar will move to that line; you can start tracing from there, in this case by pressing *F7* to trace into *Sort*.

Run | Go to Cursor (*F4*) works through multiple levels of subroutine calls, even if the source code is in another file. For example, you could place the cursor somewhere within *Sort* and press *F4*; the program would execute until it reached that line within *Sort*. For that matter, *Sort* could be in a separate unit, and the debugger would still know when to stop and what to display.

There are three cases where **Go to Cursor** (*F4*) will not run to the line containing the cursor.

1. When you have the cursor positioned between two executable lines, for example, a blank line or a comment line within a code block, the program will run to the next line containing executable statements.
2. When you have the cursor positioned outside the scope of a procedure block, for example, on the program statement or variable declarations, The debugger will tell you there is "no code generated for this line."
3. When you position the cursor on a line that will never gain control; for example, the line *above* the execution bar (assuming you're not in a loop) or the **else** part of a conditional statement when the **if** expression is true, the debugger will behave as if you had chosen **Run | Run (Ctrl-F9)**. Your program will run until it terminates or until a breakpoint occurs.

Let's say that you trace through *Sort* for a while, then want the program to finish executing so you can see the output. How would you do this? First, you would move the cursor to the final **end** statement in the main body of the program, then choose **Run | Go to Cursor (F4)**. Or you could choose **Run | Run (Ctrl-F9)**, which will tell the debugger to let your program continue normal execution. Your program will then run until it ends or hits a breakpoint that you've set, or until you press *Ctrl-Break*.

Using breakpoints

You can have up to 16 breakpoints active at a time.

Breakpoints only exist during your debugging session; they aren't saved in your .EXE file if you compile your program to disk.

Note that you cannot see the breakpoint highlight when the execution bar is on the breakpoint line.

Breakpoints are an important part of debugging. They're like having a stop sign embedded in your program: When your program encounters one, it stops execution and waits for further debugging instructions.

To set a breakpoint, move the cursor to each line in your program where you want it to pause. Any line where a breakpoint is set should contain at least one executable statement. It should not be a blank line, a comment, or a compiler directive; a constant, type, label, or variable declaration; a program, unit, procedure, or function header. To set a line as a breakpoint, choose the **Debug | Toggle Breakpoint** command (*Ctrl-F8*), which highlights it.

Once you've set your breakpoints, execute your program by choosing **Run | Run** (or pressing *Ctrl-F9*). Your program executes normally until a breakpoint is encountered. Then the program halts, the appropriate source code file (main program, unit, or

Include file) is loaded in, and the Edit window is displayed with the execution bar on top of the breakpoint line. If any variables or expressions have been added to the Watch window, they are also displayed with their current values.

At this point, you can use any number of debugging options.

- You can step through your code using **Run | Trace Into**, **Step Over**, or **Go to Cursor** (*F7*, *F8*, or *F4*). You can examine and modify variables.
- You can add or remove expressions from the Watch window.
- You can set or clear breakpoints.
- You can view program output with **Windows | User Screen** (*Alt-F5*).
- You can re-start your program from the beginning (using **Run | Program Reset** and then a step command).
- You can continue execution to the next breakpoint (or to the end of the program) by choosing **Run | Run** (*Ctrl-F9*).

To clear a breakpoint from a line, move the cursor to the line and choose **Debug | Toggle Breakpoint** (*Ctrl-F8*) again. This command toggles the breakpoint line on and off; if you use it on a breakpoint line, that line returns to normal.

Let's go back to the earlier example:

```
begin { main body of TestSort }
  Randomize;
  Count := NLMMax;
  for I := 1 to Count do
    List[I] := Random(1000);
  Sort(List, Count);
  for I := 1 to Count do
    Write(List[I]:8);
  Readln;
end. { of program TestSort }
```

The idea here was to skip over the initial loop and start tracing with the call to *Sort*. The new solution is to move the cursor to the line calling *Sort* and choose **Debug | Toggle Breakpoint** (*Ctrl-F8*), making it a breakpoint. Now, run to the breakpoint by choosing **Run | Run** (*Ctrl-F9*). When the program gets to that line, it will stop and allow you to begin debugging.

Using Ctrl-Break In addition to any breakpoints you might set, you also have an “instant” breakpoint during execution: pressing *Ctrl-Break*. This means that, barring a major crash, you can interrupt your program at any time. When you press *Ctrl-Break*, you drop out of your program and back into the editor, with the execution bar on the next line and ready for single-stepping.

What actually happens is that the debugger hooks itself into DOS, the BIOS, and other services. In this way, it knows whether or not the code currently executing is a DOS routine, BIOS routine, or your program. When you press *Ctrl-Break*, the debugger waits until the program itself is executing. It then starts stepping every machine-level instruction until the next instruction is at the beginning of a Pascal source code line. At that point, it breaks, moves the execution bar to that line and prompts you to press *Esc*.

▣▣▣▣▣ If a second *Ctrl-Break* is detected before the debugger locates and displays the source code line, then the debugger terminates the program and doesn't try to find the source line. In such a case, the exit procedures are *not* executed, which means that files, video mode, and DOS memory allocations might not be completely cleaned up.

Watching values

Program flow tells you a lot, but not as much as you'd like. What you really want to do is watch how variables change as your program executes. Suppose the *Sort* procedure for the earlier program looked like this:

```
procedure Sort(var L: NumList; C: Word);
var
  Top,Min,K: Word;
  Temp: Integer;
begin
  for Top := 1 to C-1 do
  begin
    Min := Top;
    for K := Top+1 to C do
      if L[K] < L[Min] then
        L[Min] := L[K];
    if Min <> Top then
      begin
        Temp := L[Top];
        L[Top] := L[Min];
```

Change *NLMax* in the body of the program to 10, so that you're working with a smaller array.


```

        L[Min] := Temp;
    end;
end;
end; { of proc Sort }

```

There is a bug here, so step through it (using **Run | Trace Into** or **F7**) and watch the values of *L*, *Top*, *Min*, and *K*.

The debugger lets you set up watches to monitor values within your program as it executes. The current value of each watch is shown, updated as each line in the program executes.

Set up a watches for each identifier using **Debug | Watch | Add Watch (Ctrl-F7)** to add each expression to the Watch window. The result might look like this:

```

· K: 21341
  Min: 51
  Top: 21383
  L: (163,143,454,622,476,161,850,402,375,34)

```

This presumes you've just stepped into *Sort* and the execution bar is on the initial **begin** statement. (If you haven't stepped into *Sort* yet, "Unknown Identifier" will be displayed next to each Watch expression until you do.) Note that *K*, *Min*, and *Top* just have random values, since they haven't been initialized yet. The values in *L* are supposed to be random; they won't look just like this when you run the program, but they will all be non-negative values from 0 to 999.

Pressing **F7** four times will move you down to the line **if L[K] < L[Min] then**, where you'll notice that *K*, *Min*, and *Top* now have values of 2, 1, and 1, respectively. Keep pressing **F7** until you drop out of that inner **for** loop, through the **if Min <> Top then** line, back to the top of the outer loop, and down again to **if L[K] < L[Min] then**. At this point, the Watch window would look like this (given the previous values in *L*):

```

· K: 3
  Min: 2
  Top: 2
  L: (34,143,454,622,476,161,850,402,375,34)

```

By now, you may have noticed two things. First, the last value in *L* (34)—which also happens to be the lowest value—got copied into the first location in *L*, and the value that was there (163) has disappeared. Second, *Min* and *Top* were the same value all the way through. In fact, if you look closely, you'll notice something else: *Min* gets assigned the value of *Top*, but is never changed anywhere else. Yet the test at the bottom of the loop is **if Min <>**

Top **then**. Either you have the wrong test, or there's something wacky between those two sections of code.

As it turns out, the bug is in the fifth line of code: It should read `Min := K;` instead of `L[Min] := L[K];`. Correct it, move the cursor to the initial **begin** in *Sort*, and choose **Run | Go to Cursor (F4)**. Since you've changed the program, a box will appear with the question "Source modified, rebuild? (Y/N)"; press **Y**. Your program will recompile, start running, then pause at the initial **begin** in *Sort*. This time, the code works correctly: Instead of overwriting the first location with the lowest value, it swaps values, moving the value in the first location to the position where the lowest value was previously. It then repeats the process with the second location, the third, and so on, until the list is completely sorted.

Types of watch expressions You can put any kind of constant, variable, or data structure in the Watch window as an expression; you can also put in Pascal expressions. Specifically, here are the expressions you can add and how each will be displayed:

Expression	Display
Integers	Decimal and hex. Examples: -23 \$10
Reals	Without an exponent, if possible. Examples: 38328.27 6.283e23
Characters	Printable: in single quotes (including extended graphics characters) as themselves. Control characters: as ASCII codes or printable. Examples: 'b' 'Ø' #4
Booleans	True or False
Enumerated data values	Actual named values (all uppercase). Examples: RED JAN WEDNESDAY
Pointers	<i>segment:offset</i> hex format. Examples: PTR(\$3632,\$106) PTR(CSEG,\$220)
Strings	In single quotes. Examples: 'Bruce'
Arrays	In parentheses, separated by commas. Multidimensional arrays as nested lists. Examples: (-42,23,2292,0,684)
Records	In parentheses, fields separated by commas. Nested records as nested lists. Examples: (5,10,'Borland',RED,TRUE)
Objects	Same as record. Expressions valid for records are also valid for objects.
Sets	In brackets, with expressions separated by commas; subranges are used when possible. Examples: [MON,WED,FRI] ['0'..'9','A'..'F']
Files	In (<i>status,fname</i>), where <i>status</i> is CLOSED, OPEN, INPUT, or OUTPUT, and <i>fname</i> is name of disk file assigned to file variable. Examples: (OPEN,'BUDGET.DTA')

Format specifiers

To control exactly how information is displayed in the Watch window, Turbo Pascal allows you to add *format specifiers* to your Watch expressions. A format specifier follows the Watch expression, separated from it by a single comma. (You don't need format specifiers to debug; this is an advanced topic.)

A complete list of the available format specifiers and their effects are on page 195.

A format specifier consists of an optional *repeat count* (an integer), followed by zero or more format characters; no spaces are required between the repeat count and the format characters. The repeat count is used to display consecutive variables, such as the elements of an array. For example, assuming *List* is an array of 10 integers, the Watch expression *List* would display:

```
List: (10,20,30,40,50,60,70,80,90,100)
```

If you want to look at a particular range of the array, you can specify the index of the first element, and add a repeat count:

```
List[6],3: 60,70,80
```

This technique is particularly useful for dealing with arrays that are too large to be displayed completely on a single line.

Repeat counts aren't limited to arrays; any variable may be followed by a repeat count. The general syntax *var, x* simply displays *x* consecutive variables of the same type as *var*, starting at the address of *var*. Note however, that the repeat count is ignored if the Watch expression does not denote a variable. A good rule of thumb is that a given construct is a variable if it can legally appear on the left-hand side of an assignment statement, or be used as a **var** parameter to a procedure or function.

To demonstrate the use of format specifiers, assume that the following types and variables have been declared:

```
type
  NamePtr = ^NameRec;
  NameRec = record
    Next: NamePtr;
    Count: Integer;
    Name: string[31];
  end;
var
  List: array[1..10] of Integer;
  P: NamePtr;
```

Given these declarations, the following Watch expressions can be constructed:

```

List: (10,20,30,40,50,60,70,80,90,100)
List[6],3H: $3C,$46,$50
P: PTR($3EAO,$C)
P,P: 3EAO:000C
P^: (PTR($3EF2,$2),412,'John')
P^,R$: (NEXT:PTR($3EF2,$2);COUNT:$19C;NAME:'John')
P^,Next^,R: (NEXT:NIL;COUNT:377;NAME:'Joe')
Mem[$40:0],10M: F8 03 F8 02 00 00 00 00 BC 03
Mem[$40:0],10MD: 248 3 248 2 0 0 0 0 188 3

```

Typecasting *Typecasting* is another powerful feature you can use to modify how Watch expressions are displayed, letting you interpret data as a different type than it would normally be. This can be especially useful if you're working with an address or a generic pointer, and you want to view it as pointing to a particular data type.

Suppose your program has a variable *DFile* that is of type `file` of `MyRec`, and you execute the following sequence of code:

```

Assign(DFile,'INPUT.REC');
Reset(DFile);

```

If you add *DFile* as a watch, the corresponding line in the Watch window will look like this:

```
DFile: (OPEN,'INPUT.REC')
```

But you might want more information about the file record itself. If you change your program so that it uses the *Dos* unit, then you can modify the *DFile* watch to `FileRec(DFile),rh`, which means, "Display *DFile* as if it were a record of type `FileRec` (declared in the *Dos* unit), with all record fields labeled and all integer values displayed in hexadecimal." The result in the Watch window might look something like this:

```
FileRec(DFile),rh: (HANDLE:$6;MODE:$D7B3;RECSIZE:$14;PRIVATE:($0,$0,...))
```

The record is too large to view at once; however, you can use the cursor movement keys to scroll the data not visible on the screen (see the section "Editing and deleting watches" on page 139).

With this typecasting, you can now watch specific fields of *DFile*. For example, you could view the *UserData* field by adding the expression `FileRec(DFile).UserData` to the Watch window:

```
FileRec(DFile).UserData: (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

You can apply the same technique to data structures and types of your own design. If they're declared in your program or units, you can typecast to them in the Watch window. The rules for typecasting are explained in Chapter 6 of the *Programmer's Guide*, "Expressions."

Expressions As we mentioned earlier, you can use *expressions* as Watch expressions; you could have calculations, comparisons, address offsets, and other such expressions. Table 5.1 lists the kinds of features legal in a Watch expression, as well as acceptable values.

Table 5.1: Watch expression values

Legal in a Watch Expression	Acceptable Values
Literals and Constants	All normal types: Boolean, Byte, Char, enumerated, Integer, Longint, Real, Shortint, string, and Word.
Variables	All types, including user-defined types and elements of data structures:
integer-type	Any integer expression within the variable's range bounds.
floating-point	Any floating-point (or integer) expression within the variable's exponent range; excess significant digits are dropped.
Char	Any character expression, including any printable character surrounded by single quotes; integer expressions typecast to Char using <i>Chr</i> or <i>Char()</i> ; ASCII constants (#, followed by any value from 0 to 255).
Boolean	True and False; any Boolean expression.
enumerated data type	Any compatible enumerated constant; in-range integer expressions typecast to a compatible enumerated type.
Pointer	Any compatible pointer; any compatible typecast expression; the function <i>Ptr</i> (with appropriate parameters).
string	Any string constant (text enclosed by single quotes); string variables; string expressions consisting of string constants and variables concatenated with the + operator.
set	Any set constant (compatible elements surrounded by square brackets); any compatible set expression, including the use of set operators +, -, *.
Typecasts	Following standard Pascal rules.
Operators	All normal Pascal operators, plus Turbo Pascal extensions such as xor , @ , and so on.
Built-In Functions	<i>Abs</i> , <i>Addr</i> , <i>Chr</i> , <i>CSeg</i> , <i>DSeg</i> , <i>Hi</i> , <i>IOResult</i> , <i>Length</i> , <i>Lo</i> , <i>MaxAvail</i> , <i>MemAvail</i> , <i>Odd</i> , <i>Ofs</i> , <i>Ord</i> , <i>Pred</i> , <i>Ptr</i> , <i>Round</i> , <i>Seg</i> , <i>SizeOf</i> , <i>SPtr</i> , <i>SSeg</i> , <i>Succ</i> , <i>Swap</i> , and <i>Trunc</i> .
Arrays	<i>Mem</i> , <i>MemL</i> , <i>MemW</i>

In other words, the expression must be a normal, legal Pascal expression, and can use any or all of the features described in

Table 5.1. See “Modification issues” on page 140 for information on how to modify Watch expressions.

Editing and deleting watches

*You can't change the value of the expression, only the expression itself. To change the value, use **Debug | Evaluate/Modify**.*

It's easy to edit, add, or delete watches. When the Watch window is active, the currently active expression is highlighted. To select a different expression, use the *Home*, *End*, *↑*, or *↓* keys.

To edit (change) the currently highlighted watch, you can choose **Debug | Watches | Edit Watch**. Even easier, as shown on the bottom line of the screen, you can press *Enter*. The debugger opens a pop-up window with the selected expression, and you can edit it. 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*. You can delete all of the watches by choosing **Debug | Watches | Remove All Watches**.

Evaluating and modifying

The Watch window is wonderful for tracing values as you step through your program. Other times, you may want to interactively check a variable or change its value without creating a watch point.

To accommodate these needs, the debugger offers the Evaluate and Modify window. To bring it up, choose the **Debug | Evaluate/Modify** command (or press *Ctrl-F4*). This window contains the **Expression**, **Result**, and **New value** boxes.

As with the Add watch box, the Evaluate and Modify box already contains the word found at the cursor; it's in highlight mode. Edit it as you would the Add Watch box and press *Enter* when you want to evaluate it. The current value of the constant, variable, or expression will then appear in the Result box.

The Evaluate box accepts exactly the same set of constants, variables, and expressions that the Watch window does. You have the same freedoms and restrictions we've already mentioned. You can also use the same format characters as you can for Watch expressions.

When you press *Enter*, the identifier or expression in the Evaluate box is highlighted again, which means if you start typing a new name (without pressing *Ins* or an arrow key), it will replace the old one. This lets you quickly type in a series of variables and expressions.

The New Value box allows you to modify the value of the variables named in the Evaluate box. You can enter a constant value, the name of another variable, or even an expression. The resulting value must be of a type compatible with the variable in the Evaluate box. Therefore, if you have an expression in Evaluate that does not result in a memory location, then any value entered in New Value will result in the message "Cannot be modified."

The Result box shows the current value of whatever is in the Evaluate box, using the same format as the Watch window. And, like the Watch window, the data will sometimes be too large to fit. In such cases, you can use the *Tab*, *Backtab*, *←*, *→*, *Home*, and *End* keys to scroll through the box.

In all cases, you can use the *↑* and *↓* keys (or regular keyboard editing commands) to move between the three boxes. Once you have modified a box, press *Enter* to evaluate the input.

Modifying expressions

The ability to modify a variable while the program is running is a tremendous help while debugging. It can also be dangerous, so you need to be sure you know the *do's* and *don'ts* of modification.

The simplest form of modification is to enter a variable name in the Evaluate box and a corresponding value in the New Value box. When you press *Enter* after typing in the new value, the variable's value is changed, and the Result box is updated to reflect that.

You are not limited to constant values, though. You can enter into the New Value box any variable or expression that you could enter into the Evaluate box, with one major qualification: It must be assignment-compatible with the variable or expression in the Evaluate box. In other words, if *expr1* represents what's currently in the Evaluate box, then you cannot legally enter the expression *expr2* into the New Value box if the statement

```
expr1 := expr2;
```

would cause a compiler or run-time error.

Note that the reverse is not necessarily true: There are cases when the statement

```
expr1 := expr2;
```

is legal, but you still cannot use *expr2* in the New Value box.

If the expression entered is an incompatible type—such as entering a floating-point value for an integer variable—then the Result box will instead display the message “Type mismatch.” To make the Result box redisplay the current value of the variable, move back up to the Evaluate box and press *Enter*.

If the expression entered yields an out-of-range value—such as entering 50,000 for a variable of type Integer—the Result box will display the message “Constant out of range.” The same thing will occur if you type in an array element with an index that’s out of range.

If the expression entered in the New Value box is one that can’t be assigned, then the Result box will get the message “Cannot evaluate this expression.” Such expressions include arrays, records, sets, and files.

Likewise, if the variable or expression in the Evaluate box is one that can’t be modified—a whole array, record, set, or a file—then attempting to assign a value to it will produce the message “Cannot be modified.”

What can you modify? Refer to Table 5.1 on page 138 for a list of what can be used in a Watch expression, along with acceptable values. Remember, though, that expressions can only use the built-in functions listed as acceptable for Watch expressions in Table 5.1.

Other things to keep in mind:

- You can’t modify entire arrays, entire records, or files; however, as mentioned, you can modify individual elements of arrays or records that resolve to one of the types listed in Table 5.1, provided they are not themselves arrays or records.
- You can’t directly modify untyped parameters passed into a procedure or function. You can, however, typecast them to a given type, then modify them according to the restrictions we’ve just detailed.
- Be aware that there can be some real dangers in modifying variables. For example, if you change a pointer, you could end

up making changes to memory that you didn't mean to, possibly even modifying other variables and data structures.

Navigation

When you are debugging a large program, especially one spread out over several units, you can actually get lost, or at least buried so deep that you can't figure out how best to get to where you want to go. To aid you with navigation, the debugger provides two mechanisms: the **Window | Call Stack** and **Search | Find Procedure** commands.

The call stack Each time a procedure or function is called, Turbo Pascal remembers the call and the parameters passed to it by pushing the information on the call stack. When you exit that procedure or function, then the call is popped off the stack, returning execution to the calling routine.

Whenever your program pauses because of a breakpoint or a single-step command, you can ask to see the current call stack by using the **Window | Call Stack** command (*Ctrl-F3*). This displays a window that shows the list of procedure/function calls currently active on the stack.

The call stack allows you to look back through the sequence of calls. When you first bring up the call stack, the topmost call is highlighted. You can use the arrow keys to move up and down through the stack. If you press *Spacebar*, you will be taken to the last active point within that program. Consider the following small program (TESTPOWER.PAS):

```
program TestPower;
function Power(Base,Exp: Word): Longint;
begin
  if Exp <= 0 then
    Power := 1
  else
    Power := Base * Power(Base,Exp-1);
end; { of func Power }

begin { main body of TestPower }
  Writeln('2^14 = ',Power(2,14));
end. { of program TestPower }
```

A CGA will display 9 calls; a Hercules, EGA, or VGA will display 12.

Compile TESTPOWER, and set a breakpoint on the second line of the function *Power* (the line *Power := 1*). Now run the program.

When it pauses, choose the **Window | Call Stack** command. You can use the \uparrow and \downarrow keys to move through the calls. The call stack tracks up to 128 nested calls.

Finding procedures and functions

Sometimes, in the middle of debugging, you want to find a particular procedure or function in order to set a breakpoint, execute to that point, check the parameter list, look at the variables, or any number of other reasons.

If your source code is spread out among multiple files, you'll love the **Search | Find Procedure** command. This command leads you to a small window, where you can enter the name of a procedure or function. After you type in an identifier and press *Enter*, Turbo Pascal checks its internal tables to find where that subprogram is located, loads in the appropriate source file (if necessary), and puts you in the Edit window with the cursor positioned at the beginning of the procedure or function.

There are three important things to remember about using the **Search | Find Procedure** command:

- **Find Procedure** does *not* affect your current debugging state. In other words, if you're paused at some point in your program, you are still paused there, and choosing **Run | Trace Into (F7)** will execute that line in your program, not the procedure or function you've just located.
- **Find Procedure** places the cursor at the first executable line of that procedure or function, rather than on the procedure or function header. This means you can choose **Run | Go to Cursor (F4)** to execute from your current position to the start of that procedure or function.
- You can only use this command if you have compiled your program and debug information is available for the procedure or function.

Since you may have routines with the same name in several different places in your program (in units, nested inside of other routines, and so on), it's a good idea to qualify the routine's name by preceding it with the name of the unit or program containing it, as well as any procedures or functions that might enclose it; for example, *module.proc.proc.<etc.>.proc*. If you modify the source code and the file position (or even name) of a procedure or function is changed, the **Search | Find Procedure** command won't know about any of those changes until you recompile. If you first compile program *TestPower* (see the section on "The call stack,"

page 142) and then delete the blank line above the declaration of function *Power*, Search | Find Procedure will put the cursor on the **if...then** instead of the **begin**.

Object-oriented debugging

You don't need to make any special preparations to debug an object-oriented program.

Working with objects in the IDE involves two functional areas: stepping and tracing through method calls, and examining object data. The integrated debugger "understands" objects and handles them automatically in a fashion consistent with related language components like procedures and records.

Stepping and tracing method calls

A method call is treated by the IDE as an ordinary procedure or function call. **F8 (Step Over)** treats a method call as an indivisible unit, and executes it without displaying the method's internal code; whereas **F7 (Trace Into)** loads the method's code if it's available, and traces through the method's statements.

There is no difference between tracing static method calls and tracing virtual method calls. Virtual method calls are resolved at run time, but because debugging happens at run time, there is no ambiguity, and the integrated debugger always knows the correct method to execute next.

The Call Stack window displays the names of methods prefixed by the object type that defines the method (for example, *Circle.Init* rather than simply *Init*).

Objects in the Evaluate window

When objects are displayed in the Evaluate and Modify window, they appear in a fashion very similar to records. All the same format specifiers apply, and all expressions that would be valid for records are valid for objects.

Only the data fields are displayed when the object name as a whole is presented to Evaluate. However, when the specific method name is evaluated, as in

```
ACircle.MoveTo
```

a pointer value is displayed indicating the address of the method's code. This is true for both static and virtual methods. The integrated debugger handles virtual method lookup transparently through the virtual method table (VMT), and the address of a virtual method for a given object instance is the true address of the correct method code for that instance.

When it is tracing inside a method, the IDE "knows" about the scope and presence of the *Self* parameter. You can evaluate or watch *Self*, and you can follow it with format specifiers and field or method qualifiers.

Expressions in the Find Procedure command

Turbo Pascal allows the entry of expressions at the prompt for the Find Procedure command of the Search menu. To be legal, an expression must evaluate to an address in the code segment. Note that this applies to procedural variables and parameters as well as to object methods.

General issues

You've learned how to use the debugger; now we'll cover some other issues that might arise while you're debugging.

Writing programs for debugging

There are some simple things you can do to make your programs easier to debug. In most cases, don't put more than a single statement on a line. Since the debugger executes on a line-by-line basis, this ensures that no more than one statement will be executed each time you press *F7*.

At the same time, recognize that there are cases when you might want to put multiple statements per line. If there is a list of statements you have to step through, but which aren't really relevant to the debugging, feel free to bunch them up into one or two statements so that you can step through them more quickly. That's why in one of the earlier examples we wrote

```
W := 10; X := 20; Y := 30; Z := 40;
```

instead of

```
W := 10;
```

```
X := 20;  
Y := 30;  
Z := 40;
```

You can also organize your variable declarations so that the ones you are most likely to put in the Watch window are nearest the initial **begin** statement of the procedure or function. When you step into that procedure or function, you can quickly move the cursor through the list, using **Add Watch (Ctrl-F7)** to add each variable as a watch.

In a similar fashion, if there are expressions that you commonly want to watch or evaluate at certain points in your program, insert them as comments. When you get to that point, you can move the cursor to the start of the expression and copy it into the Add Watch or Evaluate and Modify box. This is especially helpful if the expression is a complex one, involving typecasting, format characters, array elements, or record fields.

Finally, the best debugging is preventive debugging. A well-designed, clearly-written program will not only have fewer bugs, but it will make it easier for you to track down and fix what few bugs there are. Here are some basics to remember when you're writing your program:

- Program incrementally. When possible, code, test, and debug your program one (small) section at a time. Get each section working before moving on to the next section.
- Break your program into modules: units, procedures, functions. Avoid writing procedures or functions longer than about 25 lines; if one gets bigger than that, try breaking it up into a few smaller procedures and functions.
- When possible, pass information through parameters only, instead of referencing global variables inside procedures and functions. This avoids side effects and also makes the code easier to debug, since you can easily watch all information coming in and out of a given procedure or function.
- Concentrate on making your program work correctly *before* trying to make it fast.

Memory issues

It is possible to run out of memory while debugging a large program. After all, Turbo Pascal is holding the editor, compiler, debugger, current source code file, executable code, symbol tables,

and any other debugging information in memory—all at the same time. You can monitor the amount of free memory with the **File | Get Info** command.

Changes you make in the Startup Options dialog box (Options | Environment) are permanent and saved directly into TURBO.EXE. Changes made to other dialog settings can be saved in a TURBO.TP configuration file. Refer to Chapter 7.

Both the IDE and Turbo Pascal itself are very configurable and there are several steps you can take to make more workspace available for compiling and debugging your programs. Some solutions are easy to implement, while others involve altering your code or turning off debug information selectively. Always start with the options that are painless and safe and then, if necessary, take progressively more radical steps in order to increase the IDE's capacity. Once you find a system configuration that provides you with enough capacity, you might want to permanently modify the your AUTOEXEC.BAT, CONFIG.SYS, TURBO.TP, and TURBO.EXE files.

Outside the IDE

- Remove TSRs from memory. If you have Sidekick or Superkey loaded in memory or EMS, exit the IDE, remove them, and then reload TURBO.EXE.
- Modify CONFIG.SYS to remove unnecessary drivers (ANSI.SYS, disk caches, etc.). You can also reduce the number of files and buffers with FILES = 20, BUFFERS = 20. Make sure these changes are safe for any other software you are using.

Re-configuring Turbo Pascal

There are command-line parameters that you can pass to TURBO.EXE at startup that correspond to all the settings on the Options | Environment | Startup dialog box; refer to page 174.

1. Set **Compile | Destination** to *Disk*.
2. On the **Options | Linker** dialog box, set **Link Buffer** to *Disk*.
3. Using the **Options | Environment | Startup Options** dialog box, try one or more of the following:
 - a. If you have expanded memory on your system (EMS), make sure the **Use Expanded Memory** option is enabled and make plenty of EMS available to Turbo Pascal (by reducing the amount of EMS being used by resident programs or drivers like RAM disks, Sidekick, etc.). The IDE can use at least 400K of EMS for overlays, extra buffers, and other system resources. All these will increase the workspace for your programs. (Making more than 400K EMS available will increase the IDE's performance, although it will not make more memory available to compile and debug your programs.)
 - b. If you're not trying to debug a graphics program, make sure the **Graphics Screen Save** option is disabled. Like all

If you have EMS available, disabling this option will have no effect on IDE capacity.

startup options, you can enable this option on the command-line when you debug a graphics program.

- c. Reduce the default of the **Overlay** and **Window Heap Size** options. Every kilobyte you subtract here yields another kilobyte for your program. If you have EMS available, reducing these heap sizes somewhat won't have much negative impact on the IDE's performance.
- d. Disable the **Load TURBO.TPL** option. TURBO.TPL contains the commonly used standard units and is loaded into memory at startup to optimize linker performance. By disabling this option, you'll still be able to compile and debug programs, but you'll have to extract all the units from TURBO.TPL first (using the TPUMOVER utility; refer to UTILS.DOC on your distribution disk).

If you don't have the IDE load TURBO.TPL, you won't be able to evaluate expressions using the Evaluate/Modify dialog box unless a debugging session is active.

Make sure to leave the extracted units on disk and in your unit (Options | Directories | Unit directories) so your programs can make use of the Dos, Crt, Overlay, and Printer units.

Of course, if you're not debugging, you can greatly increase IDE capacity by disabling the Integrated switch (Options | Debugger).

As an alternative, you can leave the **Load TURBO.TPL** option enabled and still reduce the size of TURBO.TPL by about 15K. Just extract all units from TURBO.TPL with the exception of SYSTEM.TPU. Then delete all units from TURBO.TPL with the exception of SYSTEM.TPU.

- e. On a unit-by-unit basis, turn off debug information in those units that are already debugged. A common technique is to build a "test harness" around your code as you develop it. Once that code is implemented, tested and debugged, turn off symbol information in that unit by disabling the **Debug Information** switch (Options | Compiler dialog box) and recompiling. You can also imbed a **(\$D-)** in the unit itself. If you do so, it's a good idea to use conditional directives and defines to control enabling and disabling debug information in various units (refer to Chapter 21 in the *Programmer's Guide*). If you proceed as described here and end up with debug information disabled everywhere in your program—and are still having capacity problems—consider modifying your code as described next.

Modifying your source code

Some of the following measures are easy to do and yield big capacity gains. Others are more radical and you might want to use conditional directives (see Chapter 21 in the *Programmer's Guide*) to turn them on or off.

- Overlay units in your program. This is very safe, flexible, and can dramatically increase the IDE workspace. Refer to Chapter 13 in the *Programmer's Guide* for more information.
- Using the **Options | Memory Sizes** dialog box, reduce the **Stack Size** and **Low Heap Limit**. Make sure there's enough stack for your program, especially if you've turned off stack checking as recommended next.
- Using the settings in the **Compiler Options** dialog box, try one or more of the following:
 - Disable **Range Checking** and **Stack Checking**. **Stack Checking** is on by default. Turn it off once your program is stable and you've determined its stack requirements.
 - Disable **Emulation** during debugging. Of course, only enable **Emulation** and **8087/80287** code generation if you are doing IEEE floating point. If you have a numeric coprocessor on your debugging machine, disable **Emulation** while you're debugging non-floating point code.
- Reduce the number of symbols in the interface sections of units. Don't declare something in the interface section of a unit unless it's used by code outside the unit. Doing this is good, safe programming practice and will make more symbol space available during the compilation of large programs.

Turbo Debugger and the IDE

Turbo Pascal itself and the IDE both offer many ways for you to gain capacity by making adjustments to default settings. If you run out of memory compiling or debugging your programs and have tried most of the painless ideas offered here, consider using the IDE to edit and compile your programs, and then using Turbo Debugger to debug them. If you have Turbo Debugger and want to use it to debug programs developed in the IDE, configure the IDE as follows:

1. Set **Compile | Destination** to *Disk*.
2. In the **Options | Debugger** dialog box, disable **Integrated** and enable **Standalone** debugging.

You can also use the command-line compiler, TPC.EXE, or the extended memory command-line compiler, TPCX.EXE to build massive programs (several megabytes in size). Then you can use TD, TD286 or TD386 to debug them.

Recursive routines

Recursion is a programming technique where a procedure calls itself (directly or indirectly). For example, the function *Power* shown in an earlier example is recursive, because it calls itself to calculate the value it needs to return.

There are some considerations to keep in mind when debugging recursive code. First, deep levels of recursion can eat up lots of system stack space, which can have other side effects (such as your program halting or crashing due to stack overflow). This is a general danger of using recursion under any circumstance; just be aware that, if your program crashes while debugging, it may well be due to stack overflow rather than anything you did with the debugger.

Also, if you have deep levels of recursion, you may not be able to find your way out immediately with the call stack. That's because the call stack is limited to the last 128 function/procedure calls. You can, however, go to the bottom of the stack, use it to find the oldest call, pop out to that spot, then use the call stack again.

Each time a function is called recursively, it creates a new set of local variables and pass-by-value (non-**var**) parameters. If you have added these to the Watch window, be aware that these values will "float" to reflect the currently active local data.

Where debugging won't go

There are some cases where you can't trace into a given function or procedure. This is usually—but not always—because the source isn't available. These situations include the following:

- Any **inline** procedure or function; that is, any procedure or function of type **inline**. That's because these aren't procedure or function calls at all; the associated machine language is inserted in place of the "call." Such a call is treated as a single statement.

Note that you can trace into procedures and functions that happen to use **inline** statements. However, in that case, each **inline** statement is treated as a single line, no matter how many lines it occupies. This follows the same rule as other statements; that is, if a single statement takes up several lines, it is treated by Run | Trace Into and Step Over (*F7* and *F8*) as a single line.

- Any Turbo Pascal routine from one of the standard units (*Crt, Dos, Graph, Graph3, Overlay, Printer, System, Turbo3*).
- Any **external** procedure or function.
- Any **interrupt** procedure or function.
- Any procedure, function, or initialization code contained in a unit that was not compiled with the **{SD+}** directive (or with **Options | Compiler | Debug Information** turned on).
- Any procedure, function, or initialization code contained in a unit whose source code cannot be found. If it's not in the current or the unit directory, or if its source code is in a file named something other than *unitname.PAS* (where *unitname* is the name of the unit as given in the **uses** clause), the IDE will prompt you for the correct file name. If you enter a null file name, or if you press *Esc*, the debugger will move on as if debug information were not available.
- Any procedure set up as an exit procedure. If you step through your program with **Run | Trace Into (F7)**, you'll never step into an *Exit* procedure. Note, however, that you can set a breakpoint in an *Exit* procedure, and the debugger will break appropriately when the execution bar arrives at your breakpoint.

Common pitfalls

There are a few problems that you often run into while debugging. Here's a list of things to watch out for:

- Not generating the global and local debug information needed. By default, both of these switches are on. If you have problems stepping into a program or unit, put **{SD+,L+}** at the start of every program or unit you wish to debug.
- Starting to debug another program without clearing the breakpoints and Watch expressions from the previous one. Before loading in a new program to debug, you should always execute the following commands: **Run | Program Reset (Ctrl-F2) Debug | Watches | Remove All Watches**.
- Trying to compile and run another program when the previous one is still set up as the main file. Use the **Compile | Main File** command to clear out the previous name or set a new one.
- Press *N* when you get the "Source modified, rebuild? (Y/N)" prompt. This means that you've modified a source file while debugging, and the debugger's line-number tables may no longer be valid. This can throw off breakpoints, stepping, and

other debugging activities. If you just accidentally typed a character and then deleted it, you're probably safe in pressing *N*; if you've inserted or deleted lines, though, you're better off pressing *Y*, because the machine code you're debugging doesn't match the source code you're looking at.

Error handling

In addition to the integrated debugger, Turbo Pascal provides several compiler directives and language features to help you trap programming errors. This section briefly describes some of those features.

You can insert run-time error checking for yourself by disabling the generation of automatic error-checking code and writing your own error-handling routines. Let's take a look at some examples.

Input/output error checking

If you ran this program, entered the values *45* and *8x* when prompted, and then pressed *Enter*, what would happen?

```
program DoSum;
var
  A,B,Sum: Integer;
begin
  Write('Enter two numbers: ');
  Readln(A,B);
  Sum := A + B;
  Writeln('The sum is ',Sum);
  Readln;
end.
```

You'd get a run-time error (106, in fact) and the cursor would be positioned at the statement

```
  Readln(A,B);
```

What happened? The program expected an integer value and you entered non-numeric data—*8x*—which generated a run-time error.

In a short program like this, such an error isn't a big bother. But what if you were entering a long list of numbers and had gotten through most of the list before making this mistake? You'd be

forced to start all over again. Worse yet, what if you wrote the program for someone else to use, and *they* slipped up?

Turbo Pascal allows you to disable automatic I/O error checking and test such errors for yourself within the program. To turn off I/O error checking at some point in your program, include the compiler directive `{!-}` in your program (or the **Options | Compiler | I/O Checking** option). This instructs the compiler to prohibit the production of code that checks for I/O errors.

Range checking

Another common class of run-time errors involves out-of-range or out-of-bounds values. Some examples of how these can occur include assigning too large a value to an integer variable or trying to index an array beyond its bounds. If you want it to, Turbo Pascal will generate code to check for range errors. It makes your program slightly larger and slower, but it can be invaluable in tracking down any range errors in your program.

Let's revisit an earlier example:

```
program RangeTest;
var
  List: array[1..10] of Integer;
  Indx: Integer;

begin
  for Indx := 1 to 10 do
    List[Indx] := Indx;
  Indx := 0;
  while (Indx < 11) do
  begin
    Indx := Indx + 1;
    if List[Indx] > 0 then
      List[Indx] := -List[Indx];
    end;
  for Indx := 1 to 10 do
    Writeln(List[Indx]);
  end.
```

We discovered earlier that if you compile and run this program, it will get stuck in an infinite loop. This is caused by the **while** loop executing 11 times, not 10, and the variable *Indx* having a value of 11 the last time through the loop.

The Range Checking option is in the Options | Compile dialog box. Range checking is off by default; turning range checking on makes your program slightly larger and slower, but is strongly advised until your program is thoroughly debugged.

How do you check for things like this? You can insert **{\$R+}** at the start of the program to turn range checking on. Now, when you run it, the program will halt with run-time error 201 (out-of-range error, because the array index is out of bounds) as soon as you hit the statement **if List[Indx] > 0** with *Indx* = 11. If you were running in the IDE, it would automatically take you to that statement and display the error.

There are some situations—usually in advanced programming—in which you may need to violate range bounds, most notably when working with dynamically allocated arrays or when using *Succ* and *Pred* with enumerated data types.

You can selectively implement range checking by placing the **{\$R-}** directive at the start of your program. For each section of code that needs range checking, place the **{\$R+}** directive at the start of it, and the **{\$R-}** directive at the end. For example, you could have written the preceding loop like this:

```
while Indx < 11 do
begin
  Indx := Indx + 1;
  {$R+}                                     { Enable range checking }
  if List[Indx] > 0 then
    List[Indx] := -List[Indx];
  {$R-}                                     { Disable range checking }
end;
```

Range checking will be performed only in the **if..then** statement and nowhere else, unless, of course, you have other **{\$R+}** directives elsewhere.

Other error-handling abilities

Turbo Pascal gives you the ability to perform other error-handling techniques, but because those techniques are described more fully in other parts of this manual, we'll only touch on them briefly in this section.

When your program terminates, either normally or through a run-time error, a standard exit procedure is called that's linked in with your program. Turbo Pascal lets you add in your own exit procedures, which are called *before* the standard exit procedure. In fact, each unit can have its own exit procedure, so that you can have automatic cleanup code, as well as the usual automatic

initialization code. Exit procedures are described in more detail in Chapter 18 of the *Programmer's Guide*, "Control issues."

If you try to allocate memory (through a call to *New* or *GetMem*) and there isn't sufficient memory on the heap, a heap error procedure is automatically called, which simply causes your program to exit with a run-time error. You can, however, install your own heap error procedure to handle things as you wish, like deallocating dynamic structures no longer needed or simply causing *New* or *GetMem* to return a *nil* pointer. Heap error procedures are described in more detail in Chapter 16 of the *Programmer's Guide*, "Memory issues."

If you're using the *Graph* unit, you can perform error checking much as you do for I/O error checking. One function in the unit, *GraphError*, returns an error result set by many of the graphics routines. Chapter 12 of the *Programmer's Guide*, "The *Graph* unit and the BGI," provides you with details on how to use this and the error codes that are generated.

The *Overlay* unit contains an integer variable, *OvrResult*, that stores the result code from the last operation performed by the overlay manager. Similarly, the *Dos* unit stores its result codes in the variable *DosError*.

Project management

So far, you've learned how to write Turbo Pascal programs, how to use the predefined units, and how to write your own units. At this point, your program could become large, perhaps separated into multiple source files. How do you manage such a program?

This chapter suggests how to organize your program into units, how to take advantage of the built-in **Make** and **Build** options, how to use the stand-alone **Make** utility, how to use conditional compilation within a source code file, and how to optimize your code for speed.

Program organization

Turbo Pascal 6.0 allows you to divide your program into code segments. Your main program is a single code segment, which means that after compilation, it can have no more than 64K of machine code. However, you can exceed this limit by breaking your program up into units. Each unit can also contain up to 64K of machine code when compiled. The question is: How should you organize your program into units?

The first step is to collect all your global definitions—constants, data types, and variables—into a single unit; let's call it *MyGlobals*. This is necessary if your other units reference those definitions. Unlike include files, units can't "see" any definitions made in your main program; they can only see what's in the interface

section of their own unit and other units they use. Your units can use *MyGlobals* and thus reference all your global declarations.

A second possible unit is *MyUtils*. In this unit you could collect all the utility routines used by the rest of your program. These would have to be routines that don't depend on any others (except possibly other routines in *MyUtils*).

Beyond that, you should collect procedures and functions into logical groups. In each group, you'll often find a few procedures and functions that are called by the rest of the program, and then several (or many) procedures/functions that are called by those few. A group like that makes a wonderful unit. Here's how to convert it:

1. Copy all those procedures and functions into a separate file and delete them from your main program.
2. Open that file for editing.
3. Type the following lines in front of those procedures and functions:

```
unit unitname;  
interface  
uses MyGlobals;  
implementation
```

where *unitname* is the name of your unit (and also the name of the file you're editing).

4. Type **end.** at the very end of the file.
5. In the space between **interface** and **implementation**, copy the procedure and function headers of those routines called by the rest of the program. Those headers are simply the first line of each routine, the one that starts with **procedure** (or **function**).
6. If this unit needs to use any others, type their names (separated by commas) between *MyGlobals* and the semicolon in the **uses** statement.
7. Compile the unit you've created.
8. Go back to your main program and add the unit's name to the **uses** statement at the start of the program.

Ideally, you want your program organized so that when you are working on a particular aspect of it, you are modifying and recompiling a single module (unit or main program). This minimizes compile time; more importantly, it lets you work with smaller, more manageable chunks of code.

Initialization

Remember in all this that each unit can (optionally) have its own initialization code. This code is automatically executed when the program is first loaded. If your program uses several units, the initialization code for each unit is executed. The order of execution follows the order in which the units are listed in your program's **uses** statement; so if your program has the statement

```
uses MyGlobals, MyUtils, EditLib, GraphLib;
```

then the initialization section (if any) of *MyGlobals* will be called first, followed by that of *MyUtils*, then *EditLib*, then *GraphLib*.

To create an initialization section for a unit, put the keyword **begin** above the **end** that ends the implementation section. This defines the initialization section of your unit, much as the **begin..end** pair defines the main body of a program, a procedure, or a function. You can then put any Pascal code you want in here. It can reference everything declared in that unit, in both the public (interface) and private (implementation) sections; it can also reference anything from the interface portions of any units that this unit uses.

The Build and Make options

Turbo Pascal has an important feature to aid you in project management: a built-in Make utility. To understand its significance, let's look at the previous example again.

Suppose you have a program, *MYAPP.PAS*, which uses four units: *MyGlobals*, *MyUtils*, *EditLib*, and *GraphLib*. Those four units are contained in the text files *MYGLOBAL.PAS*, *MYUTILS.PAS*, *EDITLIB.PAS*, and *GRAPHLIB.PAS*, respectively. Furthermore, *MyUtils* uses *MyGlobals*, and *EditLib* and *GraphLib* use both *MyGlobals* and *MyUtils*.

When you compile *MYAPP.PAS*, it looks for the files *MYGLOBAL.TPU*, *MYUTILS.TPU*, *EDITLIB.TPU*, and *GRAPHLIB.TPU*, loads them into memory, links them with the code produced by compiling *MYAPP.PAS*, and writes everything out to *MYAPP.EXE* (if you're compiling to disk). So far, so good.

Suppose now you make modifications to EDITLIB.PAS. In order to recreate MYAPP.EXE, you need to recompile both EDITLIB.PAS and MYAPP.PAS. A little tedious, but no problem.

Now, suppose you modify the interface section of MYGLOBAL.PAS. To update MYAPP.EXE, you have to recompile all four units, as well as MYAPP.PAS. That means five separate compilations each time you make a change to MYGLOBAL.PAS—which could be enough to discourage you from using units at all. But wait...

The Make option

Turbo Pascal offers a solution. You can get the **Make** option (in the **Compile** menu) and Turbo Pascal to do all the work for you. The process is simple: After making any changes to any units or the main program, just **Make** the main program.

Turbo Pascal makes three kinds of checks.

1. *First, it checks and compares the date and time of the .TPU file for each unit used by the main program against the unit's corresponding .PAS file.* If the .PAS file has been modified since the .TPU file was created, Turbo Pascal recompiles the .PAS file, creating an updated .TPU file. So, in the first example, if you modified EDITLIB.PAS and then recompiled MYAPP.PAS (using the **Make** option), Turbo Pascal would automatically recompile EDITLIB.PAS before compiling MYAPP.PAS.
2. *The second check is to see if you changed the interface portion of the modified unit.* If you did, then Turbo Pascal recompiles all other units using that unit.

As in the second example, if you modified the interface portion of MYGLOBAL.PAS and then recompiled MYAPP.PAS, Turbo Pascal would automatically recompile MYGLOBAL.PAS, MYUTILS.PAS, EDITLIB.PAS, and GRAPHLIB.PAS (in that order) before compiling MYAPP.PAS. However, if you only modified the implementation portion, then the other dependent units don't need to be recompiled, since (as far as they're concerned) you didn't change that unit.

3. *The third check is to see if you changed any Include or .OBJ files (containing assembly language routines) used by any units.* If a given .TPU file is older than any of the Include or .OBJ files it links in, then that unit is recompiled. That way, if you modify and assemble some routines used by a unit, that unit is

automatically recompiled the next time you compile a program using that unit.

The Make option has no effect on units found in TURBO.TPL.

To use the **Make** option under the IDE, either select the **Make** command from the **Compile** menu, or press **F9**. To invoke it with the command-line compiler, use the option **/M**.

The Build option

The **Build** option (also in the **Compile** menu) is a special case of the **Make** option. When you compile a program using **Build**, it automatically recompiles *all* units used by that program (except, of course, those units in TURBO.TPL). This always brings everything up to date. You can invoke **Build** from the command line with the **/B** option.

The Stand-alone MAKE utility

Turbo Pascal places a great deal of power and flexibility at your fingertips. You can use it to manage large, complex programs that are built from numerous unit, source, and object files. And it can automatically perform a **Build** or a **Make** operation, recompiling units as needed. Understandably, though, Turbo Pascal has no mechanism for recreating .OBJ files from assembly code routines (.ASM files) that have changed. To do that, you need to use a separate assembler. The question then becomes, how do you keep your .ASM and .OBJ files updated?

The answer is simple: Use the **MAKE** utility that's included with Turbo Pascal. **MAKE** is an intelligent program manager that—given the proper instructions—does all the work necessary to keep your program up to date. In fact, **MAKE** can do far more than that. It can make backups, pull files out of different subdirectories, and even automatically run your programs should the data files that they use be modified. As you use **MAKE** more and more, you'll see new and different ways it can help you to manage your program development.

MAKE is documented in an online text file, UTILS.DOC.

MAKE is a stand-alone utility; it is different from the **Make** and **Build** options that are part of both the IDE and the command-line compiler. Here's an example of how you might use it.

A quick example

Suppose you're writing some programs to help you display information about nearby star systems. You have one program—GETSTARS.PAS—that reads in a text file listing star systems, does some processing on it, then produces a binary data file with the resulting information in it.

GETSTARS.PAS uses three units: STARDEFS.TPU, which contains the global definitions; STARLIB.TPU, which has certain utility routines; and STARPROC.TPU, which does the bulk of the processing. The source code for these units is found in STARDEFS.PAS, STARLIB.PAS, and STARPROC.PAS, respectively.

The next issue is dependencies. STARDEFS.PAS doesn't use any other units; STARLIB.PAS uses STARDEFS; STARPROC.PAS uses STARDEFS and STARLIB; and GETSTARS.PAS uses STARDEFS, STARLIB, and STARPROC.

Given that, to produce GETSTARS.EXE you would simply "make" GETSTARS.PAS. Turbo Pascal would recompile the units as needed.

Suppose now that you convert a number of the routines in STARLIB.PAS into assembly language, creating the files SLIB1.ASM and SLIB2.ASM, then use Turbo Assembler to create SLIB1.OBJ and SLIB2.OBJ. Each time STARLIB.PAS is compiled, it links in those .OBJ files. And, in fact, Turbo Pascal is smart enough to recompile STARLIB.PAS if STARLIB.TPU is older than either of those .OBJ files.

However, what if either .OBJ file is older than the .ASM file upon which it depends? That means that the particular .ASM file needs to be re-assembled. Turbo Pascal can't assemble those files for you, so what do you do?

You create a *make file* and let MAKE do the work for you. A make file consists of *dependencies* and *commands*. The dependencies tell MAKE which files a given file depends upon; the commands tell MAKE how to create that given file from the other ones.

Creating a makefile Your makefile for this project might look like this:

```
getstars.exe: getstars.pas stardefs.pas starlib.pas slib1.asm \  
              slib2.asm slib1.obj slib2.obj  
  
tpc getstars /m  
  
slib1.obj: slib1.asm  
          TASM slib1.asm slib1.obj  
  
slib2.obj: slib2.asm  
          TASM slib2.asm slib2.obj
```

Okay, so this looks a bit cryptic. Here's an explanation:

- The first two lines tell MAKE that GETSTARS.EXE depends on three Pascal, two assembly language, and two .OBJ files (the backslash at the end of line 1 tells MAKE to ignore the line break and continue the dependency definition on the next line).
- The third line tells MAKE how to build a new GETSTARS.EXE. Notice that it simply invokes the command-line compiler on GETSTARS.PAS and uses the built-in Turbo Pascal Make facility (/M option).
- The next two lines (ignoring the blank line) tell MAKE that SLIB1.OBJ depends on SLIB1.ASM and show MAKE how to build a new SLIB1.OBJ.
- Similarly, the last two lines define the dependencies (only one file, actually) and MAKE procedures for the file SLIB2.OBJ.

Using MAKE Let's suppose you've created this Make file using the editor in the Turbo Pascal IDE (or any other ASCII editor) and saved it as the file STARS.MAK. You would then use it by issuing the command

```
make -fstars.mak
```

MAKE works from the bottom of the file to the top.

where *-f* is an option telling MAKE which file to use. First, it checks to see if SLIB2.OBJ is older than SLIB2.ASM. If it is, then MAKE issues the command

```
TASM SLIB2.asm SLIB2.obj
```

which assembles SLIB2.ASM, creating a new version of SLIB2.OBJ. It then makes the same check on SLIB1.ASM and issues the same command if needed. Finally, it checks all of the dependencies for GETSTARS.EXE and, if necessary, issues the command

```
tpc getstars /m
```

The **/M** option tells Turbo Pascal to use its own internal MAKE routines, which will then resolve all unit dependencies, including recompiling STARLIB.PAS if either SLIB1.OBJ or SLIB2.OBJ is newer than STARLIB.TPU.

Conditional compilation

To make your job easier, Turbo Pascal 6.0 offers conditional compilation. This means that you can now decide what portions of your program to compile based on options or defined symbols. For a complete reference to conditional directives, refer to Chapter 21, "Compiler directives," in the *Programmer's Guide*.

The conditional directives are similar in format to the compiler directives you're accustomed to; in other words, they take the format

```
{directive arg}
```

where *directive* is the directive (such as **DEFINE**, **IFDEF**, and so on), and *arg* is the argument, if any. Note that there *must* be a separator (blank, tab) between *directive* and *arg*. Table 6.1 lists all the conditional directives, with their meanings.

Table 6.1
Summary of compiler
directives

{DEFINE <i>symbol</i>	Defines <i>symbol</i> for other directives
{UNDEF <i>symbol</i>	Removes definition of <i>symbol</i>
{IFDEF <i>symbol</i>	Compiles following code if <i>symbol</i> is defined
{IFNDEF <i>symbol</i>	Compiles following code if <i>symbol</i> is not defined
{IFOPT <i>x+</i>	Compiles following code if directive <i>x</i> is enabled
{IFOPT <i>x-</i>	Compiles following code if directive <i>x</i> is disabled
{ELSE}	Compiles following code if previous IFxxx is not True
{ENDIF}	Marks end of IFxxx or ELSE section

The DEFINE and UNDEF directives

The **IFDEF** and **IFNDEF** directives test to see if a given symbol is defined. These symbols are defined using the **DEFINE** directive and undefined **UNDEF** directives. (You can also define symbols on the command line and in the IDE.)

To define a symbol, insert the directive

```
{DEFINE symbol}
```

into your program. *symbol* follows the usual rules for identifiers as far as length, characters allowed, and other specifications. For example, you might write

```
{DEFINE debug}
```

This defines the symbol *debug* for the remainder of module being compiled, or until the statement

```
{UNDEF debug}
```

is encountered. As you might guess, **UNDEF** “undefines” a symbol. If the symbol isn’t defined, **UNDEF** has no effect.

Defining at the command line

If you’re using the command-line version of Turbo Pascal (TPC.EXE), you can define conditional symbols on the command line itself. TPC accepts a **/D** option, followed by a list of symbols separated by semicolons:

```
tpc myprog /Ddebug;test;dump
```

This would define the symbols *debug*, *test*, and *dump* for the program MYPROG.PAS. Note that the **/D** option is cumulative, so that the following command line is equivalent to the previous one:

```
tpc myprog /Ddebug /Dtest /Ddump
```

Defining in the IDE

Conditional symbols can be defined in the **Conditional Defines** input box (**Options | Compiler**). Multiple symbols can be defined by entering them in the input box, separated by semicolons. The syntax is the same as that of the command-line version.

Predefined symbols

In addition to any symbols you define, you also can test certain symbols that Turbo Pascal has defined. Table 6.2 lists these symbols; let’s look at each in a little more detail.

Table 6.2
Predefined conditional symbols

<i>VER60</i>	Always defined (TP 4.0 has <i>VER40</i> defined, etc.)
<i>MSDOS</i>	Always defined
<i>CPU86</i>	Always defined
<i>CPU87</i>	Defined if an 8087 is present at compile time

The `VER60` symbol The symbol `VER60` is always defined for Turbo Pascal 6.0. In a similar fashion, `VER40` is defined for version 4.0 of Turbo Pascal, `VER50` for version 5.0 and so on. Future versions will have corresponding predefined symbols; for example, version 6.5 would have `VER65` defined, version 7.0 would have `VER70` defined, and so on. This allows you to create source code files that can use future enhancements while maintaining compatibility with older versions.

The `MSDOS` and `CPU86` symbols These symbols are always defined (at least for Turbo Pascal 6.0 running under DOS). The `MSDOS` symbol indicates you are compiling under the DOS operating system. The `CPU86` symbol means you are compiling on a computer using an Intel iAPx86 (8088, 8086, 80186, 80286, 80386, 80486) processor.

As future versions of Turbo Pascal for other operating systems and processors become available, they will have similar symbols indicating which operating system and/or processor is being used. Using these symbols, you can create a single source code file for more than one operating system or hardware configuration.

The `CPU87` symbol Turbo Pascal 6.0 supports floating-point operations in two ways: hardware and software. If you have an 80x87 math coprocessor installed in your computer system, you can use the IEEE floating-point types (Single, Double, Extended, comp), and Turbo Pascal will produce direct calls to the math chip. If you don't have an 8087, you can still use the IEEE types by instructing Turbo Pascal to emulate the 8087 in software. Otherwise, you can just use the standard floating-point type `real` (6 bytes in size), and Turbo Pascal will support all your operations with software routines. Use the `$N` and `$E` directives to indicate which you wish to use.

When you load the Turbo Pascal compiler, it checks to see if an 80x87 chip is installed. If it is, then the `CPU87` symbol is defined; otherwise, it's undefined. You might then have the following code at the start of your program:

```

{$N+}                                { Always use IEEE floating point }
{$IFDEF CPU87}                        { If there's no 80x87 present }
{$E+}                                  { No hardware: Use emulation library }
{$ENDIF}

```

The IFxxx, ELSE, and ENDIF symbols

The idea behind conditional directives is that you want to select some amount of source code to be compiled if a particular symbol is (or is not) defined or if a particular option is (or is not) enabled. The general format follows:

```
{$IFxxx}  
    source code  
{$ENDIF}
```

where **IFxxx** is **IFDEF**, **IFNDEF**, or **IFOPT**, followed by the appropriate argument, and *source code* is any amount of Turbo Pascal source code. If the expression in the **IFxxx** directive is True, then *source code* is compiled; otherwise, it is ignored as if it had been commented out of your program.

Often you have alternate chunks of source code. If the expression is True, you want one chunk compiled, and if it's False, you want the other one compiled. Turbo Pascal lets you do this with the **\$ELSE** directive:

```
{$IFxxx}  
    source code A  
{$ELSE}  
    source code B  
{$ENDIF}
```

If the expression in *IFxxx* is True, *source code A* is compiled; otherwise *source code B* is compiled.

Note that all *IFxxx* directives must be completed within the same source file, which means they cannot start in one source file and end in another. However, an *IFxxx* directive can encompass an Include file:

```
{$IFxxx}  
{$I file1.pas}  
{$ELSE}  
{$I file2.pas}  
{$ENDIF}
```

That way, you can select alternate Include files based on some condition.

You can nest *IFxxx..ENDIF* constructs so that you can have something like this:

```

{$IFxxx}                                { First IF directive }
...
{$IFxxx}                                { Second IF directive }
...
{$ENDIF}                                { Terminates second IF directive }
...
{$ENDIF}                                { Terminates first IF directive }

```

The IFDEF and IFNDEF directives

You've learned how to define a symbol, and also that there are some predefined symbols. The **IFDEF** and **IFNDEF** directives let you conditionally compile code based on whether those symbols are defined or undefined. You saw this example earlier:

```

{$IFDEF CPU87}                          { If there's an 80x87 present }
{$N+,E-}                                 { Then use the inline 8087 code }
{$ELSE}
{$N+,E+}                                 { Else use the emulation library }
{$ENDIF}

```

By putting this in your program, you can automatically select the **\$N** option if an 8087 math coprocessor is present when your program is compiled. That's an important point: This is a compile-time option. If there is an 8087 coprocessor in your machine when you compile, then your program will be compiled with the **\$N+** and **E-** compiler directives, selecting direct calls to the 8087. Otherwise, it will be compiled with the **\$N+** and **\$E+** directives, using the software 8087 emulation. If you compile this program on a machine with an 8087, you can't run the resulting .EXE file on a machine without an 8087. (Of course, a program compiled using **{\$N+,E+}** will run on any system and use emulation only if no 8087 hardware is detected.)

It is also common to use the **IFDEF** and **IFNDEF** directives to insert debugging information into your compiled code. For example, if you put the following code at the start of each unit:

```

{$IFDEF debug}
{$D+,L+}
{$ELSE}
{$D-,L-}
{$ENDIF}

```

and the following directive at the start of your program:

```

{$DEFINE debug}

```

and compile your program, then complete debugging information will be generated by the compiler for use with the integrated debugger or the standalone Turbo Debugger. In a similar fashion, you can have sections of code that you want compiled only if you are debugging; in that case, you would write

```
{IFDEF debug}
    source code
{ENDIF}
```

where *source code* will be compiled only if *debug* is defined at that point.

The IFOPT directive

You may want to include or exclude code, depending upon which compiler options (range-checking, I/O-checking, numeric-processing, and so on) have been selected. Turbo Pascal lets you do that with the **IFOPT** directive, which takes two forms:

```
{IFOPT x+}
```

and

```
{IFOPT x-}
```

where *x* is one of the compiler options: **\$A, \$B, \$D, \$E, \$F, \$G, \$I, \$L, \$N, \$O, \$R, \$S, \$V, \$X** (see Chapter 21 in the *Programmer's Guide*, "Compiler directives," for a complete description). With the first form, the following code is compiled if the compiler option is currently enabled; with the second, the code is compiled if the option is currently disabled. So, as an example, you could have the following:

```
var
    {IFOPT N+}
    Radius,Circ,Area: Double;
    {$ELSE}
    Radius,Circ,Area: Real;
    {ENDIF}
```

This selects the data type for the listed variables based on whether or not 8087 support is enabled.

An alternate example might be

```
Assign(F,Filename);
Reset(F);
{$IFOPT I-}
IOCheck;
{$ENDIF}
```

where *IOCheck* is a user-written procedure that gets the value of *IOResult*, and prints out an error message as needed. There's no sense calling *IOCheck* if you've selected the **{I+}** option since, if there's an error, your program will halt before it ever calls *IOCheck*.

Optimizing code

A number of compiler options influence both the size and the speed of the code. This is because they insert error-checking and error-handling code into your program. It's best to enable them while you are developing your program, but you may want to disable them for your final version. Here are those options, with their settings for code optimization (the default settings are stated last):

- **{A+}** enables word alignment of variables and type constants; this results in faster memory access on 80x86 systems. The default is **{A+}**.
- **{B-}** uses short-circuit Boolean evaluation. This produces code that can run faster, depending upon how you set up your Boolean expressions. The default is **{B-}**.
- **{E-}** disables linking with a run-time library that emulates an 8087 numeric coprocessor if one isn't present. This forces Turbo Pascal to use either 8087 hardware or the standard 6-byte type real, depending on the state of the **\$N** numeric processing switch. The default is **{E-}**.
- **{G+}** uses additional instructions of the 80286 to improve code generation; programs compiled this way cannot run on 8088 and 8086 processors.
- **{I-}** turns off I/O error-checking. By calling the predefined function *IOResult*, you can handle I/O errors yourself. The default is **{I+}**.
- **{N-}** generates code capable of performing all floating-point operations using the built-in 6-byte type real. When the **\$N** switch is on, Turbo Pascal will use 8087 hardware or emulation

in software instead. If you compile a program and all the units it uses with **{\$N-}**, an 8087 run-time library is not required and Turbo Pascal ignores the emulation switch directive **\$E**. The default is **{\$N-}**.

- **{\$R-}** turns off range checking. This prevents code generation to check for array subscripting errors and assignment of out-of-range values. The default is **{\$R-}**.
- **{\$S-}** turns off stack-checking. This prevents code generation to ensure that there is enough space on the stack for each procedure or function call. The default is **{\$S+}**.
- **{\$V-}** turns off checking of **var** parameters that are strings. This lets you pass actual parameter strings that are of a different length than the type defined for the formal **var** parameter. The default is **{\$V+}**.
- **{\$X+}** enables functions calls to be used as statements; the result of a function call can be discarded.

See Chapter 21 of the *Programmer's Guide* for more information on compiler directives.

Optimizing your code using these options has two advantages. First, it usually makes your code smaller and faster. Second, it allows you to get away with something that you couldn't normally. However, they all have corresponding risks as well, so use them carefully, and reenable them if your program starts behaving strangely.

Note that besides embedding the compiler options in your source code directly, you can also set them using the **Options | Compiler** menu in the IDE or the **/\$X** option in the command-line compiler (where X represents a letter for a compiler directive).

The IDE reference

Turbo Pascal makes it easy and efficient for you to write, edit, compile, link, and debug your programs. That's what Borland's programmer's platform (also known as the integrated environment, or IDE for short) is all about.

The Turbo Pascal IDE furnishes these extras to make program writing even smoother:

- multiple, movable, resizable windows
- mouse support
- multi-file editing of files up to 1 Mb in size
- dialog boxes
- cut-and-paste commands (with copying allowed from the Help window and between Edit windows)
- search-and-replace capabilities
- print capabilities
- editor macro language

All of the windows, dialog boxes, etc., pictured in this chapter depict what you'd see on a monochrome monitor.

This chapter tells you briefly how to start and exit Turbo Pascal and then launches into detail about the individual menu items, dialog boxes, buttons, and so on. For an introduction to the basic components of the IDE, you can

- Go to Chapter 1. This chapter provides you with some general information about the IDE and then gets you started programming in the environment.

See page 214 for more about the online help system.

- Run TPTOUR. This interactive tutorial emulates the Turbo Pascal IDE to show you how to open files, edit them, and compile, run and debug programs, plus learn general window-management skills.
- Take advantage of Turbo Pascal's extensive online help system. You can get information about any aspect of the IDE in a keystroke (*F1*); specific language help is at your fingertips too (press *Ctrl-F1* while you're in the Edit window).

Starting and exiting

Starting Turbo Pascal is simple. You just move to your Turbo Pascal directory and type `TURBO` at the DOS command line. If you like, you can use one or more options (and file names) along with the `TURBO` command. These options make use of dual monitors, expanded memory, RAM disks, LCD screens, the EGA palette, and more.

Command-line options

The command-line options for Turbo Pascal's IDE are `/C`, `/D`, `/E`, `/G`, `/L`, `/N`, `/P`, `/SX`, `/T`, `/X`. These options use this syntax:

```
turbo [options] files
```

Placing a `+` (or a space) after the directive turns it on; placing a `-` after it turns it off. For example,

```
turbo -g -p- myfile
```

enables graphics memory save and disables palette swapping.

- The `/C` option If you use the `/C` option followed by a configuration file name, Turbo Pascal will load in that configuration file when it starts up.
- The `/D` option The `/D` option causes Turbo Pascal to work in dual monitor mode if it detects appropriate hardware (for example, a monochrome card and a color card); otherwise, the `/D` option is ignored. Use dual monitor mode when you run or debug a program, or shell to DOS (**File | DOS Shell**).

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 Pascal screen will appear on the inactive monitor, and program output will go to the active monitor. So when you type `TURBO /D` at the DOS prompt on one monitor, Turbo Pascal will come up on the other monitor. When you want to test your program on a particular monitor, exit Turbo Pascal, switch the active monitor to the one you want to test with, and then issue the `TURBO /D` command again. Program output will then go to the monitor where you typed the `TURBO` 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 can cause unpredictable results.
- When you run or debug programs that explicitly make use of dual monitors, do not use the Turbo Pascal dual monitor option (`/D`).

- The `/E` option Use the `/E` option to change the size of the editor heap. The default size is 28K, which is the minimum setting; the maximum is 128K. Making the editor heap larger than 28K will only improve IDE performance if you're using a slow disk as a swap device. If you have EMS or have placed your swap file on a RAM disk (see `/S` option), then don't change the default setting.
- The `/G` option Use the `/G` option to enable a full graphics memory save while you're debugging graphics programs on EGA, VGA, and MCGA systems. With **G**raphics Screen Save on, the IDE will reserve another 8K for the buffer (which will be placed in EMS if available).
- The `/L` option Use the `/L` option if you're running Turbo Pascal on an LCD screen.
- The `/N` option Use the `/N` option to enable or disable CGA snow checking; the default setting is on. Disable this option if you're using a CGA that doesn't experience snow during screen updates. This option has no effect unless you're using a CGA.

- The **/O** option Use the **/O** option to change the IDE's overlay heap size. The default is 112K. The minimum size you can adjust this to is 64K; the maximum is 256K. If you have EMS, you can decrease the size of the overlay heap without degrading IDE performance and therefore free more memory for compiling and debugging your programs.
- The **/P** option Use the **/P** option, which controls palette swapping on EGA video adapters, when your program modifies the EGA palette registers. The EGA palette will be restored each time the screen is swapped.
- In general, you don't need to use this option unless your program modifies the EGA palette registers or unless your program uses BGI to change the palette.
- The **/S** option If your system has no EMS available, use the **/S** option to specify the drive and path of a "fast" swap area, such as a RAM disk (for example, **/Sd:** \, where *d* is the drive). If no swap directory is specified, a swap file is created in the current directory.
- The **/T** option Disable the **/T** option if you don't want the IDE to load TURBO.TPL at startup. If TURBO.TPL is not loaded, you'll need the *System* unit (SYSTEM.TPU) available in order to compile or debug programs. You can increase the IDE's capacity by disabling the **/T** option and extracting SYSTEM.TPU from TURBO.TPL (using the TPUMOVER, see UTILS.DOC on your distribution disks for details).
- The **/W** option Use the **/W** option if you want to change the window heap size. The default setting is 32K. The minimum setting is 24K; the maximum is 64K. Reduce the window heap size to make more memory available for your programs if you don't need a lot of windows open on your desktop. The default provides for good capacity and ample window space.
- The **/X** option Disable the **/X** option if you don't want the IDE to use EMS. The default setting is on. When this option is enabled, the IDE improves performance by placing overlaid code, editor data, and other system resources in EMS.

Exiting Turbo Pascal

There are two ways to leave Turbo Pascal:

- To exit Turbo Pascal “permanently,” choose **File | Exit** (or press *Alt-X*). If you’ve made changes that you haven’t saved, Turbo Pascal prompts you whether you want to save your programs before exiting.
- To leave Turbo Pascal to enter commands at the DOS command line, choose **File | DOS Shell**. Turbo Pascal stays in memory but transfers you to DOS. You can enter any normal DOS commands, and even run other programs. When you’re ready to return to Turbo Pascal, type `EXIT` at the command line and press *Enter*. Turbo Pascal reappears just as you left it.

≡ (System) menu



The ≡ menu provides three general system-wide commands (**A**bout, **R**efresh Display, and **C**lear Desktop).

About

About displays a dialog box with copyright and version information for Turbo Pascal. Press *Esc* or *Spacebar* or click **OK** (or press *Enter*) to close the box.

Refresh Display

You can use this option to restore the IDE screen. This is handy if your program has accidentally overwritten the IDE’s screen and you need to restore it.

Clear Desktop

Choose ≡ | **C**lear Desktop to close all windows and clear all history lists.

File menu

Alt F

The **File** menu lets you open and create program files in Edit windows. The menu also lets you save your changes, perform other file functions, shell to DOS, and quit.

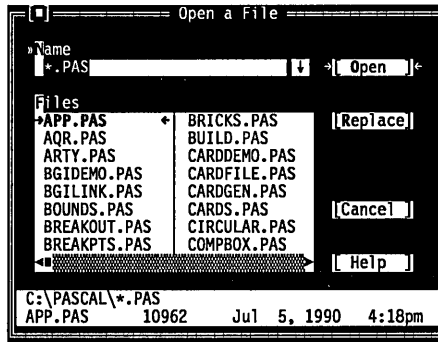
Open

F3

The **File | Open** command displays a file-selection dialog box for you to select a program file to open in an Edit window:

Figure 7.1

The Open a File dialog box



The dialog box contains an input box, a file list, buttons labeled Open, Replace, Cancel, and Help, and an information panel that describes the selected file. Now you can do any of these actions:

- Type in a full file name and choose Replace or Open. Open loads the file into a new Edit window. Replace replaces the contents of the window with the selected file; an Edit window must be active if you do this.
- Type in a file name with wildcards, which filters the file list to match your specifications.
- Press ↓ to choose a file specification from a history list of file specifications you've entered earlier.
- View the contents of different directories by selecting a directory name in the file list.

The input box lets you enter a file name explicitly or lets you enter a file name with standard DOS wildcards (* and ?) to filter the names appearing in the history list box. If you enter the entire name and press *Enter*, Turbo Pascal opens it. (If you enter a file name that Turbo Pascal can't find, it automatically creates and opens a new file with that name.)

If you press ↓ when the cursor is blinking in the input box, a scrollable history list drops down below the box. Choose a name from the list by double-clicking it or selecting it with the arrow keys and pressing *Enter*.

If you choose Replace instead of Open, the selected file replaces the file in the active Edit window instead of opening up a new window.

Once you've typed in or selected the file you want, choose the Open button (choose Cancel if you change your mind). You can also just press *Enter* once the file is selected, or you can double-click the file name.

Using the File list box

You can also type a lowercase letter to search for a file name and an uppercase letter to search for a directory name.

The File list box displays all file names in the current directory that match the specifications in the input box, displays the parent directory, and displays all subdirectories. Click the list box or press *Tab* until the list box name is highlighted. You can now press ↓ or ↑ to select a file name, and then press *Enter* to open it. You can also double-click any file name in the box to open it. You might have to scroll the box to see all the names. If you have more than one pane of names, you can also use → and ← .

The file information panel at the bottom of the Open a File dialog box displays path name, file name, date, time, and size of the file you've selected in the list box. (None of the items on this panel are selectable.) As you scroll through the list box, the panel is updated for each file.

New

The **File | New** command lets you open a new Edit window with the default name NONAME xx .PAS (the xx stands for a number from 00 to 99). These NONAME files are used as a temporary edit buffer; Turbo Pascal prompts you to name a NONAME file when you save it.

Save

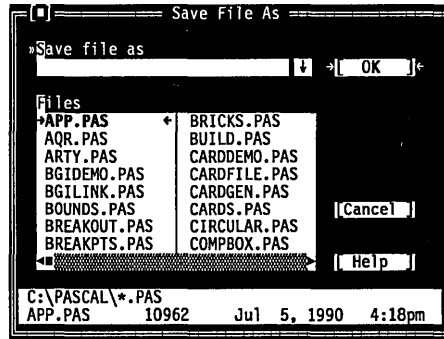
F2

The **File | Save** command saves the file in the active Edit window to disk. (This menu item is disabled if there's no active Edit window.) If the file has a default name (NONAME00.PAS, or the like), Turbo Pascal opens the Save File As dialog box to let you rename and save it in a different directory or on a different drive. This dialog box is identical to the one opened for the Save As command, described next.

Save As

The **File | Save As** command lets you save the file in the active Edit window under a different name, in a different directory, or on a different drive. When you choose this command, you see the Save File As dialog box:

Figure 7.2
The Save File As dialog box



Enter the new name, optionally with drive and directory, and click or choose **OK**. All windows containing this file are updated with the new name. If you pick an existing file name, that file will be overwritten.

Save All

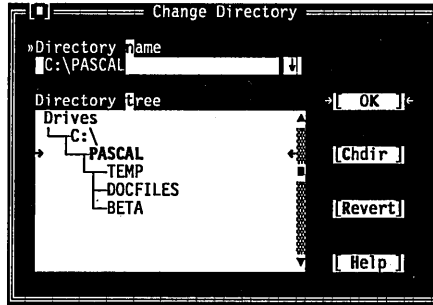
The **File | Save All** command works just like the **Save** command except that it saves the contents of all modified files, not just the file in the active Edit window. This command is disabled if no Edit windows are open.

Change Dir

The **File | Change Dir** command lets you specify a drive and a directory to make current. The current directory is the one Turbo Pascal uses to save files and to look for files. (When using relative paths in **Options | Directories**, they are relative to this current directory only.)

Here is what the Change Directory dialog box looks like:

Figure 7.3
The Change Dir dialog box



There are two ways to change directories:

- Type in the path of the new directory in the input box and press *Enter*, or
- Choose the directory you want in the Directory tree (if you're using the keyboard, press *Enter* to make it the current directory), then choose **OK** or press *Esc* to exit the dialog box.

If you choose the **OK** button, your changes will be made and the dialog box put away. If you choose the **Chdir** button, the Directory Tree list box changes to the selected directory and displays the subdirectories of the currently highlighted directory (pressing *Enter* or double-clicking on that entry gives you the same result). If you change your mind about the directory you've picked and you want to go back to the previous one (*and you've yet to exit the dialog box*), choose the **Revert** button.

Print

The **File | Print** command lets you print the contents of the active Edit window. Turbo Pascal expands tabs (replaces tab characters with the appropriate number of spaces) and then sends it to the DOS print handler. This command is disabled if the active window cannot be printed. Use *Ctrl-K P* to print selected text only.

Get Info

The **File | Get Info** command displays a box with information on the current file.

Figure 7.4
The Get Info box



The information here is for display only; you can't change any of the settings in this box. After reviewing the information in this box, press *Enter* to put the box away.

DOS Shell

The **File | DOS Shell** command lets you temporarily exit Turbo Pascal to enter a DOS command or program. To return to Turbo Pascal, type **EXIT** and press *Enter*.

Warning: Don't install any TSR programs (like SideKick) if you've shelled to DOS, because memory may get misallocated.

You may find that when you're debugging, there's not enough memory to execute this command. If that's the case, terminate the debug session by choosing **Run | Program Reset (Ctrl-F2)**.

Note: In dual monitor mode, the DOS command line appears on the Turbo Pascal screen rather than the User Screen. This allows you to switch to DOS without disturbing the output of your program. Since your program output is available on one monitor in the system, **Window | User Screen** and *Alt-F5* are disabled.

Exit

Alt X

The **File | Exit** command exits Turbo Pascal, removes it from memory, and returns you to the DOS command line. If you have made any changes that you haven't saved, Turbo Pascal asks you if you want to save them before exiting.

Edit menu

Alt E

The **Edit** menu lets you cut, copy, and paste text in Edit windows. You can also open a Clipboard window to view or edit its contents.

Before you can use most of the commands on this menu, you need to know about selecting text (because most editor actions apply to selected text). Selecting text means highlighting it. You can select text either with keyboard commands or with a mouse; the principle is the same even though the actions are different.

From the keyboard you can use any of these methods:

New!



- Press *Shift* while pressing any arrow key.
- To select text from the keyboard, press *Ctrl-K B* to mark the start of the block. Then move the cursor to the end of the text and press *Ctrl-K K*.
- To select a single word, move the cursor to the word and press *Ctrl-K T*.
- To select an entire line, press *Ctrl-K L*.

With a mouse:



- To select text with a mouse, drag the mouse pointer over the desired text. If you need to continue the selection past a window's edge, just drag off the side and the window will automatically scroll.
- To select a single line, double-click anywhere in the line.
- To select text line-by-line, click-drag over the text (that is, click once and then quickly press the mouse button again and begin to drag).
- To extend or reduce the selection, Shift-click anywhere in the document (that is, hold *Shift* and click).

Once you have selected text, the commands in the **Edit** menu become available, and the Clipboard becomes usable.

The Clipboard is the magic behind cutting and pasting. It's a special window in Turbo Pascal that holds text that you have cut or copied, so you can paste it elsewhere. The Clipboard works in close concert with the commands in the **Edit** menu.

Here's an explanation of each command in the **Edit** menu.

Restore Line

The **Edit | Restore Line** command takes back the last editing command you performed on a line (including typing text on a blank line or *Ctrl-Y*). **Restore Line** works only on the last modified or deleted line.

Cut

Shift **Del**

The **Edit | Cut** command removes the selected text from your document and places the text in the Clipboard. You can then paste that text into any other document (or somewhere else in the same document) by choosing **Paste**. The text remains selected in the Clipboard so that you can paste the same text many times.

Copy

Ctrl **Ins**

The **Edit | Copy** command leaves the selected text intact but places an exact copy of it in the Clipboard. You can then paste that text into any other document by choosing **Paste**. You can also copy text from a Help window: With the keyboard, use *Shift* and the arrow keys; with the mouse, click and drag the text you want to copy.

Paste

Shift **Ins**

The **Edit | Paste** command inserts text from the Clipboard into the current window at the cursor position. The text that is actually pasted is the currently marked block in the Clipboard window.

Copy Example

The **Edit | Copy Example** command copies the preselected example text in the current Help window to the Clipboard. The examples are already predefined as pastable blocks, so you don't need to bother with marking the example you want.

Show Clipboard

The **Edit | Show Clipboard** command opens the Clipboard window, which stores the text you cut and copy from other windows. The text that's currently selected (highlighted) is the text that gets pasted. And you can edit the Clipboard so that the text you paste is precisely the text you want.

The Clipboard window is just like other Edit windows except when you cut or copy text. When you select text in the Clipboard window and choose **Cut** or **Copy**, the selected text immediately appears at the bottom of the window. (Remember, any text that

you cut or copy is appended to the end of the Clipboard—so you can paste it later.)

Clear

Ctrl **Del**

The **Edit | Clear** command removes the selected text but does not put it into the Clipboard. This means you cannot paste the text as you could if you had chosen **Cut** or **Copy**. The cleared text is not retrievable. You can clear the Clipboard itself by selecting all the text in the Clipboard, then selecting **Edit | Clear**.

Search menu

Alt **S**

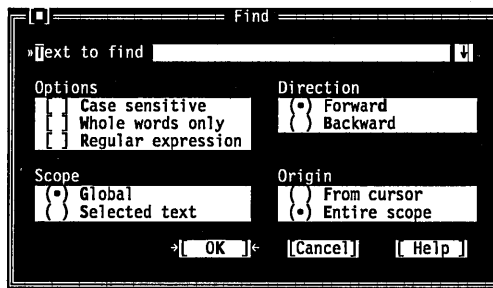
The **Search** menu lets you search for text, procedure declarations, and error locations in your files.

Find

Alt **S** **F**

The **Search | Find** command displays the Find dialog box, which lets you type in the text you want to search for and set options that affect the search. (*Ctrl-Q F* is another shortcut for this command.)

Figure 7.5
The Find dialog box



The Find dialog box contains several buttons and check boxes.

Options You can choose from three items in the Options check boxes:

Case sensitive

Check the Case Sensitive box if you do want Turbo Pascal to differentiate uppercase from lowercase.

Whole words only

Check the Whole Words Only box if you want Turbo Pascal to search for words only (that is, a string with punctuation or space characters on both sides).

Regular expression

Check the Regular Expression box if you want Turbo Pascal to recognize GREP-like wildcards in the search string. The wildcards are ^, \$, ., *, +, [], and \. Here's what they mean:

-
- ^ A circumflex at the start of the string matches the start of a line.
 - \$ A dollar sign at the end of the expression matches the end of a line.
 - . A period matches any character.
 - * A character followed by an asterisk matches any number of occurrences (including zero) of that character. For example, *bo** matches *bot*, *b*, *boo*, and also *be*.
 - + A character followed by a plus sign matches any number of occurrences (but not zero) of that character. For example, *bo+* matches *bot* and *boo*, but not *be* or *b*.
 - [] Characters in brackets match any one character that appears in the brackets but no others. For example *[bot]* matches *b*, *o*, or *t*.
 - [^] A circumflex at the start of the string in brackets means *not*. Hence, *[^bot]* matches any characters except *b*, *o*, or *t*.
 - [-] A hyphen within the brackets signifies a range of characters. For example, *[b-o]* matches any character from *b* through *o*.
 - \ A backslash before a wildcard character tells Turbo Pascal to treat that character literally, not as a wildcard. For example, *\^* matches *^* and does not look for the start of a line.
-

Enter the string in the input box and choose **OK** to begin the search, or choose **Cancel** to forget it. If you want to enter a string that you searched for previously, press ↓ to choose from the history list.

You can also pick up the word that your cursor is currently on in the Edit window and use it in the Find box by simply invoking **Find** from the **Search** menu. You can take additional characters from the text by pressing → .

Direction

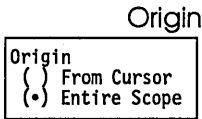
Direction
 Forward
 Backward

Choose from the Direction radio buttons to decide which direction you want Turbo Pascal to search—starting from the origin (settable with the Origin radio buttons).

Scope

Scope
 Global
 Selected text

Choose from the Scope radio buttons to determine how much of the file to search in. You can search the entire file (Global) or only the selected text.



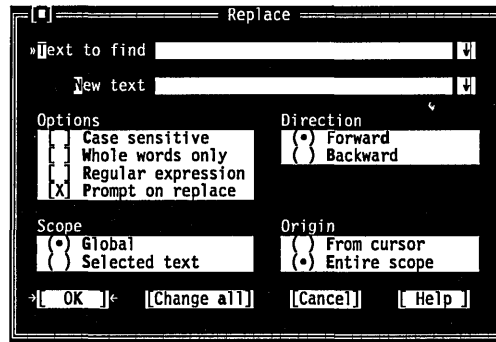
Choose from the Origin radio buttons to determine where the search begins. When Entire Scope is chosen, the Direction radio buttons determine whether the search starts at the beginning or the end of the chosen scope. You choose the range of scope you want with the Scope radio buttons.

Replace



The **Search | Replace** command displays a dialog box that lets you type in the text you want to search for and text you want to replace it with. (*Ctrl-Q A*) is another shortcut for this command.)

Figure 7.6
The Replace dialog box



The Replace dialog box contains several radio buttons and check boxes—many of which are identical to the Find dialog box, discussed previously. An additional checkbox, Prompt on Replace, controls whether you're prompted for each change.

After you've entered the search string and the replacement string in the input boxes, choose **OK** or **Change All** to begin the search, or choose **Cancel** to forget it. If you want to enter a string you used previously, press ↓ to choose from the history list.

If Turbo Pascal finds the specified text, it asks you if you want to make the replacement. If you choose **OK**, it will find and replace only the first instance of the search item. If you choose **Change All**, it replaces all occurrences found, as defined by **Direction**, **Scope**, and **Origin**.

Like in the Find dialog box, you can pick up the word your cursor is currently on in the Edit window and use it in the **Text to Find** input box by simply invoking **Find** or **Replace** from the **Search** menu. And you can add more text from the Edit window by pressing →.

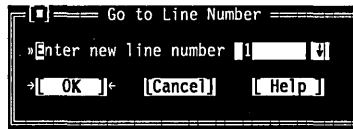
Search Again

Ctrl **L**

The **Search | Search Again** command repeats the last **Find** or **Replace** command. All settings you made in the last dialog box used (**Find** or **Replace**) remain in effect when you choose **Search Again**.

Go to Line Number

Figure 7.7
The Go to Line Number dialog box

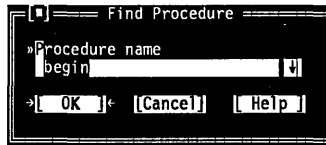


The **Search | Go to Line Number** command prompts you for the line number you want to find:

Turbo Pascal displays the current line number and column number in the lower left corner of every Edit window.

Find Procedure

Figure 7.8
The Find Procedure dialog box



The **Search | Find Procedure** command displays a dialog box for you to enter the name of a procedure or function to search for. This command is available only during a debugging session.

Enter the name of a procedure or press ↓ to choose a name from the history list. As opposed to the **Search | Find** command, this command finds the declaration of the procedure, not instances of its use.

Find Error

Alt **F8**

The **Search | Find Error** command finds the location of a run-time error. When a run-time error occurs, the address in memory of where it occurred is given in the format *seg:ofs*. When you return to the IDE, Turbo Pascal automatically locates the error for you. This command allows you to find the error again, given the *seg* and *ofs* values.

For **Find Error** to work, you must set the **Debugging** check box to **Integrated** (in the **Options | Debugger** dialog box).

If run-time errors occur in a program running within the IDE, the default values for the error address are set automatically. This allows you to relocate the error after changing files. (Note that if you just move around in the same file, you can get back to the error location with the *Ctrl-Q W* command.)

When run-time errors occur under DOS, record the segment and offset displayed onscreen. Then load the main program into the editor or specify it as the **Main File**. Be sure to set the **Destination** to *Disk*, then type in the segment offset value.

When you enter the error address, you must give it in hexadecimal segment and offset notation. The format is "xxxx:yyyy"; for example, "2BE0:FFD4."

Run menu

Alt R

The **Run** menu's commands run your program, and also start and end debugging sessions.

Run

Ctrl F9

The **Run | Run** command runs your program, using any parameters you pass to it with the **Run | Parameters** command. If the source code has been modified since the last compilation, the compiler's built-in project manager will automatically do a make and link your program.

If you want to have all Turbo Pascal's features available, keep Debugging set to Integrated.

If you don't want to debug your program, you can compile and link it with both the **Debugging** check boxes unchecked (which gives the program more room to run) in the **Options | Debugger** dialog box. If you compile your program with this check box set to **Integrated**, the resulting executable code will contain debugging information that will affect the behavior of the **Run | Run** command in the following ways:

If you have *not* modified your source code since the last compilation,

- the **Run | Run** command causes your program to run to the next breakpoint, or to the end if no breakpoints have been set.

If you *have* modified your source code since the last compilation,

- and if you're already stepping through your program using the **Run | Step Over** or **Run | Trace Into** commands, **Run | Run** prompts you whether you want to rebuild your program:
 - If you answer yes, the built-in project manager will make and link your program, and set it to run from the beginning.
 - If you answer no, your program runs to the next breakpoint or to the end if no breakpoints are set.
- and if you are not in an active debugging session, the built-in project manager makes your program and sets it to run from the beginning.

Pressing *Ctrl-Break* causes Turbo Pascal to stop execution on the next source line in your program. If Turbo Pascal is unable to find a source line, a second *Ctrl-Break* will terminate the program and return you to the IDE.

Program Reset

Ctrl **F2**

The **Run | Program Reset** command stops the current debugging session, releases memory your program has allocated, and closes any open files that your program was using.

Go to Cursor

F4

The **Run | Go to Cursor** command runs your program from the run bar (the highlighted bar in your code) to the line the cursor is on in the current Edit window. If the cursor is at a line that does not contain an executable statement, the command displays a warning. **Run | Go to Cursor** can also initiate a debug session.

Go to Cursor does not set a permanent breakpoint, but it does allow the program to stop at a permanent breakpoint if it encounters one before the line the cursor is on. If this occurs, you must choose the **Go to Cursor** command again.

Use **Go to Cursor** to advance the run bar to the part of your program you want to debug. If you want your program to stop at a certain statement every time it reaches that point, set a breakpoint on that line.

Note that if you position the cursor on a line of code that is not executed, your program will run to the next breakpoint or the end

if no breakpoints are encountered. You can always use *Ctrl-Break* to stop a running program.

Trace Into

F7

The **Run | Trace Into** command runs your program statement by statement. When it reaches a procedure call, it executes each statement within the procedure, instead of executing the procedure as a single step (see **Run | Step Over**). If a statement contains no calls to procedures accessible to the debugger, **Trace Into** stops at the next executable statement.

Use the **Trace Into** command to move the run bar into a procedure called by the procedure you are now debugging. If the statement contains a call to a procedure accessible to the debugger, **Trace Into** halts at the beginning of the procedure's definition. Subsequent **Trace Into** or **Step Over** commands run the statements in the procedure's definition. When the debugger leaves the procedure, it resumes evaluating the statement that contains the call:

```
if Min <= Max then
  DoSomething;
```



The **Trace Into** command recognizes only procedures defined in a source file compiled with two options set on:

- In the Compiler Options dialog box (**Options | Compiler**), check **Debug Information**.
- In the Debugger dialog box (**Options | Debugger**), check **Integrated**.

Step Over

F8

The **Run | Step Over** command executes the next statement in the current procedure. It does not trace into calls to lower-level procedures, even if they are accessible to the debugger.

Use **Step Over** to run the procedure you are now debugging, one statement at a time without branching off into other procedures.

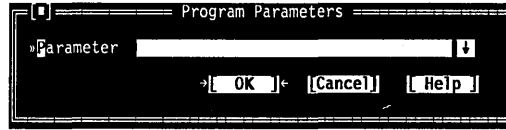
Parameters

The **Run | Parameters** command allows you to give your running programs command-line arguments exactly as if you had typed

them on the DOS command line. DOS redirection commands will be ignored.

When you choose this command, a dialog box appears with a single input box.

Figure 7.9
The Program Parameters
dialog box



You only need to enter the parameters here, not the program name.

Parameters take affect only when your program is started. If you are already debugging and wish to change the parameters, you can select **Program Reset** and then start the program with the new parameters.

Compile menu

Alt **C**

Use the commands on the **Compile** menu to compile, make, or build the program in the active window. To use the **Compile**, **Make**, and **Build** commands, you must have a file open in an *active* Edit window. For example, if you open an Output or Watch window, those selections will be disabled.

Compile

Alt **F9**

The **Compile | Compile** command compiles the active editor file. When Turbo Pascal is compiling, a status box pops up to display the compilation progress and results. When compiling/linking is complete, press any key to remove this box. If any errors occur, the Edit window containing the offending source code becomes active, an error message is displayed, and the cursor is placed on the first error location.

Make

F9

The **Compile | Make** command invokes the built-in project manager to make an .EXE file.

- If a **Primary File** has been named, that file is compiled; otherwise, the file in the active Edit window is compiled. Turbo Pascal checks all files upon which the file being compiled depends.

- If the source file for a given unit has been modified since the .TPU (object code) file was created, then that unit is recompiled.
- If the interface for a given unit has been changed, then all other units that depend upon it are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file is newer than the unit's .TPU file, then the unit is recompiled.
- If a unit includes an Include file and the Include file is newer than that unit's .TPU file, then the unit is recompiled.

If the source to a unit (.TPU file) cannot be located, that unit is *not* compiled, but is used as is.

Compile | Make rebuilds only the files that aren't current and the one in the active Edit window (or **Primary File** if specified).

Build

The **Compile | Build** command rebuilds all the files regardless of whether they're out of date. This command is similar to **Compile | Make** except that it is unconditional.

Destination

The **Compile | Destination** command lets you specify whether the executable code will be stored on disk (as an .EXE file) or whether it will be stored in memory (and thus lost when you exit Turbo Pascal). Note that even if **Destination** is set to *Memory*, any units recompiled during a **Make** or **Build** have their .TPU files updated on disk.

Setting Destination to Disk increases the memory available in the IDE to compile and debug your program.

If **Destination** is set to *Disk*, then an .EXE file is created and its name is derived from one of two names, in the following order: the **Primary File** name or, if none is specified, the name of the file in the active Edit window.

The resulting .EXE and .TPU (if any) is stored in the same directory as their respective source files, or in the EXE & TPU Directory (**Options | Directories**), if one is specified.

Primary File

Select the **Primary File** command to specify which .PAS file will get compiled when you use **Compile | Make** (F9) or **Build** (Alt-C B). You'll want to use this option when you're working on a program that uses several unit or Include files. No matter which file you've

been editing, **Make** or **Build** will always operate on your primary file. If you specify another file as a primary file, but want to compile only the file in the selected Edit window, choose **Compile** (*Alt-F9*).

Debug menu

Alt **D**

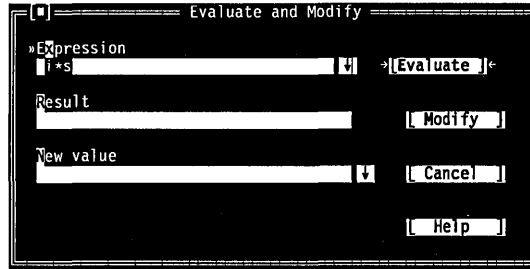
The commands on the **Debug** menu control all the features of the integrated debugger. You can change default settings for these commands in the **Options | Debugger** dialog box. For more about debugging, refer to Chapter 5, "Debugging Turbo Pascal programs."

Evaluate/Modify

Ctrl **F4**

The **Debug | Evaluate/Modify** command evaluates a variable name or expression, displays its value, and, if appropriate, lets you modify the value. The command opens a dialog box containing the Expression, the Result, and the New Value fields.

Figure 7.10
The Evaluate/Modify dialog box



The Evaluate button is the default button; when you tab to the New Value field, the Modify button becomes the default.

The Expression field shows a default expression consisting of the word at the cursor in the Edit window. You can evaluate the default expression by pressing *Enter*, or you can edit or replace it first. You can also press *→* to extend the default expression by copying additional characters from the Edit window.

If the debugger can evaluate the expression, it displays the value in the Result field. If the expression refers to a variable or simple data element, you can move the cursor to the New Value field and enter an expression as the new value.

Press *Esc* to close the dialog box. If you've changed the contents of the New Value field but do not select *Modify*, the debugger will ignore the New Value field when you close the dialog box.

Use a repeat expression to display the values of consecutive data elements. For example, for an array of integers named *ListInt*,

- `ListInt[0],5` displays five consecutive integers in decimal.
- `ListInt[0],5x` displays five consecutive integers in hex.

An expression used with a repeat count must represent a single data element. The debugger views the data element as the first element of an array if it isn't a pointer, or as a pointer to an array if it is.

The **Debug | Evaluate/Modify** command displays each type of value in an appropriate format. For example, it displays an integer in base 10 (decimal), and an array as a pointer in base 16 (hexadecimal). To get a different display format, precede the expression with a comma followed by one of the format specifiers shown in Table 7.1.

Table 7.1: Format specifiers recognized in debugger expressions

Character	Function
C	Character. Shows special display characters for control characters (ASCII 0 through 31); by default, such characters are shown ASCII values using the <code>##xx</code> syntax. Affects characters and strings.
S	String. Shows control characters (ASCII 0 through 31) as ASCII values using the <code>##xx</code> syntax. Since this is the default character and string display format, the S specifier is only useful in conjunction with the M specifier.
D	Decimal. Shows all integer values in decimal. Affects simple integer expressions as well as structures (arrays and records) containing integers.
\$, H, or X	Hexadecimal. Shows all integer values in hexadecimal with the \$ prefix. Affects simple integer expressions as well as structures (arrays and records) containing integers.
F _n	Floating point. Shows <i>n</i> significant digits (<i>n</i> is an integer between 2 and 18). The default value is 11. Affects only floating-point values.
M	Memory dump. Displays a memory dump, starting with the address of the indicated expression. The expression must be a construct that would be valid on the left side of an assignment statement, i.e., a construct that denotes a memory address; otherwise, the M specifier is ignored. By default, each byte of the variable is shown as two hex digits. Adding a D specifier with the M causes the bytes to be displayed in decimal. Adding an H , \$, or X specifier causes the bytes to be displayed in hex with a \$ prefix. An S or a C specifier causes the variable to be displayed as a string (with or without special characters). The default number of bytes displayed corresponds to the size of the variable, but a repeat count can be used to specify an exact number of bytes.
P	Pointer. Displays pointers in <code>seg : ofs</code> format rather than the default <code>Ptr(seg,ofs)</code> format. For example, displays <code>3EA0:0020</code> instead of <code>Ptr(\$3EA0,\$20)</code> . Affects only pointer values.
R	Record. Displays record and object field names such as (<code>X:1;Y:10;Z:5</code>) instead of (1, 10, 5). Affects only record variables and objects with fields.

Watches

The **Debug | Watches** command opens a pop-up menu of commands that control the use of watchpoints. The following sections describe the commands in this pop-up menu.

Add Watch The **Add Watch** command inserts a watch expression into the Watch window.

Ctrl **F7**

When you choose this command, the debugger opens a dialog box and prompts you to enter a watch expression. The default expression is the word at the cursor in the current Edit window. There's also a history list available if you want to watch an expression you've used before.

When you type a valid expression and press *Enter* or click **OK**, the debugger adds the expression and its current value to the Watch window. If the Watch window is the active window, you can insert a new watch expression by pressing *Ins*.

Delete Watch While you're in the Watch window, you can select **Delete Watch** to delete the current watch expression from the Watch window, or press either *Del* or *Ctrl-Y*. The current watch expression is marked by a bullet in the left margin.

Edit Watch The **Edit Watch** command allows you to edit the current watch expression in the Watch window. When you choose this command, you'll get a dialog box that contains a copy of the current watch expression. Edit the expression and then press *Enter*; this replaces the original expression with the edited one.

To edit a watch expression from inside the Watch window, select the expression and press *Enter*.

Remove All Watches The **Remove All Watches** command deletes all watch expressions from the Watch window.

Toggle Breakpoint

Ctrl **F8**

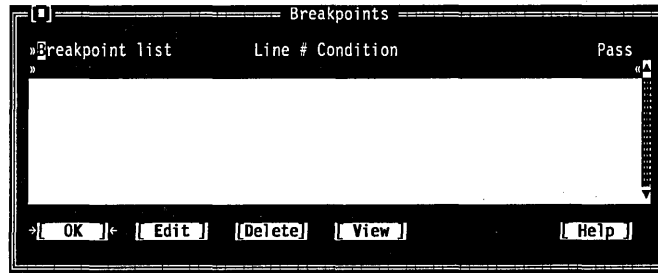
The **Debug | Toggle Breakpoint** command lets you set or clear an unconditional breakpoint on the line where the cursor is positioned. When a breakpoint is set, it is marked by a breakpoint

highlight. (The following section discusses breakpoints in more depth.)

Breakpoints

The **Debug | Breakpoints** command opens a dialog box that lets you control the use of breakpoints:

Figure 7.11
The Breakpoints dialog box



The dialog box shows you all set breakpoints, their line numbers, and the conditions. The condition has a history list so you can select a breakpoint condition that you've used before.

Whenever your running program encounters a breakpoint, it will stop with a run bar positioned on the line with the breakpoint.

*You can set an unconditional breakpoint by choosing **Debug | Toggle Breakpoint**.*

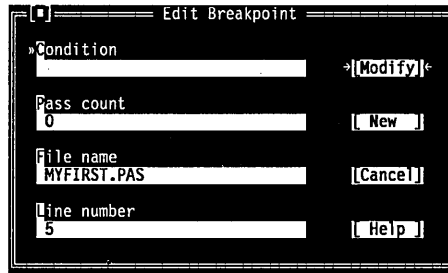
Before you compile a source file, you can set a breakpoint on any line, even a blank line or a comment. When you compile and run the file, Turbo Pascal validates any breakpoints that are set and gives you a chance to remove, ignore, or change invalid breakpoints. When you are debugging the file, Turbo Pascal knows which lines contain executable statements, and will warn you if you try to set invalid breakpoints.

*This dialog box has no **Cancel** button, so edit and delete with care.*

You can remove breakpoints from your program by choosing the **Delete** button. You can also view the source where existing breakpoints are set by choosing the **View** button. **View** moves the cursor to the selected breakpoint in the **Edit** window (it does not run your code).

Choose **Edit** to add a new breakpoint to the list. When you edit a breakpoint, this dialog box pops up over the first one:

Figure 7.12
The Edit Breakpoint dialog
box



Again, line number and conditions are that of the breakpoints you've set. Use Pass Count to set how many times the breakpoint should be skipped before stopping.

When a source file is edited, each breakpoint "sticks" to the line where it is set. Breakpoints are lost only when

- you delete the breakpoint in the Breakpoints dialog box
- you delete the source line a breakpoint is set on
- you clear a breakpoint with Toggle Breakpoint

Turbo Pascal will attempt to track breakpoints in two cases:

- If you edit a file containing breakpoints and then don't save the edited version of the file.
- If you edit a file containing breakpoints and then continue the current debugging session without remaking the program. (Turbo Pascal displays the warning prompt "Source modified, rebuild?")

Breakpoints are saved in the TURBO.DSK file if this option is enabled.

This dialog box also has a New button, which lets you enter breakpoint information for a new breakpoint, and a Modify button, which accepts the settings of the box.

Options menu

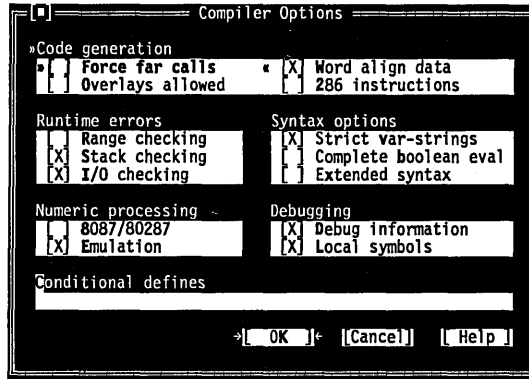


The Options menu contains commands that let you view and change various default settings in Turbo Pascal. Most of the commands in this menu lead to dialog boxes.

Compiler

The **Options | Compiler** command displays a dialog box that lets you set several options that affect code compilation. Here's what the dialog box looks like:

Figure 7.13
The Compiler Options dialog box



The following sections describe these commands.

Code Generation

You can use the check boxes in the **Code Generation** group to tell the compiler to prepare your code in certain ways. Here are what the various buttons and check boxes mean:

*This is equivalent to the **\$F** compiler directive.*

- **Force Far Calls** allows you to force all procedures and functions to use the far call model. If the option is not enabled, the compiler will use the near call models for any procedures or functions within the file being compiled.

*This is equivalent to the **\$O+** compiler directive.*

- **Overlays Allowed** enables or disables overlay code generation. Turbo Pascal allows a unit to be overlaid only if it was compiled with **Overlays Allowed** checked (set to on). In this state, the code generator takes special precautions passing string and set constant parameters from one overlaid procedure or function to another.

Checking the **Overlays Allowed** check box does not force you to overlay that unit. It instructs Turbo Pascal to ensure that the unit can be overlaid, if so desired. If you develop units that you plan to use in overlaid as well as non-overlaid applications, then compiling them with **Overlays Allowed** checked ensures that you can indeed do both with the same unit.

- **Word Align Data** (when checked) tells Turbo Pascal to align noncharacter data at even addresses. When this option is off (unchecked), Turbo Pascal uses byte-aligning, where data can be aligned at either odd or even addresses, depending on which is the next available address. (This is equivalent to the **\$A** compiler directive.)

Word alignment increases the speed with which 8086 and 80286 processors fetch and store the data.

- **286 Instructions** tells Turbo Pascal to generate code for the 80286 instruction set. Note that programs compiled with 80286 code generation turned on do not check for the presence of an 80286 at run time. (This is equivalent to the **\$G** compiler directive.)

Run-time Errors The Run-time Errors group let you select which run-time errors are generated.

- When **Range Checking** is checked, the compiler generates code to check that array and string subscripts are within bounds, and that assignments to scalar-type variables don't exceed their defined ranges. If the check fails, the program halts with a run-time error. When unchecked, **Range Checking** is disabled. (This is equivalent to the **\$R** compiler directive.)
- When **Stack Checking** is checked, the compiler generates code to check that space is available for local variables on the stack before each call to a procedure or function. If the check fails, the program halts with a run-time error. When unchecked, **Stack Checking** is disabled. (This is equivalent to the **\$S** compiler directive.)
- When **I/O Checking** is checked, the compiler generates code to check for I/O errors after every I/O call. If the check fails, the program halts with a run-time error. When the option is unchecked, **I/O Checking** is disabled; however, the user can test for I/O errors via the system function *IOResult*. (I/O checking is equivalent to the **\$I** compiler directive.)

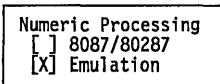
Syntax Options This group lets you select the type of syntax options you want to search for.

- With the **Strict Var-Strings** option enabled, the compiler compares the declared type of a **var**-type string parameter with the type of the actual parameter being passed. If they are not identical, a compiler error occurs. With the option disabled, no

such type checking is done. (This option is equivalent to the **\$V** compiler directive.)

- With **Complete Boolean Evaluation** enabled, all terms in a Boolean expression are always evaluated. If disabled, the compiler generates code to terminate evaluation of a Boolean expression as soon as possible; for example, in the expression `if False and MyFunc. . .`, the function *MyFunc* would never be called. (This option is equivalent to the **\$B** compiler directive.)
- With the **Extended Syntax** option enabled, Turbo Pascal's syntax is extended to let you use user-defined function calls as statements, as if they were procedures. When this option is disabled, this extension is disabled. Refer to Chapter 21 in the *Programmer's Guide* for more information. (This option is equivalent to the **\$X** directive.)

Numeric Processing



This option is ignored unless 8087/80287 is enabled.

The Numeric Processing options let you decide how you want Turbo Pascal to handle floating-point numbers.

- Choose **8087/80287** to generate direct 8087 or 80287 inline code. This option is equivalent to the **\$N** compiler directive.
- Choose **Emulation** if you want Turbo Pascal to detect whether your computer has an 80x87 coprocessor (and to use it if you do). If it is not present, Turbo Pascal emulates the 80x87. The **\$E** compiler directive is equivalent to this option.

For more information about the compiler directive equivalents, refer to Chapter 21 in the *Programmer's Guide*.

Debugging

You can set the options in the Debugging group to turn on or off debug information or local symbol generation.

- **Checking Debug Information** enables the generation of debug information, which consists of a line-number table for each Pascal statement that maps object code addresses into source text numbers. (This is equivalent to the **\$D** compiler directive.)

When you've checked **Debug Information** for a given program or unit, the IDE allows you to single-step and set breakpoints in that module. Also, when a run-time error occurs in a program or unit compiled with **Debug Information** checked, Turbo Pascal can automatically take you to the statement that caused the error with **Search | Find Error**.

For units, the debug information is recorded in the .TPU file, along with the unit's object code. Debug information increases

Debug Information is usually used in conjunction with the Local Symbols command.

the size of .TPU files, and takes up additional memory when programs compile that use the unit, but it doesn't affect the size or speed of the executable program.

To use Turbo Debugger, **\$D** should be on.

Those parts of your source code compiled and linked with Debug Information unchecked are not accessible to the debugger. If disk space is at a premium, uncheck Debug Information to create smaller .TPU files and use less memory during compilation and run time.

- **Checking Local Symbols** enables the generation of local symbol information, which consists of the names and types of all local variables and constants in a module (the symbols in the module's implementation part, and the symbols within the module's procedures and functions). (Local Symbols is equivalent to the **\$L** compiler directive.)

Local Symbols is ignored if Debug Information is unchecked.

When you've checked **Local Symbols** for a given program or unit, the IDE allows you to examine and modify the module's local variables. Also, calls to the module's procedures and functions can be examined with the **Window | Call Stack** command.

For units, the local symbol information is recorded in the .TPU file along with the unit's object code. Local symbol information increases the size of .TPU files and takes up additional room when you compile programs that use the unit, but it doesn't affect the size or speed of the executable program.

Conditional Defines

Use the **Conditional Defines** input box to enter define symbols to be referenced in conditional compilation directives (refer to Chapter 21 in the *Programmer's Guide*). You can separate multiple defines with semicolons (;), for example,

```
TestCode;DebugCode
```

Memory Sizes

The **Memory Sizes** options let you configure the default memory requirements for a program. All three settings can be specified in your source code using the **\$M** compiler directive. If you attempt to run your program and there is not enough heap space to satisfy the specified requirement, the program aborts with a run-time error. (This is equivalent to the **\$M** compiler directive.)

- **Stack Size** specifies the size (in bytes) of the stack segment. The default size is 16,384, the maximum size is 65,520.

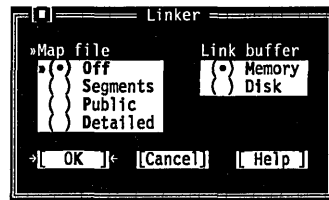
You must specify a smaller limit if your program executes other programs. Refer to Exec in Chapter 1 of the Library Reference for more detail.

- **Low Heap Limit** specifies the minimum required heap size (in bytes). The default minimum size is 0K.
- **High Heap Limit** specifies the maximum amount of memory (in bytes) to allocate to the heap. The default is 655360, which (on most systems) will allocate all available memory to the heap. This value must be greater than or equal to the smallest heap size.

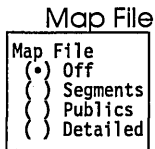
Linker

The **Options | Linker** command opens a dialog box that lets you make several settings that affect linking:

Figure 7.14
The Linker dialog box



This dialog box has several radio buttons, which are described in the following sections.



Use the **Map File** radio buttons to choose the type of map file to be produced. For settings other than **Off**, the map file is placed in the EXE and TPU directory defined in the **Options | Directories** dialog box. The default setting for the map file is off. (Segments, Publics, and Detailed are equivalent to the **/GS**, **/GP**, and **/GD** command-line options.)

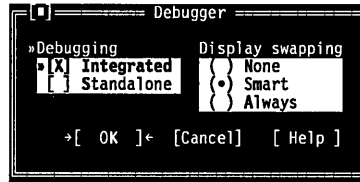
Link Buffer (memory)

The **Link Buffer** option tells Turbo Pascal to use **Memory** or **Disk** for the link buffer. When you select the **Memory** radio button, it speeds compilation, but you may run out of memory for large programs. Selecting the **Disk** radio button frees up memory, but slows compilation. (This is equivalent to the **/L** command-line option in TPC.EXE.)

Debugger

The **Options | Debugger** command opens a dialog box that lets you make several settings affecting the integrated debugger:

Figure 7.15
The Debugger dialog box



The following sections describe the contents of this box.

Debugging

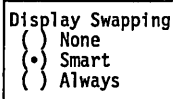


The Debugging check boxes determine whether debugging information is included in the executable file and how the .EXE is run under Turbo Pascal.

- Choose **Integrated** (the default) to debug programs with the integrated debugger or the standalone Turbo Debugger.
- Choose **Standalone** to debug programs with Turbo Debugger.

The **Integrated**, **Standalone** (Options | Debugger), and **Map File** options (Options | Linker) produce complete and local symbol information for a module only if you've compiled that module with **Debug Information** and **Local Symbols** checked, respectively.

Display swapping



The Display Swapping radio buttons let you set when the integrated debugger will change display windows while running a program.

If you're debugging in dual monitor mode (used the Turbo Pascal command-line **/d** option), you can see your program's output on one monitor and the Turbo Pascal screen on the other. In this case, Turbo Pascal never swaps screens and the Display Swapping setting has no effect.

- If you set Display Swapping to **None**, the debugger does not swap the screen at all. You should only use this setting for debugging sections of code that you're certain do not output to the screen.
- When you run your program in debug mode with the default setting of **Smart**, the debugger looks at the code being executed to see whether it will generate output to the screen. If the code does output to the screen (or if it calls a procedure), the screen is swapped from the IDE screen to the User screen long enough for output to be displayed, then is swapped back. Otherwise, no swapping occurs. Be aware of the following with smart swapping:

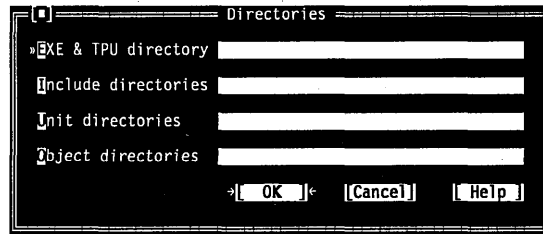
- It swaps on any procedure call, even if the procedure does no screen output.
 - In some situations, the IDE screen might be modified without being swapped; for example, if a timer interrupt routine writes to the screen.
- If you set Display Swapping to **Always**, the debugger swaps screens every time a statement executes. You should choose this setting any time the IDE screen is likely to be overwritten by your running program.

Directories

The **Options | Directories** command lets you tell Turbo Pascal where to find the files it needs to compile, link, and output files.

This command opens a dialog box containing four input boxes. The dialog box looks like this:

Figure 7.16
The Directories dialog box



Use the following guidelines when entering directories in these input boxes:

- You must separate multiple directory path names (if allowed) with a semicolon (;). You can use up to a maximum of 127 characters (including whitespace).
- Whitespace before and after the semicolon is allowed but not required.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one. For example,

```
C:\PASCAL;C:\PASCAL\MYPROJS;A:TURBO\EXAMPLES;
```

Here is a description of each input box.

- Enter the output directory for .EXE or .TPU files in the **EXE and TPU directory** input box. If the entry is blank, the files are stored in the directory where the source is found. .MAP files are

also stored here if Map File (Options | Linker) is set to anything besides Off.

- Use the **Include Directories** input box to specify the directories that contain your standard Include files. Include files are those specified with the `{$I filename}` compiler directive. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

- Use the **Unit Directories** input box to specify the directories that contain your Turbo Pascal unit files. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

To use the *Graph* unit, for example, you could create a directory `\TURBO\BGI`, copy GRAPH.TPU and specify a unit directory of `\TURBO\BGI`. If you also wanted to keep other units in a `\TURBO\UNITS` directory, your unit directory would be `\TURBO\UNITS; \TURBO\BGI`.

- Use the **Object Directory** input box to specify the directories that contain .OBJ files (assembly language routines). When Turbo Pascal encounters a `{$L filename}` directive, it looks first in the current directory, then in the directories specified here. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

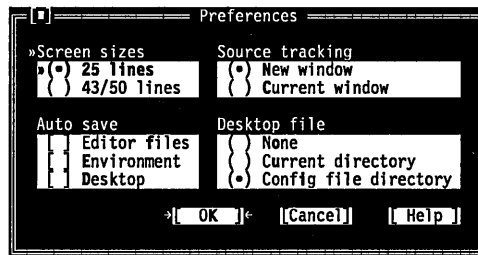
Environment

The **Options | Environment** command lets you make environment-wide settings. This command opens a menu that lets you choose settings from **P**references, **E**ditor, and **M**ouse.

Preferences

Here's what the Preferences dialog box looks like:

Figure 7.17
The Preferences dialog box



Screen Sizes
 25 lines
 43/50 lines

- The **Screen Sizes** radio buttons let you specify whether your IDE screen is displayed in 25 or 43/50 lines. One or both of these buttons will be available, depending on the type of video adapter in your PC.

When set to 25 lines (the default), Turbo Pascal uses 25 lines and 80 columns. This is the only screen size available to systems with a monochrome display or Color Graphics Adapter (CGA).

If your PC has EGA or VGA, you can set this option to 43/50 lines. The IDE is displayed in 43 lines by 80 columns if you have an EGA., or 50 lines by 80 columns if you have a VGA.

```
Source Tracking
{ } New window
(•) Current window
```

- When stepping source or locating an error position in your source code, the IDE opens a new window whenever it encounters a file that is not already loaded. Selecting **Current Window** causes the IDE to replace the contents of the topmost Edit window with the new file instead of opening a new Edit window.

```
Auto Save
[ ] Editor Files
[ ] Environment
[ ] Desktop
```

- If Editor **Files** is checked in the Auto Save options, and if the file has been modified since the last time you saved it, Turbo Pascal automatically saves the source file in the Edit window whenever you choose the **Run | Run** (or any debug/run command) or **File | DOS Shell** command.

When the **Environment** option is checked, all the settings you made in this session will be saved automatically into a TURBO.TP configuration file when you exit Turbo Pascal.

When **Desktop** is checked, Turbo Pascal controls whether your desktop (in the file TURBO.DSK) is saved on exit and whether it's restored when you return to Turbo Pascal. Your desktop information will not be saved unless a .TP file is created (automatically or manually by selecting the **Options | Save Options** command) and the **Desktop** radio button is set to something other than **None**.

```
Desktop Files
{ } None
{ } Current Directory
(•) Config file directory
```

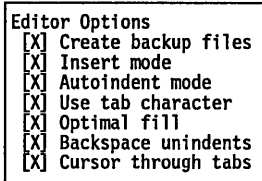
- If you want the IDE to use a desktop file to save and restore your desktop from one programming session to another, select either the **Current Directory** or **Config File Directory** radio button. When the IDE saves a TURBO.TP configuration file, it will also create a TURBO.DSK file that contains edit window information, the positions of all windows on the desktop, history lists, breakpoint locations, and other state information.

All this can be saved and restored automatically by enabling both the **Environment** and **Desktop** options in the Auto Save group. Alternatively, you can create a TURBO.TP manually by using the **Options | Save Options** dialog box.

When you next load TURBO.EXE, it will look for TURBO.TP and TURBO.DSK in the current directory. When located, they are loaded and the previous session's configuration and

desktop states are restored. If no TURBO.TP is found in the current directory, the IDE also will look for it in the directory that contains TURBO.EXE itself.

Editor If you choose **Editor** from the **Environment** menu, the **Editor Options** dialog box has several check boxes that control how Turbo Pascal handles text in Edit windows.

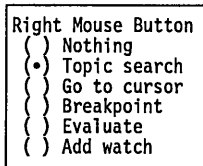


- When **Create Backup Files** is checked (the default), Turbo Pascal automatically creates a backup of the source file in the Edit window when you choose **File | Save** and gives the backup file the extension **.BAK**.
- When **Insert Mode** is not checked, any text you type into Edit windows overwrites existing text. When the option is checked, text you type is inserted (pushed to the right). Pressing *Ins* toggles Insert mode when you're working in an Edit window.
- When **Autoindent Mode** is checked, pressing *Enter* in an Edit window positions the cursor under the first nonblank character in the preceding nonblank line. This can be a great aid in keeping your program code more readable.
- When **Use Tab Character** is checked, Turbo Pascal inserts a true tab character (ASCII 9) when you press *Tab*. When this option is not checked, Turbo Pascal replaces tabs with spaces, the number of which is determined by the **Tab Size** setting.
- When you check **Optimal Fill**, Turbo Pascal begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary. This produces lines with fewer characters than when **Optimal Fill** is not checked.
- When **Backspace Unindents** is checked (the default) and the cursor is on a blank line or the first non-blank character of a line, the *Backspace* key aligns (outdents) the line to the previous indentation level.
- When you check **Cursor Through Tabs**, the arrow keys will move the cursor to the middle of tabs; otherwise the cursor jumps several columns when cursoring over a tab.
- If you check **Use Tab Character** in this dialog box and press *Tab*, Turbo Pascal inserts a tab character in the file and the cursor moves to the next tab stop. The **Tab Size** input box allows you to dictate how many characters to move for each tab stop. Legal values are 2 through 16; the default is 8.

Tab Size 8

To change the way tabs are displayed in a file, just change the tab size value to the size you prefer. Turbo Pascal redisplayes all tabs in that file in the size you chose. You can save this new tab size in your configuration file by choosing **Options | Save Options**.

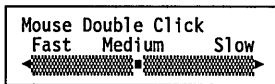
Mouse When you choose **Mouse** from the **Environment** menu, the **Mouse Options** dialog box is displayed, which contains all the settings for your mouse.



The **Right Mouse Button** radio buttons determine the effect of pressing the right button of the mouse (or the left button, if the **Reverse mouse buttons** option is checked). **Topic Search** is the default.

Here's a list of what the right button would do if you choose something other than **Nothing**:

Topic Search	Same as Help Topic Search
Go to Cursor	Same as Run Go To Cursor
Breakpoint	Same as Debug Toggle Breakpoint
Evaluate	Same as Debug Evaluate
Add Watch	Same as Debug Watches Add Watch



In the **Mouse Double Click** box, you can change the slider control bar to adjust the double-click speed of your mouse by using the arrow keys.



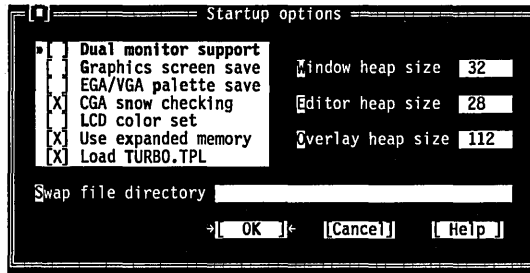
Moving the scroll box closer to **Fast** means Turbo Pascal requires a shorter time between clicks to recognize a double click. Moving the scroll box closer to **Slow** means Turbo Pascal will still recognize a double click even if you wait longer between clicks.

Reverse Mouse Buttons

When **Reverse Mouse Buttons** is checked, the active button on your mouse is the rightmost one instead of the leftmost. Note, however, that the buttons won't actually be switched until you choose the **OK** button.

Startup Choosing **Startup** from the **Environment** menu lets you select settings for the integrated environment.

Figure 7.18
The Startup Options dialog box

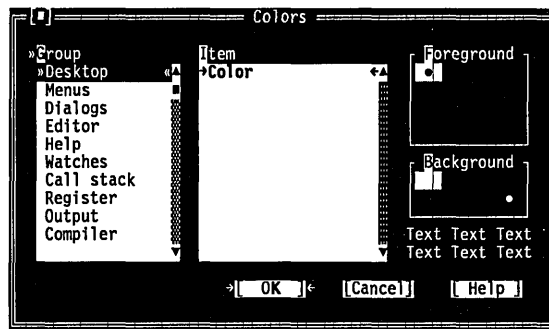


All of these options correspond to the command-line options mentioned at the beginning of this chapter (see page 174).

The changes that you make here are written directly to TURBO.EXE and don't take affect until the next time you load the IDE.

Colors
Figure 7.19
The Colors dialog box

Use the Colors dialog box to customize the IDE for your use.



The **Group** list contains the names of the different regions of the IDE that can be customized. When you select a group, the **Item** list box will contain the names of the different views in that region. On color and black and white systems, you can modify the foreground and background colors by using your mouse or cursor keys to change the palette. On all systems, the text in the lower right corner of the dialog box reflects the current settings. Changes do not take affect on the desktop until you close the dialog box (by selecting OK).

Save Options

Save your desktop state (TURBO.DSK) by setting Desktop to Current Directory or Config File Directory in the Preferences dialog box.

The **Options | Save Options** command brings up a dialog box that lets you save settings that you've made in both the **Find** and **Replace** dialog boxes (off the **Search** menu), the **Destination** and **Primary File** options (off the **Compile** menu) and all the settings under the **Options** menu. All options and the editor command table are stored in TURBO.TP; history lists, your desktop state, and breakpoint locations are stored in TURBO.DSK.

If it doesn't find the files, Turbo Pascal looks for these files' directory where TURBO.EXE is run from.

Retrieve Options

The **Options | Retrieve Options** command brings up a dialog box that lets you retrieve the settings that you've made in both the **Find** and **Replace** dialog boxes (off the **Search** menu), the **Destination** and **Primary File** options (off the **Compile** menu) and all the settings under the **Options** menu. If the **Desktop** file (.DSK) is set to either **Current Directory** or **Config File Directory**, TURBO.DSK will also be loaded.

Window menu

Refer to page 11 for information on window elements and how to use them.

The **Window** menu contains window management commands. Most of the windows you open from this menu have all the standard window elements like scroll bars, a close box, and zoom boxes.

The commands **Tile** and **Cascade** will always rearrange Edit windows in the region above a **Watch**, **Output**, or **Call Stack** window. If none of these windows are open, **Tile** and **Cascade** will use the entire desktop.

At the bottom of the **Window** menu, the **Window | List** command appears. Choose this command for a list of all open windows.

Size/Move

Ctrl **F5**

Choose **Window | Size/Move** to change the size or position of the active window. When you choose this command, the active window moves in response to the arrow keys. When the window is where you want it, press *Enter*. You can also move a window by dragging its title bar.

If you press *Shift* while you use the arrow keys, you can change the size of the window. When the window is the size you want it, press *Enter*. If a window has a resize corner, you can drag that corner or any other corner to resize it.

Zoom

F5

Choose **Window | Zoom** to resize the active window to the maximum size. If the window is already zoomed to the maximum, you can choose this command again to restore it to its previous size. You can also double-click anywhere on the top line (except where an icon appears) of a window to zoom or unzoom it.

Tile

Choose **Window | Tile** to view equally all your open Edit windows. Tiling makes all your open Edit windows a similar size and lays them out one next to the other so none overlap.

Cascade

Choose **Window | Cascade** to stack all open Edit windows. Cascade only lets you fully view the active window; only file names and window numbers are visible for the other windows.

Next

F6

Choose **Window | Next** to make the next window active, which makes it the topmost open window.

Previous

Shift **F6**

Choose **Window | Previous** to make the previous window active (the window last opened before the currently active one).

Close



Choose **Window | Close** to close the active window. You can also click the close box in the upper left corner to close a window.

Watch

Choose **Window | Watch** to open the Watch window and make it active. The Watch window displays expressions and their changing values so you can keep an eye on how your program evaluates key values.

You use the commands in the **Debug | Watches** pop-up menu to add or remove watches from this window. Refer to the section on this menu for information on how to use the Watch window (page 196).

To close the window, click its close box or choose **Window | Close**.

Register

Choose **Window | Register** to open the Register window and make it active. The Register window displays CPU registers and is especially useful when debugging inline assembler. To close the window, click its close box or choose **Window | Close**.

Output

Choose **Window | Output** to open the Output window and make it active. The Output window displays text from any DOS command-line text and any text generated from your program (no graphics).

The Output window is handy while debugging because you can view your source code, variables, and output all at once. This is especially useful when you've set the **Options | Environment** dialog box to a 43-line display and you are running a standard 25-line mode program. In that case, you can see almost all of the program output and still have plenty of lines to view your source code and variables.

If you would rather see your program's text on the full screen—or if your program generates graphics—choose the **Window | User Screen** command instead (*Alt-F5*).

To close the window, click its close box or choose **Window | Close**.

Call Stack

Ctrl **F3**

The **Window | Call Stack** command opens a window that shows the sequence of procedures your program called to reach the procedure currently running. At the bottom of the stack is *Program* (or your program name); at the top is the procedure that's currently running.

Each entry on the stack displays the name of the procedure called and the values of the parameters passed to it.

Initially the entry at the top of the stack is highlighted. To display the current line of any other procedure on the call stack, select that procedure's name and press *Enter*. The cursor moves to the line containing the call to the procedure next above it on the stack. The call stack will stay on the desktop until you close it.

User Screen

Alt **F5**

Choose **Window | User Screen** to view your program's full-screen output. If you would rather see your program output in a Turbo Pascal window, choose the **Window | Output** command instead. Clicking or pressing any key returns you to the integrated environment.

List

Choose **Window | List** to get a list of all the windows you've opened; the list contains the names of all files that are currently open. When you make a selection from the list, Turbo Pascal brings the window to the front and makes it active.

Alt **O**

Press *Alt-O* to pop up a complete list of all open windows. For a full rundown of how to manage windows, see page 13.

Help menu

The **Help** menu gives you access to online help in a special window. There is help information on virtually all aspects of the IDE and Turbo Pascal. (Also, one-line menu and dialog box hints appear on the status line whenever you select a command.)

To open the Help window, do one of these actions:

- F1 ■ Press *F1* at any time (including from any dialog box or when any menu command is selected).
- When an Edit window is active and the cursor is positioned on a word, press *Ctrl-F1* to get language help.
- Click Help whenever it appears on the status line or in a dialog box.

To close the Help window, press *Esc*, click the close box, or choose **Window | Close**. You can keep the Help window onscreen while you work in another window unless you opened the Help window from a dialog box or pressed *F1* when a menu command was selected. (If you press *F6* or click on another window while you're in Help, the Help window remains onscreen.)

When getting help in a dialog box or menu, you cannot resize the window or copy to the clipboard. In this instance, Tab takes you to dialog box controls, not the next keyword.

Help screens often contain *keywords* (highlighted text) that you can choose to get more information. Press *Tab* to move to any keyword; press *Enter* to get more detailed help. (As an alternative, move the cursor to the highlighted keyword and press *Enter*. With a mouse, you can double-click any keyword to open the help text for that item.

You can also cursor around the Help screen and press *Ctrl-F1* on *any* word to get help. If the word is not found, an incremental search is done in the index and the closest match displayed.

When the Help window is active, you can copy from the window and paste that text into an Edit window. You do this just the same as you would in an Edit window: Select the text first (using *Shift→*, Left arrow, Up arrow, Down arrow), choose **Edit | Copy**, move to an Edit window, then choose **Edit | Paste**.

To select text in the Help window, drag across the desired text or, when positioned at the start of the block, press *Shift→*, *←*, *↑*, *↓* to mark a block.

You can also copy preselected program examples from help screens by choosing the **Edit | Copy Example** command.

Contents

The **Help | Contents** command opens the Help window with the main table of contents displayed. From this window, you can branch to any other part of the help system.

- F1** You can get help on Help by pressing *F1* when the Help window is active. You can also reach this screen by clicking on the status line.

Index

The **Help | Index** command opens a dialog box displaying a full list of help keywords (the special highlighted text in help screens that let you quickly move to a related screen).

You can scroll the list or you can incrementally search it by pressing letters from the keyboard. For example, to see what's available under "printing," you can type *p r i*. When you type *p*, the cursor jumps to the first keyword that starts with *p*. When you then type *r*, the cursor then moves to the first keyword that starts with *pr*. When you then type *i*, the cursor moves to the first keyword that starts with *pri*, and so on.

When you find a keyword that interests you, choose it by cursoring to it and pressing *Enter*. (You can also double-click it.)

Topic Search

Ctrl F1

The **Help | Topic Search** command displays language help on the currently selected item.

To get language help, position the cursor on an item in an Edit window and choose **Topic Search**. You can get help on things like procedure names (*Writeln*, for example), reserved words, and so on. If an item is not in the help system, the help index displays the closest match.

Previous Topic

Alt F1

The **Help | Previous Topic** command opens the Help window and redisplay the text you last viewed.

Turbo Pascal lets you back up through 20 previous help screens. You can also click on the status line to view the last help screen displayed.

Help on Help

F1

The **Help | Help on Help** command opens up a text screen that explains how to use the Turbo Pascal help system. If you're already in help, you can bring up this screen by pressing *F1*.

The editor from A to Z

You should read this chapter even if you are familiar with the editor in other Turbo products. Turbo Pascal's new IDE includes improvements to the editor. Context-sensitive help is always just a keystroke away (F1).

This chapter is a reference to Turbo Pascal's full range of editing commands. Table 8.1 contains a list of all of the editor commands; the tables and text that follow it cover those aspects of the editor that need further explanation.

Remember, this chapter is concerned *just* with the editor. For a tutorial about the editor and the IDE, refer to Chapter 1; for an in-depth discussion of the whole Turbo Pascal integrated environment, refer to Chapter 7.

The new and the old

The new Turbo Pascal IDE still lets you use Borland's familiar hot key combinations to move around your file, insert, copy, and delete text, and search and replace. However, it also provides you with two brand-new menus on the menu bar, the Edit menu and the Search menu. In addition, Turbo Pascal now supports use of a mouse for many of the cursor movement and block-marking commands.

The Edit menu contains commands for cutting, copying, and pasting in a file, copying examples from Help to an Edit window, and viewing the Clipboard. When you first start Turbo Pascal, an Edit window is already active. To open other Edit windows, go to the File menu and choose **O**pen. From an Edit window, you still press *F10* to get to the menu bar; to return to the Edit window,



keep pressing *Esc* until you are out of the menus. If you have a mouse, you can also just click anywhere in the Edit window.

As always, you enter text pretty much as if you were using a typewriter. To end a line, press *Enter*. When you've entered enough lines to fill the screen, the top line scrolls off the screen. Don't worry—it isn't lost; you can move back and forth in your text with the scrolling commands that are described later.

The editor has a restore facility that lets you take back changes to the last line modified. This command (**Edit | Restore line**) is described on page 224 in the section titled "Other editing commands."

Editor reference

Table 8.1 summarizes all editor commands.

The editor is much more powerful than a quick tutorial can show. In addition to the menu choices, it uses approximately 50 commands to move the cursor around, page through text, find and replace strings, and so on. These commands can be grouped into four main categories:

- Cursor movement
- Insert and delete operations
- Block operations
- Miscellaneous editing operations


Most of these commands need no explanation. Those that do are described in the text following Table 8.1.

Table 8.1
Full summary of editor
commands

Movement	Command
<i>Cursor movement commands</i>	
<i>Basic cursor movement</i>	
Character left	←
Character right	→
Word left	Ctrl ←
Word right	Ctrl →
Line up	↑
Line down	↓
Scroll up one line	Ctrl-W
Scroll down one line	Ctrl-Z
Page up	PgUp
Page down	PgDn

*A word is defined as a sequence of characters separated by one of the following: space <> , ; . () ^ ` * + - / \$ # _ = | ~ ? ! " % & ` ; @ \ , and all control and graphic characters.*

Table 8.1: Full summary of editor commands (continued)

 Many of the commands in this table can also be performed with the mouse. See Chapter 7.

Movement	Command
<i>Long distance</i>	
Beginning of line	<i>Home</i>
End of line	<i>End</i>
Top of window	<i>Ctrl Home</i>
Bottom of window	<i>Ctrl End</i>
Beginning of file	<i>Ctrl PgUp</i>
End of file	<i>Ctrl PgDn</i>
Beginning of block	<i>Ctrl-Q B</i>
End of block	<i>Ctrl-Q K</i>
Last cursor position	<i>Ctrl-Q P</i>
Insert and delete commands	
Insert mode on/off	Options Environment Editor Insert mode or <i>Ins</i>
Delete character left of cursor	<i>Backspace</i>
Delete character at cursor	<i>Del</i>
Delete word right	<i>Ctrl-T</i>
Insert line	<i>Ctrl-N</i>
Delete line	<i>Ctrl-Y</i>
Delete to end of line	<i>Ctrl-Q Y</i>
Block commands	
Mark block	<i>Shift ↓, ↑, →, ←, Ctrl-K B, Ctrl-K K</i>
Mark single word	<i>Ctrl-K T</i>
Copy block	Edit Copy, Edit Paste or <i>Ctrl-Ins, Shift-Ins</i>
Move block	Edit Cut, Edit Paste or <i>Shift-Del, Shift-Ins</i>
Delete block	Edit Clear or Ctrl-Del
Read block from disk	<i>Ctrl-K R</i>
Write block to disk	<i>Ctrl-K W</i>
Hide/display block	<i>Ctrl-K H</i>
Print block	File Print or Ctrl-K P
Indent block	<i>Ctrl-K I</i>
Unindent block	<i>Ctrl-K U</i>
Other editing commands	
Autoindent on/off	Options Environment Editor Autoindent mode*
Control character prefix**	<i>Ctrl-P</i>
Find place marker	<i>Ctrl-Q n***</i>
Go to menu bar	<i>F10</i>
New file	File New
Open file	File Open (F3)
Optimal fill mode on/off	Options Environment Editor Optimal fill*
Pair matching	<i>Ctrl-Q [and Ctrl-Q]</i>
Print file	File Print
Quit IDE	File Quit (Alt-X)

Table 8.1: Full summary of editor commands (continued)

Movement	Command
Repeat last search	Search Search Again or <i>Ctrl-L</i>
Restore error message	<i>Ctrl-Q W</i>
Restore line	Edit Restore Line or <i>Ctrl-Q L</i>
Return to editor from menus	<i>Esc</i>
Save	File Save (<i>F2</i>)
Search	Search Find or <i>Ctrl-Q F</i>
Search and replace	Search Replace or <i>Ctrl-Q A</i>
Set place marker	<i>Ctrl-K n***</i>
Tab	<i>Tab</i>
Tab mode	Options Environment Editor Use tab characters*
Unindent mode	Options Environment Editor Backspace unindents*

*This command opens the Editor Options dialog box, in which you can set the appropriate check box or radio buttons.

**Enter control characters by first pressing *Ctrl-P*, then pressing the desired control character. Depending on your screen setup, control characters appear as low-intensity or inverse capital letters.

****n* represents a number from 0 to 9.

Jumping around

There are three cursor movement commands that need further explanation: *Ctrl-Q B* (Beginning of block), *Ctrl-Q K* (End of block), and *Ctrl-Q P* (Last cursor position).

Ctrl-Q B and *Ctrl-Q K* move the cursor to the block-begin or block-end marker. Both these commands work even if the block is not displayed (see "Hide/display block" in Table 8.2). *Ctrl-Q B* works even if the block-end marker is not set, and *Ctrl-Q K* works even if the block-begin marker is not set.

Ctrl-Q P moves to the last position of the cursor before the last command. This command is particularly useful after a search or search-and-replace operation has been executed, and you'd like to return to where you were at before you ran the search.

Block commands

A block of text is any amount of text, from a single character to hundreds of lines, that has been surrounded with special block-marker characters. There can be only one block in a window at a time. A block is marked by placing a block-begin marker on the first character and a block-end marker after the last character of

the desired portion of the text. Once marked, the block can be copied, moved, deleted, printed, or written to a file.

Table 8.2: Block commands in depth

Movement	Command(s)	Function
Mark block	<i>Shift</i> ↓, ↑, →, ←	Marks (highlights) a block as the cursor is moved. Marked text is displayed in a different intensity.
Mark single	<i>Ctrl-K T</i>	Marks a single word as a block. If the cursor is placed within a word, that word will be marked. If it is not within a word, then the word to the left of the cursor will be marked.
Copy block	<i>Edit Copy, Ctrl-Ins</i> <i>Edit Paste, Shift-Ins</i>	Copies a previously marked block to the Clipboard and pastes it to the current cursor position. The original block is unchanged, and the block markers are placed around the new copy of the block. If no block is marked or the cursor is within the marked block, nothing happens.
Move block	<i>Edit Cut, Shift-Del</i> <i>Edit Paste, Shift-Ins</i>	Moves a previously marked block from its original position to the Clipboard and pastes it to the cursor position. The block disappears from its original position; the markers remain around the block at its new position. If no block is marked, nothing happens.
Delete block	<i>Edit Clear, Ctrl-Del</i> <i>Ctrl-K Y</i>	Deletes a previously marked block. No provision exists to restore a deleted block, so be careful with this command.
Write block to disk	<i>Ctrl-K W</i>	Writes a previously marked block to a file. The block is left unchanged, and the markers remain in place. 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 .PAS). If you prefer to use a file name without an extension, append a period to the end of its name. Note: You can use wildcards to select a file to overwrite; a directory is displayed. If the file specified already exists, a warning is issued before the existing file is overwritten. If no block is marked, nothing happens.
Read block from disk	<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 marked 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.
Hide/display block	<i>Ctrl-K H</i>	Causes the visual marking of a block to be alternately switched off and on. The block manipulation commands (copy, move, delete, print, and write to a file) work only when the block is displayed. Block-related cursor movements (jump to beginning/end of block) work whether the block is hidden or displayed.
Print block Print	<i>Ctrl-K P</i> <i>File Print</i>	Sends the marked block in the active Edit window to the printer. Sends the entire file in the active Edit window to the printer.

Other editing commands

The next table describes certain editing commands in more detail. The table is arranged alphabetically by command name.

Table 8.3: Other editor commands in depth

Movement	Command(s)	Function
Autoindent	Options Environment Editor	Opens the Editor options dialog box, in which you can toggle the Autoindent mode check box. Provides automatic indenting of successive lines. When Autoindent is active, the indentation of the current line is repeated on each following line; that is, when you press <i>Enter</i> , the cursor does not return to column one but to the starting column of the preceding non-empty line. When you want to change the indentation, use the <i>Spacebar</i> and <i>←</i> key to select the new column. Autoindent is on by default.
Find place marker	<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.
New file	File New	Opens a new window.
Open file	File Open (F3)	Lets you load an existing file into an Edit window.
Quit edit	File Quit (Alt-X)	Quits Turbo Pascal. You are asked whether you want to save the file to disk.
Restore line	Edit Restore Line	Lets you undo changes made to the last line worked on. The line is restored to its original state regardless of any changes you have made. This works only on the last modified or deleted line.
Save file	File Save (F2)	Saves the file and returns to the editor.
Set place	<i>Ctrl-K n</i>	Mark up to ten places in text by pressing <i>Ctrl-K</i> , followed by a single marker digit (0 to 9). After marking your location, you can work elsewhere in the file and then easily return to your marked location by using the <i>Ctrl-Q N</i> command (being sure to use the same marker number). You can have ten places marked in each window.
Tab	<i>Tab</i>	Tabs default to eight columns apart in the Turbo Pascal editor.
Tab mode	Options Environment Editor	Opens the Editor options dialog box, in which you can set the Use tab character check box. When the option is on, you can insert tab characters (ASCII character 8); when it's off, the tab is automatically inserted as the correct number of spaces.

Search and replace

The search string is also called the target string.

Searching and searching again

The **Search | Find** and **Search | Replace** commands let you search for (and optionally replace) strings.

The search string can contain any characters, including control characters. You can enter control characters with the *Ctrl-P* prefix. For example, search for a *Ctrl-T* by holding down the *Ctrl* key as you press *P* and then *T*. You can include a line break in a search string by specifying *Ctrl-M* (carriage return). (For searching regular expressions, take a look at the online file UTILS.DOC.)

The following sections list the steps for performing these operations.

1. Choose **Search | Find**. This opens the Find dialog box.
2. Type the string you are looking for into the Text to Find input box.
3. You can also set various search options:
 - The Direction radio buttons control whether you do a forward or backward search.
 - The Scope radio buttons control how much of the file you search.
 - The Origin radio buttons control where the search begins.
 - The Options check boxes determine whether the search will be case sensitive for whole words only, and for regular expressions.



Use *Tab* or your mouse to cycle through the options. Use \uparrow and \downarrow to set the radio buttons and *Space* to toggle the check boxes.

4. Finally, choose the OK button to carry out the search or the Cancel button to cancel. Turbo Pascal performs the operation.
5. If you want to search for the same item repeatedly, use **Search | Search Again**.

Search and replace

1. Choose **Search | Replace**. This opens the Replace dialog box.
2. Type the string you are looking for into the Text to Find input box.
3. Press *Tab* or use your mouse to move to the New text input box. Type in the replacement string.

4. You can then set the same search options as in the Find dialog box.
5. Finally, choose OK or Change all to begin the search, or choose Cancel to cancel. Turbo Pascal performs the operation. Choosing Change all will replace every occurrence found.
6. If you want to stop the operation, press *Esc* at any point when the search has paused.

Pair matching

There you are, debugging your source file that is full of functions, parenthesized expressions, nested comments, and a whole slew of other constructs that use delimiter pairs. In fact, your file is riddled with

- braces: { and }
- parentheses: (and)
- brackets: [and]
- double quotes: "
- single quotes: '

Finding the match to a particular paired construct can be tricky. Suppose you have a complicated expression with a number of nested expressions, and you want to make sure all the parentheses are properly balanced. Or say you're at the beginning of a function that stretches over several screens, and you want to jump to the end of that function. With Turbo Pascal's handy pair-matching commands, the solution is at your fingertips. Here's what you do:

1. Place the cursor on the delimiter in question.
2. To locate the mate to this selected delimiter, simply press *Ctrl-Q*.
3. The editor immediately moves the cursor to the delimiter that matches the one you selected. If it moves to the one you had intended to be the mate, you know that the intervening code contains no unmatched delimiters of that type. If it moves to the wrong delimiter, you know there's trouble; now all you need to do is track down the source of the problem.

We've told you the basics of Turbo Pascal's "Match Pair" commands; now you need some details about what you can and can't

do with these commands, and notes about a few subtleties to keep in mind. This section covers the following points:

- There are actually two match pair editing commands: one for forward matching (*Ctrl-Q [*) and the other for backward matching (*Ctrl-Q]*).
- If there is no mate for the delimiter you've selected, the editor doesn't move the cursor.

Directional and nondirectional matching

Opening braces and brackets and closing braces and parentheses are directional; the editor knows which way to search for the mate, so it doesn't matter which match pair command you give.

Double and single quotes are not directional. You must specify the correct match pair command.

Two match pair commands are necessary because some delimiters are *nondirectional*.

For example, suppose you tell the editor to find the match for an opening brace ({) or an opening bracket ([). The editor knows the matching delimiter can't be located *before* the one you've selected, so it searches forward for a match. If you tell the editor to find the mate to a closing brace (}) or a closing parenthesis ()), it knows that the mate can't be located *after* the selected delimiter, so it automatically searches backward for a match.

However, if you tell the editor to find the match for a double quote (") or a single quote ('), it doesn't know automatically which way to go. You must specify the search direction by giving the correct match pair command. If you give the command *Ctrl-Q Ctrl-]*, the editor searches forward for the match; if you give the command *Ctrl-Q Ctrl-[*, it searches backward for the match.

The following table summarizes the delimiter pairs, whether they imply search direction, and whether they are nestable:

Table 8.4
Delimiter pairs

Nestable delimiters are explained after this table.

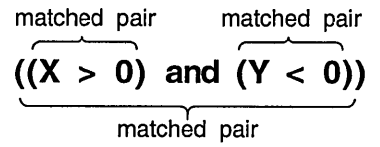
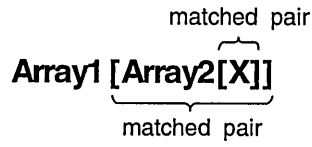
Delimiter pair	Direction implied?	Are they nestable?
{ }	Yes	Yes
()	Yes	Yes
[]	Yes	Yes
" "	No	No
' '	No	No

Nestable delimiters

Nestable means that, when the editor is searching for the mate to a directional delimiter, it keeps track of how many delimiter levels it enters and exits during the search.

This is best illustrated with some examples:

Figure 8.1
Search for match to square
bracket or parenthesis



The command-line compiler

TPCX is available only in the Professional package.

Turbo Pascal's command-line compiler (TPC.EXE) lets you invoke all the functions of the IDE compiler (TURBO.EXE) from the DOS command line. You can run the command-line compiler in either real or protected mode; both TPC and TPCX generate real mode programs only. The protected mode compiler (TPCX.EXE) lets you use extended memory to compile very large programs; it uses the same options as TPC.EXE.

You run TPC.EXE from the DOS prompt using a command line with the following syntax:

TPC [*options*] *files*

options are zero or more optional parameters that provide additional information to the compiler. *files* are the names of the sources file to compile. If you type TPC alone, it displays a help screen of command-line options and syntax.

If *files* does not have an extension, TPC assumes .PAS. If you don't want the file you're compiling to have an extension, you must append a period (.) to the end of *files*. If the source text contained in *files* is a program, TPC creates an executable file named FILENAME.EXE. If *files* contains a unit, TPC creates a Turbo Pascal unit file named FILENAME.TPU.

You can get help at the command line using THELP; see THELP.DOC in ONLINE.ZIP on your disk.

You can specify a number of options for TPC. An option consists of a slash (/) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such

as a number, a symbol, or a directory name. Options can be given in any order and can come before and/or after the file name.

Compiler options

The IDE allows you to set various options through the menus; TPC gives you access to most of these same options using the slash (/) command. You can also precede options with a hyphen (-) instead of a slash (/), but those options that start with a hyphen must be separated by blanks. For example, the following two command lines are equivalent and legal:

```
TPC -IC:\TP\TVISION -DDEBUG SORTNAME -$S- -$F+
TPC /IC:\TP\TVISION/DDEBUG SORTNAME /$S-/ $F+
```

The first uses hyphens with at least one blank separating options; the second uses slashes and no separation is needed.

Table 9.1 lists all the command-line options and gives their integrated environment equivalents. In some cases, a single command-line option corresponds to two or three menu commands.

Table 9.1
Command-line options

Command Line	Menu Command	Setting
/\$A+	Options Compiler Word Align Data	Word
/\$A-	O C Word Align Data	Byte
/\$B+	O C Complete Boolean Eval	Complete
/\$B-	O C Complete Boolean Eval	Short
		Circuit
/\$D+	O C Debug Information	On
/\$D-	O C Debug Information	Off
/\$E+	O C Emulation	On
/\$E-	O C Emulation	Off
/\$F+	O C Force Far Calls	On
/\$F-	O C Force Far Calls	Off
/\$G+	O C 286 Instructions	On
/\$G-	O C 286 Instructions	Off
/\$I+	O C I/O-Checking	On
/\$I-	O C I/O-Checking	Off
/\$L+	O C Local Symbols	On
/\$L-	O C Local Symbols	Off
/\$M _{SS,min,max}	O Memory Sizes	
/\$N+	O C Numeric Processing	8087/80287
/\$N-	O C Numeric Processing	
/\$O+	O C Overlays Allowed	On
/\$O-	O C Overlays Allowed	Off
/\$R+	O C Range Checking	On

Table 9.1: Command-line options (continued)

Command Line	Menu Command	Setting
/SR-	O C Range Checking	Off
/S+	O C Stack Checking	On
/S-	O C Stack Checking	Off
/V+	O C Strict Var-string	On
/V-	O C Strict Var-string	Off
/X+	O C Extended Syntax	On
/X-	O C Extended Syntax	Off
/B	Compile Build	
/Ddefines	Options Compiler Conditional Defines	
/Epath	Options Directories EXE & TPU Directory	
/Fseg:ofs	Search Find Error	
/GS	Options Linker Map File	Segments
/GP	Options Linker Map File	Public
/GD	Options Linker Map File	Detailed
/path	Options Directories Include Directories	
/L	Options Linker Link Buffer	Disk
/M	Compile Make	
/Opath	Options Directories Object Directories	
/Q	(none)	
/Tpath	Options Directories Turbo Directory	
/Upath	Options Directories Unit Directories	
/V	Debugger Standalone	On

Compiler directive options

Turbo Pascal supports several compiler directives, all of which are described in Chapter 21 of the *Programmer's Guide*, "Compiler directives." When embedded in the source code, these directives take one of the following forms:

```
{ $directive+ }
{ $directive- }
{ $directive info }
```

The **/S** and **/D** command-line options allow you to change the default states of most compiler directives. Using **/S** and **/D** on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The switch directive option

The **/S** option allows you to change the default state of the following switch directives: **\$A**, **\$B**, **\$D**, **\$E**, **\$F**, **\$G**, **\$I**, **\$L**, **\$N**, **\$O**, **\$R**, **\$S**, **\$V**, and **\$X**. The syntax of a switch directive option is **/S**

followed by the directive letter, followed by a plus (+) or a minus (-). For example,

```
TPC MYSTUFF /$R-
```

would compile MYSTUFF.PAS with range checking turned off, while

```
TPC MYSTUFF /$R+
```

would compile it with range-checking turned on. Note that if a **/\$R+** or **/\$R-** compiler directive appears in the source text, it overrides the **/\$R** command-line option.

You can repeat the **/\$** option in order to specify multiple compiler directives:

```
TPC MYSTUFF /$R-/$I-/$V-/$F+
```

Alternately, TPC allows you to write a list of directives (except for **\$M**), separated by commas:

```
TPC MYSTUFF /$R-,I-,V-,F+
```

Note that only one dollar sign (\$) is needed.

In addition to changing switch directives, **/\$** also allows you to specify a program's memory allocation parameters, using the following format:

```
/$MSTACK,HEAPMIN,HEAPMAX
```

where *stack* is the stack size, *heapmin* is the minimum heap size, and *heapmax* is the maximum heap size. All three values are in bytes, and each is a decimal number unless it is preceded by a dollar sign (\$), in which case it is assumed to be hexadecimal. So, for example, the following command lines are equivalent:

```
TPC MYSTUFF /$M16384,0,655360  
TPC MYSTUFF /$M$4000,$0,$A0000
```

Note that, because of its format, you cannot use the **\$M** option in a list of directives separated by commas.

The conditional defines option

The **/D** option lets you define conditional symbols, corresponding to the **/\$DEFINE *symbol*** compiler directive or the **O/C | Conditional Defines** option in the IDE. The **/D** option must be followed by one or more conditional symbols, separated by semicolons (;). For example, the following command line

```
TPC MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, *iocheck*, *debug*, and *list*, for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{DEFINE IOCHECK}  
{DEFINE DEBUG}  
{DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple */D* directives, you can concatenate the symbol lists are concatenated. Thus

```
TPC MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

is equivalent to the first example.

Compiler mode options

A few options affect how the compiler itself functions. These are */M* (Make), */B* (Build), */F* (Find Error), */L* (Link Buffer) and */Q* (Quiet). As with the other options, you can use the hyphen format (remember to separate the options with at least one blank).

The make (/M) option

TPC has a built-in MAKE utility to aid in project maintenance. The */M* option instructs TPC to check all units upon which the file being compiled depends.

A unit will be recompiled if

- the source file for that unit has been modified since the .TPU file was created, or
- any file included with the *\$I* directive, or any .OBJ file linked in by the *\$L* directive, is newer than the unit's .TPU file, or
- the interface section of a unit referenced in a *uses* statement has changed



Units in TURBO.TPL are excluded from this process.

If you were applying this option to the previous example, the command would be

```
TPC MYSTUFF /M
```

The build all (/B) option

*You can't use /M and /B at
the same time.*

Instead of relying on the **/M** option to determine what needs to be updated, you can tell TPC to update *all* units upon which your program depends using the **/B** option. This is the same as **Compile | Build**.

If you were using this option in the previous example, the command would be

```
TPC MYSTUFF /B
```

The find error option

*This is the same as Find Error
on the Search menu.*

When a program terminates due to a run-time error, it displays an error code and the address (*seg:ofs*) at which the error occurred. By specifying that address in a **/Fseg:ofs** option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the **\$D** compiler directive).

Suppose you have a file called TEST.PAS that contains the following program:

```
program Test;
var
  i : integer;
begin
  i := 0;
  i := i div i;           { Force a divide by zero error }
end.
```

First, compile this program using the command-line compiler:

```
TPC TEST
```

If you do a **DIR TEST.***, DOS lists two files: TEST.PAS, your source code, and TEST.EXE, the executable file.

Now, run TEST and you'll get a run-time error:

```
C:\>TEST
Run-time error 200 at 0000:0018
```

Notice that you're given an error code (200) and the address (0000:0018 in hex) of the instruction pointer (CS:IP) where the error occurred. To figure out which line in your source caused the

error, simply invoke the compiler, use **/F** and specify the segment and offset as reported in the error message:

```
C:\>TPC TEST /F0:18
Turbo Pascal Version 6.0 Copyright (c) 1983,90 Borland
International
TEST.PAS(7)
TEST.PAS(6): Target address found.
  i := i div i;
  ^
```



In order for TPC to find the run-time error with **/F**, you must compile the program with all the same command-line parameters you used the first time you compiled it.

The compiler now gives you the file name and line number, and points to the offending line number and text in your source code.



As mentioned previously, you *must* compile your program and units with debug information enabled for TPC to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a **/\$D-** compiler directive or a **/SD-** option, TPC will not be able to locate run-time errors.

The link buffer option

This is the same as the Disk setting (O|L|Link Buffer).

The **/L** option disables buffering in memory when .TPU files are linked to create an .EXE file. Turbo Pascal's built-in linker makes two passes. In the first pass through the .TPU files, the linker marks every procedure that gets called by other procedures. In the second pass, it generates an .EXE file by extracting the marked procedures from the .TPU files. By default, the .TPU files are kept in memory between the two passes; however, if the **/L** option is specified, they are reread during the second pass. The default method is faster but requires more memory; for very large programs, you may have to specify **/L** to link successfully.

The quiet option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When TPC is invoked with the quiet mode option

```
TPC MYSTUFF /Q
```

its output is limited to the sign-on message and the usual statistics at the end of compilation. If an error occurs, it will be reported.

Directory options

TPC supports several options that allow you to specify the five directory lists used by TPC: Turbo, EXE & TPU, Include, Unit, and Object.

The EXE & TPU directory option

This option lets you tell TPC where to put the .EXE and .TPU files it creates. It takes a directory path as its argument:

```
TPC MYSTUFF /EC:\TP\BIN
```

*This is the same as the **O/D/I** EXE & TPU Directory command.*

If no such option is given, TPC creates the .EXE and .TPU files in the same directories as their corresponding source files.

The include directories option

Turbo Pascal supports include files through the `{$I filename}` compiler directive. The `/I` option lets you specify a list of directories in which to search for Include files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPC to search for include files in C:\TP\INCLUDE and D:\INC *after* searching the current directory:

```
TPC MYSTUFF /IC:\TP\INCLUDE;D:\INC
```

*This is the same as **O/D/I** Include Directories command.*

If multiple `/I` directives are specified, the directory lists can be concatenated. Thus

```
TPC MYSTUFF /IC:\TP\INCLUDE/ID:\INC
```

is equivalent to the first example.

The unit directories option

When you compile a program that uses units, TPC first attempts to find the units in TURBO.TPL (which is loaded along with TPC.EXE). If they cannot be found there, TPC searches for `unitname.TPU` in the current directory. The `/U` option lets you specify additional directories in which to search for units. As with

*This is the same as the **O|D|I** Unit Directories command.*

the previous options, you can specify multiple directory paths as long as you separate them with semicolons (;). For example, the following command line causes TPC to look in C:\TP\UNITS and C:\LIBRARY for any units it doesn't find in TURBO.TPL or the current directory:

```
TPC MYSTUFF /UC:\TP\UNITS;C:\LIBRARY
```

As with the **/I** option, if multiple **/U** options are specified, the directory lists can be concatenated.

The object files directories option

*This is the same as the **O|D|I** Object Directories command.*

Using **{\$L filename}** compiler directives, Turbo Pascal allows you to link in .OBJ files containing external assembly language routines, as explained in Chapter 22, "The inline assembler," in the *Programmer's Guide*. The **/O** option lets you specify a list of directories in which to search for such .OBJ files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPC to search for .OBJ files in C:\TP\ASM and D:\OBJECT *after* searching the current directory:

```
TPC MYSTUFF /OC:\TP\ASM;D:\OBJECT
```

Like the **/I** option, if multiple **/O** options are specified, the directory lists can be concatenated.

Debug options

Turbo Pascal's IDE features a built-in debugger; TPC has a number of command-line options that also enable you to generate debugging information for standalone debuggers, including Borland's *Turbo Debugger*.

The map file option

Unlike the binary format of .EXE and .TPU files, a .MAP file is a legible text file that can be output on a printer or loaded into the editor.

The **/G** option, like the **O|L|I** Map File command, instructs TPC to generate a .MAP file that shows the layout of the .EXE file. The **/G** option must be followed by the letter **S**, **P**, or **L** to indicate the desired level of information in the .MAP file. A .MAP file is divided into three sections:

- Segment
- Publics

■ Line Numbers

The **/GS** option outputs only the Segment section, **/GP** outputs the Segment and Publics section, and **/GD** outputs all three sections.

For modules (program and units) compiled in the **{SD+,L+}** state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the **{SD+,L-}** state, only symbols defined in a unit's **interface** part are listed in the Publics section.



For modules compiled in the **{SD-}** state, there are no entries in the Line Numbers section.

The standalone debugging option

When you specify the **/V** option on the command line, TPC appends Turbo Debugger-compatible debug information at the end of the .EXE file. Turbo Debugger includes both source- and machine-level debugging, powerful breakpoints (including breakpoints with conditionals or expressions attached to them), and it lets you debug huge applications via virtual machine debugging on a 80386 or two-machine debugging (connected via the serial port).

This is the same as the Standalone option (Options / Debugger.

Even though the debug information generated by **/V** makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and if it is executed from DOS, the .EXE file does not require additional memory.

Turbo Debugger (TD.EXE) is a powerful, standalone debugger that works on Turbo Pascal, Turbo C++, and Turbo Assembler .EXE files.

The extent of debug information appended to the .EXE file depends on the setting of the **\$D** and **\$L** compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the **{SD+,L+}** state, which is the default, *all* constant, variable, type, procedure, and function symbols become known to the debugger. In the **{SD+,L-}** state, only symbols defined in a unit's **interface** section become known to the debugger. In the **{SD-}** state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

The TPC.CFG file

You can set up a list of options in a configuration file called TPC.CFG, which will then be used in addition to the options entered on the command line. Each line in TPC.CFG corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a TPC.CFG file, you can change the default setting of any command-line option.

TPC allows you to enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a TPC.CFG file, you can still override them on the command line.

When TPC starts, it looks for TPC.CFG in the current directory. If the file isn't found there, and if you are running DOS 3.x, TPC looks in the Turbo directory (where TPC.EXE resides). To force TPC to look in a specific list of directories (in addition to the current directory), specify a */T* command-line option as the first option on the command line.

If TPC.CFG contains a line that does not start with a slash (/) or a hyphen (-), that line defines a default file name to compile. In that case, starting TPC with an empty command line (or with a command line consisting of command-line options only and no file name) will cause it to compile the default file name, instead of displaying a syntax summary.

Here's an example TPC.CFG file, defining some default directories for include, object, and unit files, and changing the default states of the **\$F** and **\$S** compiler directives:

```
/IC:\TP\INC;C:\TP\SRC
/OC:\TP\ASM
/UC:\TP\UNIT
/$F+
/$S-
```

Now, if you type

```
TPC MYSTUFF
```

at the system prompt, TPC acts as if you had typed in the following:

```
TPC /IC:\TP\INC;C:\TP\SRC /OC:\TP\ASM /UC:\TP\UNIT /$F+ /$S- MYSTUFF
```

Compiling in protected mode

*TPCX uses the same
command-line options as
TPC.*

If you've purchased the Professional package and have a 286, 386, or 486 machine with at least 1 Mb of extended memory, you can run TPCX.EXE. TPCX can build very large programs by running in protected mode and using extended memory. Note that TPCX can only make use of extended memory, not EMS.

TPCX is much larger than TPC, and running in protected mode involves more overhead than running in real mode. Use TPC to do command-line compiling unless you need the extended memory capacity of TPCX.

- \$ *See* compiler, directives
- 8087/80287/80387 coprocessor *See* numeric coprocessor
- 8087/80287 option 201
- 80286 code generation compiler switch 1, 170
- 43/50-line display 206
- » (chevron) in dialog boxes 16
- 286 Instructions option 1, 200
- ^ (indirection) operator 41
- 25-line display 206
- ; (semicolons) in directory path names 205
- ≡ (System) menu 177
- (arrows) in dialog boxes 15

A

- \$A compiler directive 170
- About command 177
- Abs function 138
- abstract objects 104
- activating
 - menu bar 8
- actual parameters, defined 51
- Add Watch
 - box 128
 - command 25, 134, 196
 - hot key 11
- Addr function 138
- address, Borland 6
- address-of (@) operator 41
- address operators 41
- alignment
 - word 199
- ancestors 75, 78
 - assigning descendants to 97
 - immediate 78
- arguments
 - command-line compiler 229
- arithmetic operators 39

- arrows (→) in dialog boxes 15
- .ASM files, MAKE utility and 161
- assembly language
 - linking routines 160
 - MAKE utility and 161
- assignment, operators 38
- Auto Save option 207
- Autoindent Mode option 208

B

- /B command-line option
 - in TPC 234
- \$B compiler directive 41, 170, 204
- Backspace Unindents option 208
- backup
 - files (.BAK) 208
 - files, automatic 22
- backward
 - pair matching 227
 - searching 186
- .BAK files 208
- bar, title 12
- binary
 - arithmetic operators 39
 - floating-point arithmetic 32
 - format 237
- binding
 - early 96
 - late 97
 - with polymorphic objects 104
- bitwise operators 39
- Boolean 31
 - evaluation 170
 - expressions 36
 - types 35
- Borland
 - address 6
 - CompuServe Forum 6

- technical support 6
- Borland Graphics Interface (BGI)
 - EGA palettes and 176
- Breakpoints
 - command 197
 - dialog box 197
- breakpoints 124, 131-132
 - clearing 198
 - controlling 197
 - deleting 197
 - editing 197
 - instant 133
 - losing 198
 - setting 196
 - viewing 197
- bugs
 - reporting to Borland 6
- Build command 161, 193, 234
- build command-line option 234
- buttons
 - Change 226
 - Change All 187
 - Change all 226
 - choosing 15
 - in dialog boxes 15
 - mouse 209
 - radio 16
- byte data type 32

C

- C++ 74
- /C integrated environment option (config) 174
- Call Stack
 - command 142, 202, 214
 - window 125
- calls, tracking 143
- Cancel button 15
- Cascade command 212
- case sensitivity
 - in searches 185
- case statements 45
- CGA
 - snow checking option 175
- Change All button 187, 226
- Change button 226
- Change Dir command 180
- Change Directory dialog box 181

- Char data types 33
 - defined 31
- characters
 - control
 - integrated environment and 16
 - tab
 - printing 181
- check boxes 16
- chevron symbol (») 16
- choosing menu commands
 - integrated environment 8
- Chr function 138
- circular unit reference 63
- Clear command 185, 223
 - hot key 10
- Clear Desktop command 177
- click speed (mouse) 209
- Clipboard 183
 - clearing 185
 - editing text in 184
 - showing 184
- close boxes 12
- Close command 213
 - hot key 10
- code
 - conditional execution 49
 - iterative execution 49
- Code Generation
 - group 199
- Colors dialog box 210
- columns
 - numbers 12
- command line
 - viewing from integrated environment 213, 214
- command-line
 - compiler reference 229-240
 - options 231-238
 - /B 234
 - /D 232
 - debug 237
 - directory 236
 - /E 236
 - /F 234
 - /G 237
 - /GD 238
 - /GP 238

- /GS 238
- /I 236
- /L 203, 235
- /M 233
- mode 233
- /O 237
- /Q 235
- switching directive defaults (/) 231
- /U 236
- /V 238
- command-line compiler 149
 - arguments 229
 - compiling and linking with 229
 - protected mode 240
- commands *See also* individual listings
 - choosing
 - with a mouse 9
 - with keyboard 8
 - editor 220-226
 - block operations 221, 222-223
 - cursor movement 220, 222
 - insert and delete 221
- comments 52
 - program 52
- compatibility
 - object 97, 98
 - pointers to objects 98
- compilation 20
 - conditional 164
 - unit 68
- Compile
 - command 27, 192
 - hot key 11
 - menu 20, 192
- compile-time errors 20, 122
- Compiler
 - command 199
- compiler
 - command-line *See* command-line, compiler
 - directives
 - \$R
 - virtual method checking 103
 - \$R 200
 - \$A 170
 - \$B 41, 170, 201
 - \$D 125, 148, 201, 234
 - \$DEFINE 164, 232
 - \$E 170
 - \$ELSE 164, 166, 167
 - emulation 170
 - \$ENDIF 166
 - \$F 199
 - \$G 1, 170, 200
 - \$I 170, 200, 206
 - \$IFDEF 164, 166, 168
 - \$IFNDEF 164, 168
 - \$IFOPT 164, 169
 - \$IFOPT N+ 169
 - \$L 1, 58, 125, 202, 206
 - local symbol 1
 - \$M 202, 232
 - \$N 168, 201
 - 32
 - \$O 199
 - \$R 171
 - \$S 171, 200
 - \$UNDEF 164
 - \$V 171, 201
 - \$X 1, 171, 201
 - mode, command-line options *See* command-line, options
 - options *See* command-line, options
- compiling
 - protected mode 240
 - to .EXE file 192, 193, 233, 234
 - to disk 147, 149
- Complete Boolean Eval option 41, 201
- compound statements 44
- CompuServe Forum, Borland 6
- computerized simulations 88
- conditional
 - compilation 164
 - defines (command-line option) 232
 - execution 30
 - statements 43
 - symbols 165
- Conditional Defines option 202
- Config File Directory option 207
- CONFIG.SYS file
 - modifying 147
- Configuration file integrated environment
 - option 174
- configuration files
 - retrieving 211

- saving *147, 207, 211*
- TPC.CFG *239*
- constructor (reserved word) *102*
- constructors
 - defined *102*
 - virtual methods and *102, 110*
- Contents command *215*
 - hot key *11*
- context-sensitive help *7*
- control characters *34*
 - entering in integrated environment *16*
 - format specifier *195*
- conventions
 - typographic *5*
- Copy command *184, 223*
 - hot key *10*
- Copy Example command *184, 215*
- copyright information *177*
- CPU
 - registers *213*
 - symbols *166*
- Create Backup Files option *208*
- Crt unit *56, 66*
- Ctrl-Break *189, 190*
- Current Directory option *207*
- Current window option *207*
- Cursor Through Tabs option *208*
- customizing
 - color *210*
 - integrated environment *206*
- Cut command *184, 223*
 - hot key *10*

D

- /D command-line option *232*
- \$D compiler directive *125, 148, 201, 234*
- /D integrated environment option (dual monitors) *174*
- data *30*
 - aligning *199*
 - defined *30*
 - types
 - Boolean *31, 35*
 - byte *32*
 - Char *33*
 - char *31*

- defined *31*
- integer *31*
- longint *32*
- pointer *31, 36*
- real *33*
- real numbers *31*
- shortint *32*
- string *35*
- word *32*
- Debug Information option *148, 201*
 - Trace into command and *191*
- Debug menu *194*
- debugger, integrated *See* integrated, debugger; debugging
- Debugger command *203*
- Debugger Options dialog box *204*
- debugging
 - Add Watch box *128*
 - basic unit of execution *125*
 - compile-time errors *122*
 - dialog box choices *204*
 - display swapping *204*
 - dual monitors and *204*
 - example *127*
 - expressions *194*
 - format specifiers *195*
 - global identifiers *125*
 - hot keys *11*
 - I/O error checking *152*
 - IFDEF and *168*
 - IFNDEF and *168*
 - inability to trace *150*
 - information *189, 204*
 - disabling *148*
 - generating *201*
 - line-number *201*
 - local identifiers *125*
 - memory *146*
 - navigation *142*
 - options, command-line *237*
 - pitfalls *151*
 - preventive *146*
 - range checking *153*
 - restarting *126*
 - run-time errors *122*
 - starting a session *189*
 - Step Over command *191*

- stopping *See* Program Reset command
- syntax errors 122
- Trace Into command 191
- tracing 124
- variables 194
- watchpoints
 - adding 196
 - controlling 196
 - deleting 196
 - editing 196
 - watch window 213
- Debugging command 189
 - and Trace Into command 191
- declaration
 - methods 81, 82
 - object instances 79
- declarations, unit 60
- default buttons 15
- \$DEFINE compiler directive 164, 232
- Delete Watch command 196
- deleting line
 - undoing 183
- delimiters
 - directional 227
 - nesting 227
 - nondirectional 227
- descendants 78
 - immediate 78
- designators
 - field 85
- desktop
 - clearing 177
- Desktop File option 207
- Desktop option 207
- Destination command 22, 147, 193
- destination default setting 193
- destructors
 - declaring 113
 - defined 112, 113
 - dynamic object disposal 114
 - polymorphic objects and 113
 - static versus virtual 113
- dialog boxes
 - arrows in 15
 - Breakpoints 197
 - Change Directory 181
 - Colors 210
 - Debugger Options 204
 - defined 15
 - Directories 205
 - entering text 16
 - Environment Options 221, 224
 - Find 185, 225
 - Find Procedure 188
 - Get Info 182
 - Go to Line Number 188
 - Linker 203
 - Load a File 224
 - Open a File 178
 - Preferences 206
 - Program Parameters 192
 - Replace 187, 225
 - Save File As 180
 - Startup Options 210
- directional pair matching 227
- directives *See* compiler, directives
- Directories
 - command 205
 - dialog box 205
- directories
 - changing 180
 - command-line options 236
 - configuration 205
 - defining 205
 - semicolons in paths 205
- display
 - formats
 - debugger 195
 - swapping 204
 - dual monitors and 204
- Display Swapping command 125
- Dispose procedure 37
 - extended syntax 112
- distribution disks
 - backing up 3
- div operator 33
- DOS
 - MODE command 174
 - output
 - viewing from integrated environment 213, 214
 - symbol 166
 - wildcards 178
- DOS Shell command 22, 177, 182

- Dos unit 56, 66
- dotting 79, 83, 86
- double-click speed (mouse) 209
- dual monitor mode 174, 175
- dual monitors 174
 - display swapping and 204
 - DOS command line and 182
- dynamic object instances 110-118
 - allocation and disposal 115

E

- /E command-line option 236
- \$E compiler directive 170, 201
- /E integrated environment option (dual monitors) 175
- early binding 96
- Edit
 - menu 182
 - window 18
- Edit Watch command 196
- Edit windows
 - activating 219
 - cursor
 - moving 220, 222
 - option settings 208
- editing 17
 - autoindent mode 224
 - block operations 221, 222-223
 - deleting 223
 - hiding/unhiding 223
 - printing 223
 - reading and writing 223
 - selecting blocks 182, 223
 - breakpoints 197
 - Clipboard text 184
 - commands 220-226
 - cursor movement 220, 222
 - insert and delete 221
 - copy and paste 223
 - hot key 10
 - cut and paste 183, 184, 223
 - entering text 220
 - hot keys 10, 220-226
 - insert mode
 - overwrite mode vs. 208
 - miscellaneous commands 224
 - place marker 224

- print file 223
- quitting 224
- restore line 224
- search and replace 225-226
 - options 225
- selecting text 182
- tab mode toggle 224
- tabs 224
 - undelete 224
- undoing line edits 183
- watchpoints 196
- editor
 - features 17
 - tabs in 208
- Editor Files option 207
- Editor Options 208
- EGA 26
- ellipsis mark (...) 8, 15
- \$ELSE compiler directive 164, 166, 167
- ELSE symbol 167
- EMS
 - memory 147
- emulation, 80x87
 - floating point 201
- Emulation option 149
- encapsulation 75, 88
- \$ENDIF compiler directive 166
- ENDIF symbol 167
- Enhanced Graphics Adapter (EGA) 207
 - palette
 - integrated environment option 176
- Environment
 - command 206
- Environment option
 - Auto Save 207
- Environment Options dialog box 221, 224
- errors 21
 - checking 200
 - compile-time 20, 122
 - handling 152, 170
 - I/O 152
 - messages
 - searching 188, 234
 - out-of-memory/bounds 153
 - run-time *See* run-time errors
 - syntax 20, 122
- Esc shortcut 15

- Evaluate
 - command 139
 - format specifiers and 195
 - window 139
 - objects and 144
- Evaluate/Modify command 148, 194
- hot key 11
- event handling
 - virtual methods and 107
- examples
 - copying from Help 184, 215
- EXE & TPU Directory command 205
- EXE & TPU directory command-line option 236
- .EXE files
 - creating 192, 193, 233, 234
 - storing 193, 205
- executable
 - code, storing 193
 - directories command 205
- execution
 - bar 128
- exiting Turbo Pascal 177
- expanded memory 176
 - RAM disk and 147, 176
 - TSRs and 147
- Expanded Memory Specification *See* EMS
- exported object types 85
- expressions
 - debugging 194
 - nested
 - pair matching 226
 - values
 - displaying 194
 - Watch *See* Watch, expressions
- extended
 - memory support *See* EMS memory
 - syntax 1, 171
- Extended Syntax option 1, 201
- extensibility 108

F

- /F command-line option 234
- \$F compiler directive 199
- FAR call
 - model, forcing use of 199
- features
 - editor 17
 - integrated environment 173
- field-width specifiers 42
- fields
 - object 79
 - accessing 80, 82, 88
 - designators 85
 - inherited 79
 - scope 83
 - method parameters and 85
 - private and encapsulation 82, 87, 88
- File menu 20, 178
- files
 - backup (.BAK) 208
 - closed
 - reopening 214
 - .EXE
 - storing 193, 205
 - information on 181
 - .MAP 237
 - storing 205
 - new 179, 224
 - NONAME 179
 - .OBJ 237
 - locating 206
 - open
 - choosing from List window 214
 - opening 178, 224
 - hot key 9
 - printing 181
 - saving 179, 224
 - all 180
 - automatically 207
 - hot key 9
 - with new name or path 180
 - .TPU 69, 193
 - debug information 201
 - local symbol information 202
 - storing 205
- filling lines with tabs and spaces 208
- Find command 185, 225
- Find dialog box 185, 225
- Find Error command 21, 188, 234, 235
- find error command-line option 234
- Find Procedure
 - command 125, 142, 143, 188
 - methods and 145
 - dialog box 188

- floating point
 - code generation *201*
 - format specifier *195*
 - numbers *31*
- for
 - statements, loop *48*
- Force Far Calls option *199*
- formal parameters, defined *51*
- format specifiers *136*
 - debugging and *195*
 - table *195*
 - objects *144*
 - repeat count *136*
 - using *136*
- 43/50-line display *206*
- forward
 - forward searching *186*
 - pair matching *227*
- FreeList
 - version compatibility *1*
- FreeMin
 - version compatibility *1*
- functions
 - defined *49*
 - finding *143*
 - structure *50*

G

- /G command-line option *237*
- \$G compiler directive *1, 170, 200*
- /G integrated environment option (Graphics save) *175*
- /GD command-line option *238*
- generating line-number tables *125*
- Get Info
 - command *181*
 - dialog box *182*
- Get info
 - command *147*
- global
 - identifiers *125*
- Go to Cursor command *126, 190*
 - hot key *9, 11*
- Go to Line Number
 - command *188*
 - dialog box *188*
- /GP command-line option *238*

- Graph3 unit *56, 67*
- Graph unit *27, 56, 67*
- graphics *26*
 - integrated environment option (/G) *148, 175*
 - palette
 - EGA *176*
- Graphics Screen Save option *148, 175*
- GREP (file searcher)
 - wildcards in Turbo Pascal *185*
- /GS command-line option *238*

H

- heap
 - editor *175*
 - management
 - sizes *202, 232*
 - overlay *176*
 - size *149*
 - window *176*
- Help
 - button *15*
 - menu *214*
 - windows
 - closing *215*
 - copying from *184, 215*
 - keywords in *215*
 - opening *214*
 - selecting text in *215*
- help
 - accessing *214*
 - help on help *217*
 - hot keys *9, 11*
 - index *216*
 - keywords *215*
 - language *216*
 - online
 - in integrated environment *2, 7*
 - Pascal *216*
 - previous topic *216*
 - status line *15*
 - table of contents *215*
- Help on Help command *217*
 - hot key *11*
- hexadecimal constants *32*
- Hi function *138*
- hierarchies
 - object *78*

- common attributes in 104, 107
- high heap limit setting 203
- history lists 16
 - closing 177
 - wildcards and 178
- hot keys
 - debugging 11
 - editing 10
 - editor 219, 220-226
 - help 9, 11
 - menus 10
 - using 9

I

- /I command-line option 236
- \$I compiler directive 170, 200, 206
- I/O
 - defined 30
 - error checking 152
 - disabling 153
 - error-checking 170
- I/O Checking option 200
- IDE *See* integrated, development environment
- identifiers 37
 - defined 37
 - naming restrictions 37
- IEEE floating-point 166
- if statements 44
- IFDEF 167
- \$IFDEF compiler directive 164, 166, 168
- IFNDEF 167
- \$IFNDEF compiler directive 164, 168
- IFOPT 167
- \$IFOPT compiler directive 164, 169
- IFxxx symbol 167
- immediate ancestors and descendants 78
- implementation sections 65
 - uses clauses in 63
- Include Directories command 206
- include directories command-line option 236
- include files 236
 - help 216
- incremental search 17
- indenting automatically 208
- Index command
 - help 216
 - hot key 11

- index variable 48
- indirection (^) operator 41
- infinite loop 28, 129
- inheritance 75, 76, 78
- initialization
 - units 159
 - variables 59
- inline assembler 1
- input 30
 - boxes 16
 - defined 30
 - functions 43
- Insert Mode option 208
- instances
 - defined 77
 - dynamic object 110-118
 - object
 - declaring 79
 - linked lists of 115
 - static object 76-110
- Instructions
 - 286 200
- integers
 - defined 31
 - types 31
- integrated
 - debugger 24, *See also* debugging
 - development environment
 - commands *See* individual listings
 - Edit window *See* Edit, window
 - menus *See also* menus
 - windows *See also* windows
- Integrated (debugging) option 148, 189
- integrated environment 173
 - command-line arguments and 191
 - command-line options 147, 174
 - config (/C) 174
 - dual monitors (/D) 174
 - editor heap (/E) 175
 - EGA palette (/P) 176
 - expanded memory (/X) 176
 - graphics (/G) 148, 175
 - laptops (/L) 175
 - loading TURBO.TPL (/T) 148, 176
 - /O overlay heap 176
 - overlay heap (/O) 176
 - /P EGA palette 176

- RAM disk (/S) 176
- snow checking (/N) 175
- syntax 174
- /T loading TURBO.TPL 148, 176
- window heap size (/W) 176
- /X expanded memory 176
- compiling in 20
- context-sensitive help 2, 7
- control characters and 16
- customizing 206
- features 173
- Graph unit and 27
- graphics 26
- increasing capacity 147
- loading 18
- main screen 18
- memory issues 147
- menus
 - choosing commands from 8
 - saving files in 20
 - statements 19
 - tutorial 18
 - variables 19
- interface sections 65
- IOResult function 138, 200

K

- keyboard
 - choosing buttons with 15
 - choosing commands with 8
 - selecting text with 183
- keywords 7
 - help 216
 - Help windows 215

L

- /L command-line option 203, 235
- \$L compiler directive 1, 58, 125, 202, 206
- /L integrated environment option (LCD screen) 175
- language help 216
- laptops
 - integrated environment option (/L) 175
- large programs, managing 157
- late binding 97
 - with polymorphic objects 104

- left-handed
 - mouse support for 209
- Length function 138
- license statement 3
- line-number tables 201
- lines
 - filling with tabs and spaces 208
 - moving cursor to 188
 - numbering 12
 - restoring (in editor) 183
- Link Buffer
 - option 203, 235
 - setting 147
- linked lists 115
- Linker
 - command 203
 - dialog box 203
- linking
 - buffer option 235
 - \$L compiler directive 58
- List
 - command
 - hot key 10
 - window 211, 214
- list boxes 17
 - file names 179
 - searching incrementally 216
- Lo function 138
- Load a File dialog box 224
- Load TURBO.TPL option 148
- loading
 - Turbo Pascal 18
- local
 - identifiers 125
 - symbol information, generating 202
 - symbol information switch 1
- Local Symbols option 202
- logic errors 122
- logical operators 40
- longint data type 32
- loops
 - defined 31
 - for 48
 - repeat..until 46
 - while 46
- low heap limit setting 202
- low-level operations 39

M

- /M command-line
 - option 233
- \$M compiler directive 202, 232
- Make command 160, 192, 233
 - hot key 9, 11
- make command-line option 233
- MAKE utility
 - .ASM files and 161
 - command-line options 163
- Map File
 - command 206, 238
 - option 204
- map file command-line option 237
- map files
 - options 203
- .MAP files 237
 - storing 205
- math coprocessor *See* numeric coprocessor
- MaxAvail function 138
- MaxInt 31
- Mem array 138
- MemAvail function 138
- MemL array 138
- memory 146
 - allocation 232
 - conserving 147
 - defaults, configuring 202
 - dump
 - format specifier 195
 - EMS 147
 - expanded 176
 - RAM disk and 147, 176
 - menu command 202
- Memory Sizes command 149, 202
- MemW array 138
- menus
 - accessing 8
 - choosing commands 8
 - commands *See* individual listings
 - File 20
 - hot keys 9, 10
 - opening 8, 219
 - Run 21
 - with arrows (►) 8
 - with ellipsis marks (...) 8, 15

- methods
 - assembly language 85
 - calling 81
 - debugging 144
 - declaring 81, 82
 - defined 80
 - external 85
 - Find Procedure command and 145
 - identifiers, qualified
 - accessing object fields 86
 - in method declarations 81, 83
 - overriding inherited 90
 - parameters
 - naming 85
 - Self 84
 - debugging and 145
 - explicit use of 84
 - positioning in hierarchy 107
 - procedures versus 106
 - scope 83
 - static 95
 - problems with inherited 93
 - virtual 95
 - event handling and 107
 - polymorphic objects and 101
 - static versus 107
- MODE command (DOS) 174
- modifying expressions and variables 140
- monitors
 - dual 174, 182, 204
 - number of lines 206
- mouse
 - buttons
 - switching 209
 - choosing commands with 9, 15
 - double-click speed 209
 - left-handed
 - support for 209
 - reversing buttons 209
 - right button action 209
 - selecting text with 183
 - support for 173
- Mouse Double Click option 209

N

- \$N compiler directive 32, 168, 201

/N integrated environment option (CGA snow checking) 175

New

command 179, 224

procedure 110

extended syntax 111

used as function 112

New Value field 194

New Window option 207

Next command 212

hot key 9, 10

NONAME file name 179

nondirectional pair matching 227

numbers

decimal

format specifier 195

hexadecimal

format specifier 195

numeric

constants 27

coprocessor 32, 166, 168

coprocessors

inline instructions 201

O

/O command-line option 237

\$O compiler directive 199

/O integrated environment option (overlay heap) 176

.OBJ files 237

locating 206

MAKE utility and 161

object (reserved word) 78

object directories command-line option 237

Object Directories option 237

Object Directory command 206

objects

abstract 104

ancestor 78

constructors

defined 102

virtual methods and 102, 110

debugging 144

Evaluate window and 144

stepping and tracing 144

Watch window and 135

defined 74

descendant 78

destructors

declaring 113

defined 112, 113

dynamic object disposal 114

polymorphic objects and 113

static versus virtual 113

dynamic instances 110-118

allocation and disposal 115

extensibility 108

fields 79

accessing 80, 82, 88

designators 85

inherited 79

scope 83

method parameters and 85

hiding data representation 90

hierarchies 78

common attributes in 104, 107

instances

declaring 79

linked lists of 115

passed as parameters

compatibility 98

pointers to

compatibility 98

polymorphic 99

late binding and 104

relative position 118

static instances 76-110

types

compatibility

97

exported by units 85

units and 85

virtual method table

pointer

initialization 102

Ofs function 138

OK button 15

Open a File dialog box 178

Open command 178, 224

hot key 9, 10

operations 30

defined 30

low-level 39

- operators
 - ^ (indirection) 41
 - address 41
 - address-of (@) 41
 - arithmetic 39
 - assignment 38
 - binary 38, 39
 - bitwise 39
 - defined 38
 - div 33
 - logical 40
 - precedence of 38
 - relational 39
 - set 41
 - string 41
 - unary 38, 39
- Optimal Fill option 208
- optimization of code 170
- Options menu 198
- Ord function 138
- out-of-memory/bounds errors 153
- output
 - defined 30
 - devices 41
 - to DOS
 - viewing from integrated environment 213, 214
 - User Screen 214
 - Writeln 41
- Output command 213
- Overlay Heap Size option 148
- Overlay unit 56, 66
- overlays
 - enabling 199
- Overlays Allowed option 199
- overriding inherited methods 90
- Overwrite Mode 208

P

- /P integrated environment option (EGA palette) 176
- pair matching
 - backward 227
 - braces 226
 - brackets 226
 - commands 226
 - directional 227

- double quotes 226
- forward 227
- nested expressions 226
- nondirectional 227
- parentheses 226
- rules 226
- single quotes 226
- Parameters
 - command 191
- parameters
 - method, naming 85
- Self 84
 - debugging and 145
 - explicit use of 84
- Pascal
 - language help 216
- Paste command 184, 223
 - hot key 10
- path names in Directories dialog box 205
- place markers (editor) 224
- pointers 36
 - defined 31
 - format specifier 195
- polymorphic objects 99
 - late binding and 104
 - virtual methods and 101
- polymorphism 95, 97, 98
- pop-up menus 8
- Pred function 138
- Preferences dialog box 206
- Previous command 212
 - hot key 10
- Previous Topic command 216
 - hot key 11
- Primary File command 193
- Print Block command 223
- Print command 181
- Print File command 223
- Printer unit 56, 67
- private 88
 - fields and methods 82, 87, 88
- procedures
 - defined 49
 - Dispose
 - extended syntax 112
 - finding 143, 188
 - help 216

- methods versus 106
- New 110
 - extended syntax 111
 - used as function 112
- searching for 188
- stepping over 191
- structure 50
- tracing into 191
- Program Parameters
 - dialog box 192
- Program Reset command 126, 190
 - hot key 11
- programming, elements of 29
- programs
 - comments 52
 - compiling 20, 27
 - debugging *See* debugging
 - editing 18
 - ending 189
 - rebuilding 190, 193, 234
 - reinitializing 126
 - resetting 190
 - running 21, 189
 - parameters for 191
 - to cursor 190
 - Trace Into 191
 - saving 20
 - stepping through 25
 - structure of 49, 157
 - tracing 124
 - updating 21
- project management 157
- protected mode
 - command-line compiler and 240
- Ptr function 138

Q

- /Q command-line option 235
- qualified method identifiers
 - accessing object fields 86
 - in method declarations 81
- quiet mode command-line option 235
- Quit
 - command 177, 182, 224
- quitting
 - debugging *See* Program Reset command

R

- /R command-line option
 - \$R and 232
- \$R compiler directive 171, 200
 - virtual method checking 103
- radio buttons 16
- RAM disk
 - integrated environment and 147, 176
- range checking 171, 200, 232
 - errors 153
 - selectively implementing 154
- Range Checking option 149, 200
- Read procedure
 - text files 43
- Readln procedure 43
- real numbers 31, 32
- records
 - types 77
- recursion 150
- Refresh Display command 126, 177
- Register command 213
- registers
 - windows 213
- reinitializing a program 126
- relational operators 39
- relative position 118
- Remove All Watches command 196
- repeat..until loop 46
- repeat count 136
- Replace
 - command 187, 225
 - dialog box 187, 225
- reserved words 56, 57
 - constructor 102
 - object 78
 - virtual 101
- resetting programs 190
- resize corner 12, 13
- restarting a debugging session 126
- Restore Line command 183, 220, 224
- Result field 194
- Retrieve Options command 211
- Reverse Mouse Buttons option 209
- Right Mouse Button option 209
- Round function 138
- routines, recursive 150

- Run
 - command 189
 - hot key 11
 - menu 21, 189
 - run
 - bar 24
 - run-time errors 23, 122, 200
 - Debug Information command and 188
 - Find Error command and 188, 234
 - finding 234
 - running programs 21, 189
- S**
- \$S compiler directive 171, 200
 - /S integrated environment option (RAM disk) 176
 - sample programs
 - copying from Help window 184
 - Save All command 180
 - Save As
 - command 180
 - Save command 179, 224
 - hot key 9, 10
 - Save File As dialog box 180
 - Save Options command 211
 - saving
 - programs 20
 - scope, object fields and methods 83
 - Screen Sizes
 - option 206
 - screens
 - LCD
 - integrated environment option 175
 - number of lines 206
 - swapping 125
 - two
 - using 174
 - scroll bars 12, 13
 - scrolling windows 13
 - Search Again command 188, 225
 - hot key 10
 - search and replace 225-226
 - Search menu 185, 225-226
 - searching *See* GREP utility (file searcher)
 - and replacing text 225, 225-226
 - direction 186
 - in list boxes 216
 - origin 187
 - procedures 188
 - regular expressions 185
 - repeating 188
 - and replacing text 187
 - run-time error messages 188, 234
 - scope of 186
 - search and replace 187
 - Seg function 138
 - Self parameter 84
 - debugging and 145
 - explicit use of 84
 - semantic errors 122
 - semicolons (;) in directory path names 205
 - separate compilation 55
 - sets, operators 41
 - setting breakpoints 126
 - short-circuit Boolean
 - expressions 201
 - shortint data type 32
 - Show Clipboard command 184
 - significant digits, defined 32
 - Simula-67 88
 - simulations, computerized 88
 - single-step tracing 127
 - Size/Move command 212
 - SizeOf function 138
 - Smalltalk 74, 88
 - snow checking
 - integrated environment option (/N) 175
 - software license agreement 3
 - software numeric processing *See* Numeric Processing command
 - Source Tracking option 207
 - spaces vs. tabs 208
 - SPtr function 138
 - SSeg function 138
 - stack
 - checking 171, 200
 - size 202
 - decreasing 149
 - Stack Checking option 149, 200
 - Standalone Debugging
 - command 238
 - option 204
 - standalone debugging
 - command-line option 238

- information 204
- standard units *See* units, standard
- Startup
 - dialog box 210
 - options 147, 209
- Startup Options dialog box 147, 210
- statements 19
 - case 45
 - compound 44
 - conditional 43
 - if 44
 - uses 57, 59
 - with 79, 86
 - implicit 84
- static
 - methods 95
 - problems with scope of inherited 93
 - object instances 76-110
- status line 14
- status window *See* compilation window
- Step Over command 191
 - hot key 9, 11
 - methods and 144
- stepping through a program 25
- strict error checking 200
- Strict Var-strings option 200
- strings 35
 - format specifier 195
 - operators 41
- subroutines 31, 49
- Succ function 138
- Swap function 138
- swapping
 - displays 204
 - screens 125
- symbols
 - local information 1
- syntax
 - errors 122
 - extended 1, 171, 201
 - integrated environment command line 174
 - options 200
- System unit 56, 60, 66

T

- /T integrated environment option (load TURBO.TPL) 176

- Tab Size option 208
- tabs
 - characters
 - printing 181
 - size of 208
 - spaces vs. 208
 - using in the editor 208
- taxonomy 75
- TD.EXE 238
- technical support 6
- text
 - copy and paste 184
 - cutting 184
 - deleting 185
 - entering 220
 - in dialog boxes 16
 - inserting vs. overwriting 208
 - pasting 184
 - screen display of 206
 - selecting 182
 - Help window 215
- Tile command 212
- title bars 12
- Toggle Breakpoint command 126, 131, 196
 - hot key 11
- Topic Search command 216
 - hot key 11
- TPC.CFG file 239
 - sample 239
- TPTOUR 173
- .TPU files 69, 193
 - debug information 201
 - local symbol information 202
 - storing 205
- TPUMOVER.EXE 68, 71
- TPUMOVER utility 71, 148
- Trace Into command 24, 191
 - Debug Information and 191
 - Debugging command and 191
 - hot key 9, 11
 - methods and 144
- tracing
 - programs 124
 - single-step 127
- trapping, I/O errors 152
- Trunc function 138

TSRs
removing from memory 147

Turbo3 unit 56, 67

Turbo Debugger 238, *See also* debugging
standalone 149

TURBO.DSK file
saving 207, 211

TURBO.EXE 18

Turbo Pascal

bugs

reporting 6

quitting 177, 182

starting 174

version 3.0 56

conversion

Graph3 unit 67

Turbo3 unit 67

TURBO.TP file

modifying 147

options stored in 211

saving 147, 207

TURBO.TPL 56, 66, 236

loading 148, 176

tutorial 7, 29

25-line display 206

two's complement 39

typecasting 137

typefaces used in these books 5

types *See* data, types

object

exported by units 85

record 77

typographic conventions 5

U

/U command-line option 236

unary

minus 39

plus 39

\$UNDEF compiler directive 164

Unit Directories

command 148, 206

option 237

units 27, 55

Build option 161

circular reference 63

compiling 68, 160

declarations 60

definition 55

forward declarations and 57

global 157

large programs and 70, 157

implementation section 57

initialization section 59

initializing 159

interface section 57

large programs and 70

Make option 160

merging 68

objects in 85

standard

Crt 56, 66

Dos 56, 66

Graph 56, 67

Graph3 56, 67

Overlay 56, 66

Printer 56, 67

System 56, 60, 66

Turbo3 56, 67

structure 56

.TPU files 68

TPUMOVER 71

TURBO.TPL file 56, 66, 68, 70

Unit Directories

input box 68

Unit directories

option

uses statement 57, 59

unit directories

option 236

use of 59

writing 68

Use Expanded Memory option 147

Use Tab Character option 208

User Screen

command 22, 214

hot key 10

User screen 21

uses

clause

in an implementation section 63

statement 57, 59

utilities

- BINOBJ *See* BINOBJ utility
- GREP *See* GREP utility (file searcher)
- INSTALL *See* INSTALL utility
- MAKE *See* MAKE utility
- TOUCH *See* TOUCH utility
- TPUMOVER *See* TPUMOVER utility
- UPGRADE *See* UPGRADE program

V

- /V command-line option 238
- \$V compiler directive 171, 201
- var parameters
 - checking 171
- variables 19
 - debugging 194
 - index 48
 - initializing 59
 - modifying 140
- VER60 166
- version number information 177
- Video Graphics Array Adapter (VGA) 207
- virtual
 - method table 102
 - pointer
 - initialization 102
 - methods 95
 - event handling and 107
 - polymorphic objects and 101
 - static versus 107
 - reserved word 101

W

- /W integrated environment option (window heap) 176
- Watch
 - expressions
 - acceptable values 138
 - built-in functions 138
 - display 135
 - format specifiers 136
 - repeat count 136
 - using 136
 - modifying 140
 - objects 135

- typecasting 137
- types 135
 - arrays 135
 - Booleans 135
 - characters 135
 - enumerated data types 135
 - files 135
 - integers 135
 - pointers 135
 - reals 135
 - records 135
 - sets 135
 - strings 135

- window
 - editing 139
- watches 124
 - deleting 139
 - editing 139
 - setting up 134
- Watches command 196
- while (syntax)
 - loop 46
- whole-word searching 185
- wildcards 185
 - DOS 178
 - GREP 185
- Window Heap Size option 148
- Window menu 211
- Windows
 - call stack 214
- windows
 - active 14
 - defined 11
 - hot key 10
 - cascading 212
 - Clipboard 184
 - closing 12, 14, 177, 213
 - Edit *See* Edit, window
 - elements of 11
 - Evaluate 139
 - List 214
 - menu 211
 - moving 14, 212
 - next 212
 - open 214
 - opening 14, 211
 - Output 213

- position
 - hot key *10*
- previous *212*
- Register *213*
- reopening *211*
- repainting *126*
- resizing *13, 14, 212*
- scrolling *12, 13*
- size
 - hot key *10*
- source tracking *207*
- swapping in debug mode *204*
 - dual monitors and *204*
- tiling *212*
- title bar *12*
- User Screen *214*
- Watch *213*, *See* Watch, window
- window number *13*
- zooming *12, 13, 14, 212*

- with (reserved word)
 - statement *79, 86*
 - implicit *84*
- Word Align Data option *199*
- word data type *32*
- WriteIn procedure *41*
 - field-width specifiers and *42*

X

- `$X` compiler directive *1, 171, 201*
- `/X` integrated environment option (expanded memory) *176*

Z

- zoom box *12, 13*
- Zoom command *212*
 - hot key *9, 10*

6.0

USER'S
GUIDE

TURBO PASCAL®

B O R L A N D

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300
Offices in: Australia, Denmark, England, France, Germany, Italy, Japan and Sweden ■ Part# 11MN-PAS02-60 ■ BOR 1850