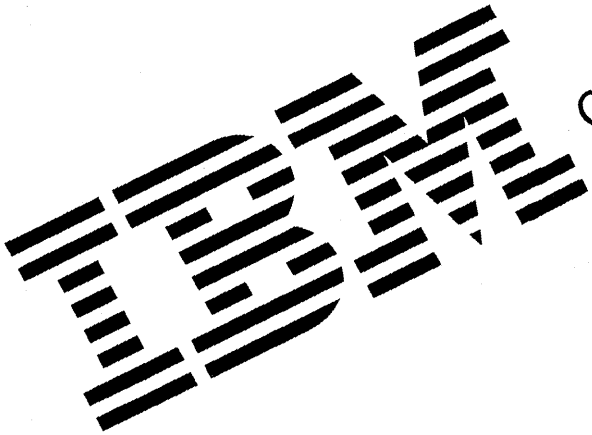


C/2™
Version 1.10

Fundamentals

Programming Family

15F0384

International Business Machines Corporation *Armonk, New York 10504*

IBM Program License Agreement

BEFORE OPENING THIS PACKAGE, YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

This is a license agreement and not an agreement for sale. IBM owns, or has licensed from the owner, copyrights in the Program. You obtain no rights other than the license granted you by this Agreement. Title to the enclosed copy of the Program, and any copy made from it, is retained by IBM. IBM licenses your use of the Program in the United States and Puerto Rico. You assume all responsibility for the selection of the Program to achieve your intended results and for the installation of, use of, and results obtained from, the Program.

The Section in the enclosed documentation entitled "License Information" contains additional information concerning the Program and any related Program Services.

LICENSE

You may:

1) use the Program on only one machine at any one time, unless permission to use it on more than one machine at any one time is granted in the License Information (Authorized Use);
2) make a copy of the Program for backup or modification purposes only in support of your Authorized Use. However, Programs marked "Copy Protected" limit copying;
3) modify the Program and/or merge it into another program only in support of your Authorized Use; and
4) transfer possession of copies of the Program to another party by transferring this copy of the IBM Program License Agreement, the License Information, and all other documentation along with at least one complete, unaltered copy of the Program. You must, at the same time, either transfer to such other party or destroy all your other copies of the Program, including modified copies or portions of the Program merged into other programs. Such transfer of possession terminates your license from IBM. Such other party shall be licensed, under the terms of this Agreement, upon acceptance of this Agreement by its initial use of the Program.

You shall reproduce and include the copyright notice(s) on all such copies of the Program, in whole or in part.

You shall not:

1) use, copy, modify, merge, or transfer copies of the Program except as provided in this Agreement;
2) reverse assemble or reverse compile the Program;
   and/or
3) sublicense, rent, lease, or assign the Program or any copy thereof.

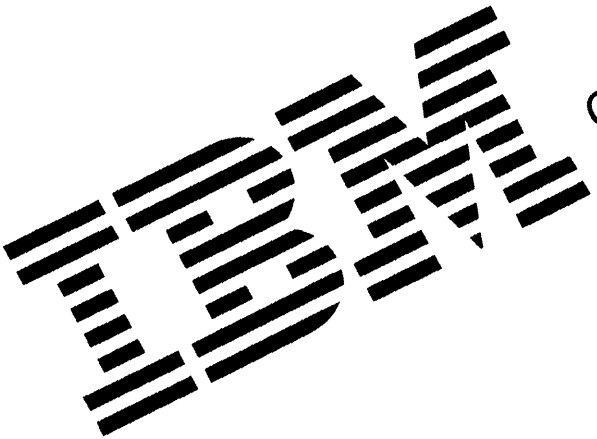
LIMITED WARRANTY

Warranty details and limitations are described in the Statement of Limited Warranty which is available upon request from IBM, its Authorized Dealer or its approved supplier and is also contained in the License Information. IBM provides a three-month limited warranty on the media for all Programs. For selected Programs, as indicated on the outside of the package, a limited warranty on the Program is available. The applicable Warranty Period is measured from the date of delivery to the original user as evidenced by a receipt.

Certain Programs, as indicated on the outside of the package, are not warranted and are provided "AS IS."

Continued on inside back cover.

84X1712



C/2™
Version 1.10

Fundamentals

Programming Family

First Edition (September 1988)

Preface

This book is Volume 1 of a four-volume set explaining the IBM C/2 compiler. It contains fundamental information needed to write programs using the IBM version of the C language, including program structure, functions, variables, expressions, and preprocessing.

This book assumes that first-time users of IBM C/2 have completed at least one year of computer science studies. This book is also intended for experienced applications programmers or system programmers. Users should also be familiar with their personal computer and operating system.

The following table lists some common tasks you may want information about, and which book you can find the information in.

If You Want To...	**Refer To...**
Install IBM C/2	*Fundamentals*
Learn basic facts about IBM C/2	*Fundamentals*
Learn the format of a function	*Language Reference*
Understand error messages	*Compile, Link, and Run*
Debug a program	*Debug*
Compile a program	*Compile, Link, and Run*
Link a program	*Compile, Link, and Run*
Write a program	*Fundamentals* and *Language Reference*

Related Publications

The following books cover topics related to the IBM C/2 Library:

- *IBM C/2 Compile, Link, and Run*
- *IBM C/2 Language Reference*
- *IBM Debug*
- *IBM MASM/2 Fundamentals*
- *IBM MASM/2 Assemble, Link, and Run*
- *IBM MASM/2 Language Reference*
- IBM Operating System/2 Version 1.00 (Standard and Extended Editions)
  - *Programmer's Guide*
- IBM Operating System/2 Version 1.10
  - *Programming Guide*
- The technical reference for your personal computer.
- The technical reference for your operating system.
- *IBM System Application Architecture Common Programming Interface C Reference*
- *iAPX 86, 88 User's Manual*, Copyright 1981, Intel Corp., Santa Clara, CA.
- *iAPX 286 Hardware Reference Manual*, Copyright 1983, Intel Corp., Santa Clara, CA.
- *iAPX 286 Programmer's Reference Manual*, Copyright 1985, Intel Corp., Santa Clara, CA.

Contents

Chapter 1. Introducing IBM C/2 1-1
How This Book Is Organized 1-1
Conventions Used In This Book 1-3
Hexadecimal Representation 1-4
Operating Systems 1-4
Features and Functions 1-5
About Standards 1-6

Chapter 2. Installation and Practice Session 2-1
About the SETUP and INSTAID Programs 2-1
Space Required for Installation 2-4
Verifying Installed Options 2-4
Installing with SETUP 2-5
Installing with INSTAID 2-5
Environment Variables 2-6
System Configuration 2-7
Verifying Compiler Environment 2-7
Building Special Libraries (SETUP /L) 2-8
Practice Session 2-9
Compiling and Linking 2-10
Running the Demo Program 2-15
Using Batch Files 2-16
Understanding IBM C/2 Software 2-17
Executable Files 2-17
Include Files 2-18
Include \SYS Files 2-19
Library Files 2-19
Other Files 2-21
Manually Installing on Other Storage Devices 2-22
Setting the Environment 2-29
Setting the Configuration 2-29
Building Combined Libraries 2-30
Using a Numeric Coprocessor 2-30
Using an 80186, 80188, 80286 or 80386 Processor 2-30
Installing IBM C/2 On Local Area Network 2-31

Chapter 3. Building a C Program 3-1
Character Sets 3-1
Constants 3-6

Identifiers 3-12
Keywords 3-14
Comments 3-14
Tokens 3-16

Chapter 4. Program Structuring 4-1
Source Program 4-1
Source Files 4-2
Running the Program 4-4
Lifetime and Visibility 4-5
Naming Classes 4-9

Chapter 5. Declaring Variables, Functions, and Data Types . . . 5-1
Type Specifiers 5-2
Range of Values 5-5
Declarators 5-8
Pointer, Array, and Function Declarators 5-8
Complex Declarators 5-9
Declarators with Special Keywords 5-12
Variable Declarations 5-18
Simple Variable Declarations 5-19
Enumeration Declarations 5-19
Structure Declarations 5-21
Bit-fields 5-22
Union Declarations 5-26
Array Declarations 5-27
Pointer Declarations 5-31
Function Declarations 5-32
Storage Classes 5-35
Variable Declarations at the External Level 5-36
Variable Declarations at the Internal Level 5-39
Function Declarations at the External and Internal Levels . . 5-41
Initialization 5-42
Simple and Pointer Types 5-42
Aggregate Types 5-43
String Initializers 5-45
Type Declarations 5-45
Structure, Union, and Enumeration Types 5-46
Typedef Declarations 5-47
Type Names 5-48

Chapter 6. Forming Expressions and Making Assignments 6-1
Operands 6-1

Constants 6-2
Identifiers 6-2
Strings 6-3
Function Calls 6-3
Subscript Expressions 6-4
Multidimensional Array References 6-5
Member Selection Expressions 6-7
Expressions with Operators 6-8
Expressions in Parentheses 6-9
Type-Cast Expressions 6-9
Constant Expressions 6-9
Operators 6-10
Standard Arithmetic Conversions 6-10
Unary Operators 6-12
Indirection and Address-of Operators 6-13
Sizeof Operator 6-14
Multiplicative Operators 6-14
Additive Operators 6-15
Shift Operators 6-18
Relational Operators 6-19
Bitwise Operators 6-20
Logical Operators 6-21
Sequential Evaluation Operator 6-22
Conditional Operator 6-23
Assignment Operators 6-24
lvalue Expressions 6-24
Unary Increment and Decrement 6-25
Simple Assignment Operator (=) 6-26
Compound Assignment Operators 6-26
Precedence and Order of Evaluation 6-27
Side Effects 6-30
Type Conversions 6-31
Assignment Conversions 6-32
Type Cast Conversions 6-38
Operator Conversions 6-39
Function-Call Conversions 6-39

Chapter 7. Using C Statements 7-1
break 7-2
Compound Statement 7-3
continue 7-4
do 7-5
Expression Statement 7-6

for 7-7
goto and Labeled Statements 7-9
if 7-10
null 7-12
return 7-13
switch 7-15
while 7-18

Chapter 8. Using IBM C/2 Functions 8-1
Function Definitions 8-1
Storage Class 8-2
Return Type 8-3
Parameters 8-4
Function Body 8-9
Function Declarations 8-9
Function Calls 8-11
Arguments 8-14
Using Functions with a Variable Number of Arguments 8-16
Using Recursive Functions 8-18

Chapter 9. Using Preprocessor Directives and Pragmas 9-1
Manifest Constants and Macros 9-2
#define Define Directive 9-3
Special Operators Used Within Macros 9-6
#undef Undefine Directive 9-9
#include Include Files 9-10
Conditional Compiling 9-11
#error Error Directive 9-12
#if, #elif, #else, #endif If, Else-if, Else, and End-if Directives . . 9-13
#ifdef, #ifndef Ifdef and Ifndef Directives 9-17
#line Line Control 9-18
#pragma 9-20

Appendix A. Differences from the Proposed ANSI Standard for C A-1

Appendix B. Compiler, Linker and Run-Time Limits B-1
Compiler Limits B-1
Linker Limits B-2
Run-Time Limits B-4

Glossary X-1

Index X-13

Summary of Changes

Following are the differences between IBM C/2 Version 1.00 and Version 1.10 that affect this book.

Technical Changes

New Preprocessor Features
Under this version, new preprocessor features:

- Allow arguments in macro expansions to be expanded into a string literal containing the expanded argument.
- Join the tokens on either side of the operator into a new token in macro expansions.
- Allow production of user error messages during compiling.
- Allows single line comments.

There are also three new predefined macros: __DATE__, __TIME__, and __STDC__.

Changes to the Language Syntax
Under this version, changes made to the language syntax make it conform more closely to the new ANSI standard. This includes new pragmas, new object type modifiers, the **unary plus** operator and the **const** keyword.

Installation Procedure Modifications
Under this version, the automatic installation programs build combined libraries and an installation log file.

Organizational Changes

The Installation and Practice Session information has been moved to this book from *IBM C/2 Compile, Link, and Run*.

Chapter 1. Introducing IBM C/2

The C language is a general-purpose programming language known for its efficiency, economy, and portability. In many cases, C programs are comparable in speed to assembler-language programs and are easier to maintain and read. You can write portable code because the strict definition of the language makes it independent of any particular operating system or machine, or you can add system-specific routines to take advantage of the efficiencies of a particular machine. While these advantages make it a good choice for almost any kind of programming, C is especially useful in systems programming. The books in the IBM C/2™ library describe over 300 functions that perform useful tasks in your programs.

C is different from most structured languages. It does not include built-in functions to perform tasks such as input and output, reservation of storage, screen manipulation, and process control. Instead, C provides run-time libraries to perform such tasks. This design contributes to the adaptability and compactness of C. Run-time routines provide support as needed, which lets you minimize their use or tailor them for special purposes.

The books in this library show you how IBM C/2 resembles what you have learned about C from programming texts, where IBM has extended the C language to increase its usefulness, and where the IBM version differs from other versions of C.

How This Book Is Organized

Chapter 1, Introducing IBM C/2: Describes the notation conventions used in this book and some of the features and functions of IBM C/2.

C/2 is a trademark of the International Business Machines Corporation.

Chapter 2, Installation and Practice Session: Explains how to install the IBM C/2 compiler using the automatic installation programs provided or a Local Area Network Installation Aid. It provides instructions for compiling, linking, and running a sample C program and for using batch files to set up the operating environment.

Chapter 3, Building a C Program: Discusses differences between the C character set and the representable character set. It also discusses the rules for coding constants, identifiers, keywords, and tokens.

Chapter 4, Program Structuring: Discusses the major parts of a C program: preprocessor directives, declarations, the **main( )** function, and called functions.

Chapter 5, Declaring Variables, Functions, and Data Types: Discusses fundamental data types of variables as well as structures, arrays, and unions of the fundamental types. It also explains how to create new data type names using the **typedef** declaration.

Chapter 6, Forming Expressions and Making Assignments: Explains the format of expressions and assignment statements. These are the basis of C processing.

Chapter 7, Using C Statements: Discusses the forms of C statements. C statements control the flow of program processing.

Chapter 8, Using IBM C/2 Functions: Explains how to create and use a function, the primary unit of organization of a C program.

Chapter 9, Using Preprocessor Directives and Pragmas: Discusses the directives you use to control the action of the preprocessor.

Appendix A, Differences from the Proposed ANSI Standard for C: Contains information about the differences between the proposed ANSI standards and those used by IBM C/2.

Appendix B, Compiler, Linker and Run-Time Limits: Contains information about the limits imposed by C/2 .

Glossary: Includes the glossary for all books in the IBM C/2 library.

Index: Includes index entries for this book only.

Conventions Used In This Book

This book uses certain conventions in defining operating system commands, formats of functions, names, and terms.

Convention	Meaning
Boldface	Words or numerics printed in **bold** indicate procedural tasks, menu items, directives, function calls, library functions, statements, keywords, and values.
Italics	Words or numerics printed in *italics* represent information you supply, such as variables and filenames. Italics also introduce new terms or concepts.
Uppercase	Words printed in CAPITAL letters include DOS commands, OS/2 commands, dialog commands, options, programs, filenames, libraries, and utilities.
Color	Color indicates screen responses and programming examples.
Ellipses	Ellipses (...) indicate that you supply additional information in the form shown.
Brackets	Brackets [] indicate optional items supplied to commands.
Vertical Bars	Items separated by a vertical bar (|) mean that you can enter either one of the separated items. For example: `ON|OFF` means you can enter ON or OFF but not both.

The following terms have the specified reference:

Term	Reference
LINK	IBM Linker/2, Version 1.10
LIB	IBM Library Manager/2, Version 1.10
MAKE	IBM MAKE/2, Version 1.10
CodeView[1]	IBM CodeView
EXEMOD	IBM EXEMOD/2 Version 1.10
Assembler	IBM Macro Assembler/2.

Hexadecimal Representation

This book represents hexadecimal numbers in two ways. The letter **H** (or **h**) shows hexadecimal system calls, such as **59H** (or **59h**), in DOS. All other hexadecimal numbers use the standard C representation **0x***hexdigits*, such as **0x1F**.

Operating Systems

Throughout these books, the references to operating systems have the following meaning:

Abbreviation	Meaning
DOS	DOS 3.30 or 4.00
DOS mode	DOS or the DOS mode of OS/2
OS/2	IBM Operating System/2™.

---

[1]CodeView is a trademark of the Microsoft Corporation.

Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

Features and Functions

C is a flexible language that leaves much of the decision-making to you. Since C imposes few restrictions in matters such as type conversion, you need to understand the language well to understand the behavior of your programs. The C language:

- Provides a full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- Offers a large set of operators that correspond to common machine instructions, allowing direct translation into machine code. Various operators let you specify multiple operations with minimal code.
- Provides data types that include several sizes of integers as well as characters and single- and double-precision floating-point types. You can design more complex data types, such as arrays, structures, and unions to suit specific program needs.
- Lets you declare pointers to variables and functions. A pointer to an item corresponds to the machine address of that item. Using pointers can increase program efficiency. You can use pointer arithmetic in C.
- Gives you a preprocessor that acts on the text of files before compiling. Among the most useful applications of preprocessor directives for C programs are the definition of program constants, the substitution of function calls with faster macro look-alikes, and conditional compiling.

About Standards

This release of IBM C/2 follows most of the features of the draft proposed American National Standards Institute (ANSI) standard for C.

This release of IBM C/2 also participates in the Common Programming Interface component of IBM Systems Application Architecture. In these publications, features of IBM C/2 which are non-standard or non-portable are marked in a box labeled "IBM Extension."

For a complete listing of the IBM C/2 non-ANSI features see Appendix B, "Compiler, Linker and Run-Time Limits"

Chapter 2. Installation and Practice Session

IBM C/2 provides two automatic installation programs, SETUP and INSTAID. Both programs:

- Are menu-driven; they prompt you for various installation options and provide an explanation of each option as it is displayed.
- Install the IBM C/2 compiler to run under DOS mode and OS/2 mode and to produce programs for either mode.
- Copy the files necessary to run the compiler into directories they create according to your responses.
- Set the environment so the operating system can find these files whenever you call the compiler.
- Build run-time libraries that you can link with your programs.
- Create a log file listing the options you choose during installation.

If you have special needs not covered by SETUP and INSTAID, you can make the directories and copy the files yourself using the MKDIR and COPY commands. The information and file tables in "Understanding IBM C/2 Software" on page 2-17 can guide you through such an installation. Specific instructions are also found there on how to set the environment and the CONFIG.SYS file.

The practice session (see "Practice Session" on page 2-9) demonstrates basic procedures to compile, link, and run a sample C program.

To install C/2 as an application under PC Local Area Network, see "Installing IBM C/2 On Local Area Network" on page 2-31.

About the SETUP and INSTAID Programs

The information provided here is similar to that displayed on the screen by the SETUP and INSTAID programs. Read the information here first or read it on the screen as you go through the automatic installation process. See "Installing with SETUP" or "Installing with INSTAID" on page 2-5 to read about starting the automatic installation programs.

What SETUP and INSTAID Do

The SETUP and INSTAID programs install the compiler, supporting libraries, utility programs, and other files. They can also build combined libraries without going through the entire installation process.

The SETUP and INSTAID programs operate as full screen interactive programs. They prompt for the directories on your fixed disk where you wish to copy various types of files into. Each prompt has a default value presented in brackets ([]). To use the default value, press Enter.

Type a response at the cursor to override the default. As you type, the programs check each character to see if it is valid. An invalid character is not accepted and causes the speaker to beep. When you finish entering the response, the programs check for valid input. If the input is not valid, hear a beep and an error message is displayed at the bottom of the screen.

After you have entered responses to all items, the SETUP program asks if you would like to change any of the responses. If you accept the default (Y), SETUP returns to the first question. The responses you gave the first time are now the default values. You can move quickly through the questions to review what you entered, making changes if necessary. (With INSTAID, you can press the Esc key to review the prompts you have already answered as you move through the program.)

If you enter N, the programs prompt for the following names :

- Binary directory/ies: INSTAID prompts for only one but SETUP prompts for the following:
  - Bound: Contains bound utilities, including the compiler and linker. *Bound* means that they can run under DOS mode and OS/2 mode.
  - OS/2 mode binary directory: Contains run-time modules and utilities that run only under OS/2 mode. Prompts for this appear only if you request OS/2 mode support.
  - DOS mode binary directory: Contains run-time modules and utilities that run only under DOS mode. Prompts for this appear only if you request DOS mode support.
- Library directory: Contains C-library files, to hold the combined libraries built by SETUP or INSTAID.

- Include directory: Contains C-include files for the regular C environment and a subdirectory for the multi-thread support library.
- Source directory: Contains the sample C files, the startup sources, and the CodeView tutorial.
- Temporary directory: Contains work files created by the compiler.

CAUTION:
SETUP and INSTAID overwrite any files in an existing directory that contains files with the same names as the ones being copied. SETUP and INSTAID create the directories if they do not exist.

When all of the directory prompts have been answered, SETUP asks again if you want to change any of the responses. Entering Y returns you to the first directory prompt. Your last responses now appear as the new defaults. (With INSTAID, you can press the Esc key to review the prompts you have already answered as you move through the program.)

When you specify N, indicating you do not want to change any of the options, the programs start copying files from the diskettes to your fixed disk. They ask you to insert each disk that they need and to press a key to signify that you are ready to start copying. They check that the proper disk has been inserted and start copying files. For each file copied, SETUP displays a message showing the directory the file was copied into; INSTAID does not.

Component Libraries
Using combined libraries reduces program link time considerably. After loading the appropriate files from the C disk set, the programs use the information you specified about the math package(s), target operating system(s), and memory model(s) to create one or more combined libraries.

When the programs finish building the requested combined libraries, they delete the component libraries (those components used to build the combined libraries) if you specified that option in your earlier responses. Normally, you will not need the component libraries and should delete them since they occupy significant disk space.

Space Required for Installation

The following table lists the amount of disk space you will need to install the files and libraries available with the IBM C/2 compiler. To estimate how much storage you will need, add the value for each option you are choosing. Verify that you have the required disk space available before you begin; SETUP and INSTAID stop installation if you don't have enough.

Type of file	Required disk space
.EXE	1.077MB for bound executables 244KB for DOS mode executables 238KB for OS/2 mode executables.
Standard Include	100KB
Multi-thread Include	88KB
Source (optional)	27KB for DLL 125KB for SAMPLE 78KB for STARTUP 57KB for DOS 63KB for OS/2
Library	401KB for DLL Support 200KB for MT Support 63KB for Graphics.lib Approximately 200KB each for Combined

Verifying Installed Options

SETUP and INSTAID create a file named C2INSTAL.LOG that lists the options you chose during installation. This file resides in the binary directory you specify during installation. (The default is \IBMC2\BIN).

Installing with SETUP

Note: SETUP installs the compiler under DOS mode and OS/2 mode. When installation is complete the compiler runs under both modes and generates programs for either mode.

To use SETUP:

1. Place the diskette labeled SETUP in drive A.
2. Make sure the current directory is A:\.
3. At the command prompt type:

   SETUP *driveletter*

   where *driveletter* specifies the fixed disk used to start the operating system. If you start the system with a diskette, *driveletter* can refer to any fixed disk on your system.
4. Press Enter.
5. Follow the instructions on the screen.

Note: When installation is complete you can verify the options you chose by checking the C2INSTAL.LOG file in the binary directory you specified during installation. (The default is \IBMC2\BIN).

Installing with INSTAID

Note: To install the compiler with INSTAID you must be running under OS/2 mode. However, when installation is complete the compiler runs under DOS mode or OS/2 mode and generates programs for either mode.

To use INSTAID:

1. Place the diskette labeled CVP in drive A.
2. Make sure the current directory is A:\.
3. At the command prompt type:

   CINSTAID *driveletter*

   where *driveletter* represents the device (usually C) used to start OS/2.
4. Press Enter.
5. Follow the instructions on the screen.

Note: When installation is complete you can verify the options you chose by checking the C2INSTAL.LOG file in the binary directory you specify during installation. (The default is \IBMC2\BIN).

Environment Variables

To access the temporary, binary, include, and library directories that you installed, the environment variables must be set. As you go through the installation process, SETUP and INSTAID offer you two options:

1. **Set the environment when SETUP or INSTAID is finished.**
   - Choose Y (the default) if you want SETUP or INSTAID to set the environment for you.

     Note: The environment will not be set unless you have enough environment space. If you receive an environment space error message use the following configuration command in CONFIG.SYS to create an environment space of 1024 bytes:

     ```
     shell=command.com /p /e:1024
     ```

     You must restart your system after you add this line.

     See the technical reference information for DOS if you need more information about environment space.
   - Choose N if you are not using the compiler immediately after installation.
2. **Modify the automatic batch files.**
   - Choose N (the default) if you do not want SETUP or INSTAID to modify your automatic batch files. To set the environment, run NEW-VARS.BAT (for DOS mode) or NEW-VARS.CMD (for OS/2) after the command prompt.

     The NEW-VARS.BAT and NEW-VARS.CMD files reside in the appropriate binary directory specified earlier in the installation process. From those directories, these files can be run directly from the command prompt as batch files.

     Note: Using NEW-VARS.BAT or NEW-VARS.CMD adds to your existing environment strings; it does not replace them.

   - Choose Y to cause SETUP or INSTAID to add lines to AUTOEXEC.BAT (for DOS mode) or OS2INIT.CMD (for OS/2

Version 1.00). This will automatically set the environment variables each time you start your system.

Notes:

a. SETUP and INSTAID will not modify automatic batch files under DOS Version 4.00 or OS/2 Version 1.10.

b. This option will not work on all automatic batch files, such as those that end with a call to a program from which they never return. You should be familiar with the automatic batch file(s) on the system before choosing this option.

System Configuration

If you are using DOS, your start-up configuration should be set to provide enough buffers and files to run the C compiler. (See "Setting the Configuration" on page 2-29 for more information.) SETUP places the appropriate commands in the NEW-CONF.SYS file. This file resides in the same binary directory as the NEW-VARS.BAT file. If you have a CONFIG.SYS file on your root directory, copy the commands in NEW-CONF.SYS into it. If you do not, rename NEW-CONF.SYS to CONFIG.SYS and copy it to the root. If you change or create the CONFIG.SYS file, you must restart the computer to execute the commands in the file before using the compiler.

If you are using OS/2, your system should be set to have enough available files to open the C runtime libraries. To do this, include the following line in your CONFIG.SYS file:

```
FILES=255
```

If you plan to run the CodeView debugger in OS/2 mode, include the following line in your CONFIG.SYS:

```
IOPL=YES
```

Note: After you make changes to CONFIG.SYS you must restart your system for them to take effect.

Verifying Compiler Environment

Before running the practice session or another program, verify that the compiler environment is set correctly. To do this, type

```
SET
```

When you issue the SET command without an argument, it lists all environment variables and their current settings. Make sure the PATH, INCLUDE, LIB, and TMP variables are in the list and that they include the directories you installed the compiler file groups into. For example:

```
PATH=C:\IBMC2\BIN;
INCLUDE=C:\IBMC2\INCLUDE;
LIB=C:\IBMC2\LIB
TMP=C:\IBMC2\TMP
```

Building Special Libraries (SETUP /L)

After installing all of the files on your disk, you may later realize that you wanted a different set of combined libraries built. You can build the libraries yourself using LIB and the list of libraries given in the C Compiler documentation, or you can have SETUP build the libraries. If you want SETUP to build the libraries for you, type the following command, then follow the instructions on the screen.

```
SETUP driveletter /L
```

where *driveletter* specifies the fixed disk used to start the operating system. If you start the system with a diskette, *driveletter* can refer to any fixed disk on your system.

By specifying /L to SETUP when you start it, SETUP creates combined libraries without doing a complete installation of the product. If you need to load the files from disk, SETUP copies only the files that it needs to create the libraries that you specify with the math, memory model, and operating mode options.

In all aspects but one, it works the same as SETUP; the difference is the source of the files. If you chose to retain the component libraries when you originally installed IBM C/2, they would already be on your fixed disk. Therefore, when you give the /L option, SETUP allows the source of the files to be a directory specification. This is the directory the files can be found in and can be the same directory you place the resulting libraries into.

If SETUP cannot find a file in the directory you named, it prompts you to enter a new directory. If the file no longer exists on your fixed disk, you can specify a diskette drive, and SETUP will read the files from the original distribution disks.

Note: Once you start reading from a diskette, you cannot specify a directory again.

Practice Session

This practice session shows how to create an executable program using either the CL command or the CC and LINK commands separately. These instructions cover the options you need to select in order to operate under each of the different library modules. They also give the options that determine whether the program runs under DOS mode or OS/2 mode. These commands work in either DOS or OS/2 mode; that is, in either mode, you can create an executable program to run under either mode.

The source file used in this practice session is called DEMO.C. It is in the directory you chose for source files when you installed the compiler. DEMO.C is a simple C program that contains only one function, the **main** function. The **main** function is designed to display any command-prompt arguments you pass to the program at run time. It also displays the current environment values. You can examine the DEMO.C source file to see how it accomplishes this. For a full description of passing command-prompt data to programs, getting access to the program environment from within a program, and declaring the *argc*, *argv*, and *envp* parameters, see the "Passing Data to a Program" section in Chapter 4 of *IBM C/2 Compile, Link, and Run*.

Start the operating system and change the current directory to:

`C:\dest\src`

C	The current drive.
\dest	The directory you chose to contain the compiler directories. Omit this if *\dest* is the root directory.
\src	The directory you chose to contain source files when you installed the compiler.

The compiler allows you to compile and link a program in one or two steps. The CL command compiles and links in one step. The CC and LINK commands allow you to perform each operation separately. The practice session uses both methods.

If you get an error message during the practice session, see Appendix A of the *IBM C/2 Compile, Link, and Run* for suggestions on how to recover from the error condition.

Note: IBM C/2 is sensitive to uppercase and lowercase. When entering commands, type them as they appear.

Compiling and Linking

During this practice session you create an executable program using either the CL command or the CC and LINK commands.

Using the CL Command

The following shows what to type to compile and link the DEMO program:

```
CL [/Ax] [/Lmode] DEMO.C
```

where

/A*x* indicates the memory model you are operating under.

If you are using the Small-model, leave /A*x* out. If you are using any other library, replace *x* with the variable for the library model you have installed:

Variable	Installed Library
M	Medium-model
C	Compact-model
L	Large-model.

/L*mode* indicates the mode the executable program will run under.

If, during the SETUP process, you chose to run programs under only one mode, do not use the /L option. Use the /L option only if:

a. You did not choose a default operating mode or

b. You chose a default operating mode but want to create a program to run under the other mode.

Replace *mode* with the variable for the mode you want the executable program to run under:

Variable	Mode
p	OS/2 mode
c	DOS mode.

Press Enter after typing the correct information. The CL command calls the compiler executable files that create the object file and link it with the libraries that you have specified in order to create the executable program. No prompts are displayed, only compiler and linker messages. When the command prompt returns, the Demo program is ready to run.

For more information, see "Running the Demo Program" on page 2-15.

Using the CC and LINK Commands

This section describes the two-step compile and link method.

1. Type

   ```
   CC
   ```

   and press Enter.

   The CC command calls the compiler control program CC.EXE, which displays prompts and guides you through the compiling process. After the compiler copyright statements appear, the first prompt displayed is:

   ```
   Source Filename [.C]:
   ```

   This prompt asks for the name of the file to be compiled. If you do not include the filename extension, CC.EXE assumes that the extension is .C (or .c).

2. Select the correct DEMO information, depending on the library you installed, from the following list:

Type	Installed Library
DEMO.C	Small-model
/AM DEMO.C	Medium-model
/AC DEMO.C	Compact-model
/AL DEMO.C	Large-model.

3. Type the correct information and press Enter.

   The next prompt is:

   ```
   Object filename[DEMO.OBJ]:
   ```

   Following the Object filename prompt you can:

   a. Press Enter. This causes CC.EXE to use the default name for the object file: DEMO.OBJ. The object file is created in the current working directory.

b. Supply a name for the object file.

4. Press Enter after typing the correct information.

   The next prompt is:

   ```
   Source listing [NUL.LST]:
   ```

   This prompt lets you create a listing of your source file. The source listing contains your source code, on numbered lines, and symbol table information. Any error messages that occur during compiling are shown in the source listing immediately following the line that caused the error.

5. Type

   ```
   DEMO
   ```

   and press Enter.

   The listing file DEMO.LST is created in the current working directory.

   The next prompt is:

   ```
   Object listing [NUL.COD]:
   ```

   This prompt lets you create a listing of your object file containing the machine instructions that correspond to your C instructions.

6. Type

   ```
   DEMO
   ```

   and press Enter.

   CC.EXE adds the default extension .COD to the name DEMO and creates a listing file named DEMO.COD. The listing file is created in the current working directory.

   CC.EXE now begins to compile your program. If your program has errors, they are displayed as the compiler operates. (DEMO.C does not have errors.) When the compiling process is finished, the command prompt is displayed.

   You now have an object file named DEMO.OBJ, a source listing file named DEMO.LST, and a listing file named DEMO.COD in your current working directory.

7. To link your file, type

   ```
   LINK
   ```

   and press Enter.

After the linker copyright statements, the first linker prompt on the screen is:

```
Object Modules [.OBJ]:
```

8. Type

   ```
   DEMO
   ```

   and press Enter.

   The LINK program adds the .OBJ extension to your object module files so you can easily locate them on the disk. Because the file is in the current working directory, you do not have to specify a pathname for DEMO.

   The next prompt is:

   ```
   Run File [DEMO.EXE]:
   ```

   This prompt lets you name the executable program file.

9. Type

   ```
   /NOI
   ```

   and press Enter.

   The linker uses the default name, shown in brackets, for the executable file. The executable file is created in the current working directory.

   The /NOI stands for the linker option NOIGNORECASE. Although not necessary for the DEMO program, it is recommended that you use this option for compatibility. Many compilers distinguish lowercase from uppercase in identifiers and assume the linker does the same.

   The next prompt is:

   ```
   List File [NUL.MAP]:
   ```

   If you enter a filename for this prompt, the linker creates a map file that lists all the external symbols in the program and their locations.

10. Type

    ```
    DEMO/MAP
    ```

    and press Enter.

    This response tells the linker to create a listing file named DEMO.MAP. The map file is created in the current directory. The

/MAP option causes global symbols to be listed at the end of DEMO.MAP.

The next prompt is:

```
Libraries [.LIB]:
```

Following the Libraries prompt you can:

a. Press Enter to accept the default library name. Do this if you chose to run programs under only one mode during the SETUP process.
b. Enter

   /NOD *pathname*:*x*LIBC*yz*

   if you

   1) Did not choose a default operating mode.

   2) Chose a default operating mode but want to create a program to run under the other mode.

Replace *x* with the variable for the library model you have installed:

Variable	**Installed Library**
S	Small-model
C	Compact-model
M	Medium-model
L	Large-model.

Replace *y* with the variable for the math library you have installed:

Variable	**Installed Library**
A	Altmath
E	Emulator
7	8087 floating point.

Replace *z* with the variable for the mode you want the executable file to run under.

Variable	**Mode**
R	DOS mode
P	OS/2 mode.

11. Press Enter after typing the correct information.

    The /NOD (NODEFAULT) switch tells the linker to link with the next specified library, not the default (*x*LIBC*yz*.) The default

library name exists if you want to always create executable files to run under only one mode. To do this, rename your *x*LIBC*yz*.LIB to *x*LIBCE.LIB. LINK and CL link with it by default, without any library specification.

The last prompt is:

```
Definitions File [NUL.DEF]:
```

This prompt allows you to define specific attributes of the application.

12. Press Enter.

    The LINK program now proceeds to link your file, displaying any errors. When the command prompt returns, the linker has finished processing your file. You now have an executable file named DEMO.EXE in your working directory, plus a map file named DEMO.MAP.

    You may want to examine the source listing (DEMO.LST), the object listing (DEMO.COD), and the map file (DEMO.MAP) to familiarize yourself with their formats. These files are useful for debugging programs. However, the listing and map files are not required for running the program, so you can delete them.

    You can also delete the object file (DEMO.OBJ) because you have the executable program file. Chapter 5 in *IBM C/2 Compile, Link, and Run* explains how to use the IBM LIB program to organize object files into libraries of useful functions.

Running the Demo Program

Run the DEMO program and pass three arguments by typing the following at the command prompt:

```
DEMO ONE TWO THREE
```

The program name is displayed on the screen, followed by the arguments ONE, TWO, and THREE, and a listing of all current environment settings. The environment settings include PATH, LIB, INCLUDE, and TMP, as well as any other settings that are currently in effect (whether or not they apply to the C program or to the compiling and linking process).

Using Batch Files

You can create batch files to set up the compiler environment and call the compiler. A batch file is a text file containing a series of executable commands. Under DOS mode, batch files have the extension .BAT. In OS/2 mode, batch files have the extension .CMD. Run a batch file by typing the filename without the .BAT or .CMD extension. This causes the system to execute the commands in the file. For more information about creating and using batch files see the operating system reference information.

The examples that follow use the command-prompt method of calling CC and LINK. The command-prompt method lets you give all responses to the prompts on a single line instead of waiting for the individual prompts. The command-prompt method is discussed under "Using the Command Prompt" in Chapter 2 and "Using a Command Prompt to Specify Link Files" in Chapter 3 of *IBM C/2 Compile, Link, and Run*.

For example, if you have installed the IBM C/2 compiler on drive C, and the destination directory is \C, you can use the following batch file to set the correct environment and then compile and link a given source file.

```
PATH=C:\IBMC2\BIN;
SET INCLUDE=C:\IBMC2\INCLUDE;
SET LIB=C:\IBMC2\LIB
SET TMP=C:\IBMC2\TMP
CC %1,,,;
LINK %1,/NOI,%1,/NOD:SLIBCE SLIBCER;
```

The values given to PATH, INCLUDE, LIB, and TMP set the environment for the CC and LINK commands. As given, these set commands will replace the previous environment strings.

The symbol %1 tells DOS to look for an argument on the command-prompt line when you run the batch file. When you type

```
COMPILE DEMO
```

at the prompt, the filename DEMO is substituted for %1, and DEMO.C is compiled, producing the object file DEMO.OBJ.

The last line links the object file to the library, assuming that you have chosen not to rename the DOS mode library to SLIBCE.LIB (the default.) The CC program returns an exit code to allow testing for successful compiling. The exit code 0 indicates success. For infor-

mation on the other codes, see "Compiler Exit Codes" in Chapter 2 of *IBM C/2 Compile, Link, and Run*. The DOS batch command IF ERRORLEVEL can be used to test the exit code; see either the user's reference information for the operating system. for more information on this command.

The following lines could replace the LINK command in the previous batch file. (The @ tells the system not to echo the command itself to the screen.)

```
@IF ERRORLEVEL 1 GOTO FAILED
LINK %1,/NOI,%1,/NOD:SLIBCE SLIBCER;
@GOTO END
:FAILED
@ECHO Compile of %1.C failed
:END
```

If compiling is successful, the object file DEMO.OBJ is linked to produce DEMO.EXE (the default name, since none is supplied). The name DEMO is also supplied (by means of the symbol %1) for the Map file prompt, so a map file named DEMO.MAP is produced. If compiling is not successful, the LINK step is skipped and an explanation is displayed.

The environment set by the COMPILE batch file remains in effect until you explicitly change it or until you restart your machine. To restore your usual environment settings, create a batch file that resets the environment variables to the directories you most frequently use, and call it NEW-VARS.BAT or NEW-VARS.CMD.

Understanding IBM C/2 Software

The software for IBM C/2 consists of three main file categories: compiler executable files, include files, and library files. Additional files that do not fall into the three main categories are discussed separately under "Other Files" on page 2-21.

Executable Files

The executable files allow you to compile and link your program. They should be in the \BIN directory.

Compiling: CC.EXE runs the compiler. Call it by typing `CC` at the command prompt. The three stages, or passes, of the compiler are C1.EXE, C2.EXE, and C3.EXE. They run in order when the compiler program is processing a file.

CL.EXE starts the compiler, and then links the object file to the appropriate library. Call the CL.EXE by typing `CL` at the command prompt. See "Compiling and Linking in One Step Using the CL Command" in Chapter 3 of *IBM C/2 Compile, Link, and Run* for more information.

Linking: The file LINK.EXE is the IBM object linker. You call the linker by typing `LINK` after you have compiled a file or files. The linker produces an executable program file from your compiled files. See Chapter 3 in *IBM C/2 Compile, Link, and Run* for more information.

Others: The files CV.EXE and CVP.EXE are the CodeView symbolic debugger. The first file runs in DOS mode, the second file in OS/2 mode. For example, to debug DEMO.EXE, enter either `CV DEMO` (for DOS mode) or `CVP DEMO` (for OS/2.) For more information, see Chapter 1 in *Debug*.

You can use the library manager program LIB.EXE to create and organize libraries of object modules. To call this utility, type `LIB`. The EXEMOD.EXE utility modifies an executable program file. See Chapter 6 in *IBM C/2 Compile, Link, and Run* for more information about these utilities.

Include Files

Include files are text files you can incorporate into your program by using the C preprocessor directive **#include**. The include files are in the \INCLUDE directory. These files contain definitions used by run-time library routines. For more information about include files, see Chapter 4 in *IBM C/2 Language Reference*.

Include \SYS Files

Some include files are stored in a subdirectory named SYS under the \INCLUDE directory. The subdirectory contains some of the files that define system-level constants and types. When you use files from this subdirectory in a program, give the subdirectory name as well as the filename.

Note: When you installed IBM C/2, you chose a name for the subdirectory for the INCLUDE files. You either accepted the default \INCLUDE or specified another name. The SETUP and INSTAID programs create a subdirectory named SYS.

For example, having installed the compiler with SETUP or INSTAID, if you want to include the file **timeb.h**, put the following line in your program:

```
#include <sys\timeb.h>
```

Note that although case is significant in C programs, case is not significant to DOS. Both sys and SYS are acceptable when used as DOS directory names.

Library Files

Library files contain compiled run-time library routines to be linked with your program. Four sets of library files are included in the \LIB directory: small-model, medium-model, compact-model, and large-model. Each library set comes in two versions, one for DOS mode and the other for OS/2 mode. Huge-model programs use the large-model library files.

The terms small-model, medium-model, compact-model, large-model, and huge-model refer to standard storage models you can choose for your program, based on its storage requirements for code and data.

You do not have to choose a storage model to process and run your program. The small model is appropriate for most programs, and the compiler uses the small model and the small-model library files by default. For more information about storage models, see the "Working With Storage Models /A" section in Chapter 2 of *IBM C/2 Compile, Link, and Run*.

Two additional library files, EM.LIB and 87.LIB, can be used with all storage models. EM.LIB is the floating-point emulator used to

perform floating-point operations. 87.LIB is the floating-point library. This library provides minimal floating-point support and can be used only when a numeric coprocessor is present. These files were merged with the standard C libraries when the installation program created your combined libraries. The compiler uses the emulator by default, but you can cancel the default to use the floating-point library (if you have a coprocessor) or the alternate math library. Floating-point options are described in more detail in "Selecting the Floating-Point Options" and "Controlling Floating-Point Operations" in Chapter 2 of *IBM C/2 Compile, Link, and Run*.

The object file BINMODE.OBJ is provided for modifying the default mode for data files from text mode to binary mode. The same file can be used with all three storage models.

The SETARGV.OBJ file provides a routine that expands the DOS global filename characters (?) and (*) in filename arguments passed to C programs from the command prompt. Global filename expansion is performed only if you explicitly link with the SETARGV file. See "Expanding Global Filename Arguments" in Chapter 3 of *IBM C/2 Compile, Link, and Run* for more information.

The library files beginning with an S belong to the small-model library set. The small-model C run-time libraries are organized into two files:

SLIBCER.LIB = DOS version of C run-time libraries
SLIBCEP.LIB = OS/2 version of C run-time libraries

The SLIBCER.LIB library contains all C run-time library support you need for DOS or DOS mode. The SLIBCEP.LIB library contains all C run-time library support you need for OS/2 mode.

SLIBCER.LIB and SLIBCEP.LIB contain an object module named CRT0.OBJ, which is the operating system startup routine for small-model programs. The startup routine performs several important tasks. It reserves the stack for the program and initializes the segment registers. It sets up the *argv*, *argc*, and *envp* variables to allow command-prompt arguments and environment settings to be passed to the program. The startup routine is responsible for setting up and maintaining the operating environment for the program. The startup routine also initializes the emulator if the emulator is loaded. The assembler source code for the DOS startup routine is provided in the directory \STARTUP\DOS; the source code for the OS/2 startup

routine is in \STARTUP\OS2. You can create your own startup routines using the STARTUP.BAT file. For more information about creating your own startup routines, see Appendix B in *IBM C/2 Language Reference*.

SLIBFP.LIB is the floating-point math library. It is required whenever a program uses EM.LIB or 87.LIB.

SLIBFA.LIB is the alternate floating-point library. You can use SLIBFA.LIB instead of EM.LIB and SLIBFP.LIB when speed is more important than precision in floating-point calculations. See "Selecting the Floating-Point Options" and "Controlling Floating-Point Operations" in Chapter 2 of *IBM C/2 Compile, Link, and Run* for more information about this option.

The compiler places the names of the default combined library (SLIBCE.LIB) in every object file. LINK reads these names and links the program to them automatically.

The files beginning with an **M** are medium-model library files, the files beginning with a **C** are compact-model library files, and the files beginning with an **L** are large-model library files (also used with huge-model programs). The organization and content of these files are similar to the small-model library set. For example, CLIBCER.LIB, MLIBCER.LIB, and LLIBCER.LIB, like SLIBCER.LIB, contain a startup routine named CRT0.OBJ.

If you specify the medium-, compact-, large-, or huge-model when you process your program, the compiler uses the appropriate combined libraries (by default, *x*LIBCE.LIB) when placing information in the object file for the linker. Otherwise, the compiler uses the small-model files.

The library LIBH.LIB contains the long integer helper routines used by the compiler and run-times internally to do long integer arithmetic.

Other Files

The following files are optional.

Startup Files: These files are used to build the startup portions of the C run-time library. For more information see Appendix B in *IBM C/2 Language Reference*.

CodeView Files: Included in the SAMPLE directory is a CodeView tutorial, which demonstrates the various capabilities of the CodeView debugger.

Information Files: The file called README.DOC on the DRIVER diskette explains changes that were most recently made to the product.

Installation Files: Some files are used by the installation programs.

SETUP Program	INSTAID Program
SETUP.BAT	C11PIP.LIB
SETUP.CMD	C11.PIP
SETUPC2.EXE	CINSTAID.CMD

The IBM PC Local Area Network uses PROFILE.NIA, located in the root directory of the COMPILER disk.

Manually Installing on Other Storage Devices

This section has general information about installing the IBM C/2 compiler on devices other than fixed disks.

You need to create directories on a storage device so you can copy all the necessary files into them. It is recommended that you install the compiler into at least three separate directories and that you install the recommended files into each directory. The compiler will search the directories for those files. You can install the compiler on any number of devices as long as you can copy all the recommended files into the recommended directories. The three directories are:

\BIN	For compiler executable files
\LIB	For all the libraries
\INCLUDE	For all the include files.

You can create additional directories and install Source, Sample, and Startup files, but you do not need these to compile a program.

These tables list the files on each diskette and the directories they should be copied to. To install the compiler in some way not pro-

vided by the installation programs, use these tables as a guide to manually copy the files.

From SETUP diskette	To device:
C1.EXE	\BIN
EXEMOD.EXE	\BIN
LIB.EXE	
CV.HLP	
\SOURCE\DEMO.C	\SRC
\SOURCE\GRDEMO.C	
\SOURCE\SIEVE.C	

From Include diskette	To device:
LINK.EXE	\BIN
\STARTUP\FILE2.H	\SRC\STARTUP
\STARTUP\MSDOS.H	
\STARTUP\REGISTER.H	
\STARTUP\STARTUP.BAT	
\STARTUP\MAKEFILE	
\STARTUP\CHKSTK.ASM	
\STARTUP\CHKSUM.ASM	
\STARTUP\NULBODY.C	
\STARTUP\README.DOC	
\STARTUP\MSDOS.INC	
\STARTUP\CMACROS.INC	
\STARTUP\BRKCTL.INC	
\STARTUP\VERSION.INC	
\STARTUP\WILD.C	
\STARTUP_FILE.C	
\STARTUP\CRT0FP.ASM	
\STARTUP\FMSGHDR.ASM	
\STARTUP\SETARGV.ASM	
\STARTUP\DOS\NULBODY.LNK	\SRC\STARTUP\DOS
\STARTUP\DOS\STDENVP.ASM	
\STARTUP\DOS\CRT0.ASM	
\STARTUP\DOS\CRT0DAT.ASM	

From include diskette	To device:
\STARTUP\DOS\CRT0MSG.ASM	
\STARTUP\DOS\EXECMSG.ASM	
\STARTUP\DOS\NMSGHDR.ASM	
\STARTUP\DOS\STDALLOC.ASM	
\STARTUP\DOS\STDARGV.ASM	
\INCLUDE\ASSERT.H	\INCLUDE
\INCLUDE\BIOS.H	
\INCLUDE\CONIO.H	
\INCLUDE\CTYPE.H	
\INCLUDE\DIRECT.H	
\INCLUDE\DOS.H	
\INCLUDE\ERRNO.H	
\INCLUDE\FCNTL.H	
\INCLUDE\FLOAT.H	
\INCLUDE\GRAPH.H	
\INCLUDE\IO.H	
\INCLUDE\LIMITS.H	
\INCLUDE\MALLOC.H	
\INCLUDE\MATH.H	
\INCLUDE\MEMORY.H	
\INCLUDE\PROCESS.H	
\INCLUDE\SEARCH.H	
\INCLUDE\SETJMP.H	
\INCLUDE\SHARE.H	
\INCLUDE\SIGNAL.H	
\INCLUDE\STDARG.H	
\INCLUDE\STDDEF.H	
\INCLUDE\STDIO.H	
\INCLUDE\STDLIB.H	
\INCLUDE\STRING.H	
\INCLUDE\TIME.H	
\INCLUDE\SYS\LOCKING.H	\INCLUDE\SYS
\INCLUDE\SYS\STAT.H	
\INCLUDE\SYS\TIMEB.H	

From Include diskette	To device:
\INCLUDE\SYS\TYPES.H	
\INCLUDE\SYS\UTIME.H	
\LIB\EM.LIB	\LIB
\LIB\87.LIB	
\LIB\BINMODE.OBJ	
\LIB\SETARGV.OBJ	

From Compiler diskette	To device:
C1.ERR	\BIN
C1L.EXE	
C3.EXE	

From DRIVER diskette	To device:
CL.EXE	\BIN
CL.ERR	
CL.HLP	
C2.EXE	
C23.ERR	
CC.EXE	
CC.HLP	

From SMALL diskette	To device:
MAKE.EXE	\BIN
SLIBCR.LIB	\LIB
SVARSTCK.OBJ	
MLIBCR.LIB	
MVARSTCK.OBJ	

From LARGE diskette	To device:
GRAPHICS.LIB	\LIB
CLIBCR.LIB	
CVARSTCK.OBJ	
LLIBCR.LIB	

From LARGE diskette	To device:
LVARSTCK.OBJ	

From FLOAT diskette	To device:
SLIBFP.LIB	\LIB
MLIBFP.LIB	
CLIBFP.LIB	
LIBH.LIB	
LLIBFP.LIB	
SLIBFA.LIB	
MLIBFA.LIB	
CLIBFA.LIB	
LLIBFA.LIB	

From CodeView diskette	To device:
CV.EXE	\BIN
\SAMPLE\CODEVIEW.DOC	\SRC\SAMPLE
\SAMPLE\SAMPLE.BAT	
\SAMPLE\LIFE.C	
\SAMPLE\LIFE.EXE	
\SAMPLE\LIFE.R	
\SAMPLE\C_AUTO.CV	
\SAMPLE\E_AUTO.CV	
\SAMPLE\L_AUTO.CV	
\SAMPLE\M_AUTO.CV	
\SAMPLE\Q_AUTO.CV	
\SAMPLE\S_AUTO.CV	
\SAMPLE\RESPOND.COM	

From CVP diskette	To device:
\CVP\CVP.EXE	\BIN
\CVP\CVP.HLP	
\STARTUP\OS2\NULBODY.LNK	\SRC\STARTUP\OS2
\STARTUP\OS2\CRT0.ASM	

From CVP diskette	To device:
\STARTUP\OS2\STDENVP.ASM	
\STARTUP\OS2\CRT0DAT.ASM	
\STARTUP\OS2\CRT0MSG.ASM	
\STARTUP\OS2\EXECMSG.ASM	
\STARTUP\OS2\NMSGHDR.ASM	
\STARTUP\OS2\STDALLOC.ASM	
\STARTUP\OS2\STDARGV.ASM	
\STARTUP\OS2\S\CRT255.OBJ	\SRC\STARTUP\OS2\S
\STARTUP\OS2\S_FILE255.OBJ	
\STARTUP\OS2\M\CRT255.OBJ	\SRC\STARTUP\OS2\M
\STARTUP\OS2\M_FILE255.OBJ	
\STARTUP\OS2\C\CRT255.OBJ	\SRC\STARTUP\OS2\C
\STARTUP\OS2\C_FILE255.OBJ	
\STARTUP\OS2\L\CRT255.OBJ	\SRC\STARTUP\OS2\L
\STARTUP\OS2\L_FILE255.OBJ	

From DYNALINK diskette	To device:
\LIB\SLIBCP.LIB	\LIB
\LIB\MLIBCP.LIB	
\LIB\CLIBCP.LIB	
\LIB\LLIBCP.LIB	
\LIB\CDLLSUPP.LIB	
\LIB\LLIBCDLL.LIB	
\LIB\LLIBCMT.LIB	
\LIB\CDLLOBJS.LIB	
\LIB\CDLLOBJS.DEF	
\LIB\CDLLOBJS.CMD	
\LIB\CRTEXE.OBJ	
\LIB\CRTDLL.OBJ	
\LIB\CRTLIB.OBJ	
\LIB\APILMR.OBJ	
\LIB\CLM.CMD	\BIN
\LIB\EXAMPLE\MHELLO.C	\SRC\DLL
\LIB\EXAMPLE\MKMHELLO.CMD	

From DYNALINK diskette	To device:
\LIB\EXAMPLE\STMAIN.C	
\LIB\EXAMPLE\STMAIN.DEF	
\LIB\EXAMPLE\STDLL.C	
\LIB\EXAMPLE\STDLL.DEF	
\LIB\EXAMPLE\MKSTDLL.CMD	
\LIB\EXAMPLE\MTMAIN.C	
\LIB\EXAMPLE\MTMAIN.DEF	
\LIB\EXAMPLE\MTDLL.C	
\LIB\EXAMPLE\MTDLL.DEF	
\LIB\EXAMPLE\MKMTDLL.CMD	
\INCLMT\ASSERT.H	\INCLUDE\MT
\INCLMT\CONIO.H	
\INCLMT\CTYPE.H	
\INCLMT\DIRECT.H	
\INCLMT\DOS.H	
\INCLMT\ERRNO.H	
\INCLMT\FCNTL.H	
\INCLMT\FLOAT.H	
\INCLMT\IO.H	
\INCLMT\LIMITS.H	
\INCLMT\MALLOC.H	
\INCLMT\MATH.H	
\INCLMT\MEMORY.H	
\INCLMT\PROCESS.H	
\INCLMT\SEARCH.H	
\INCLMT\SETJMP.H	
\INCLMT\SHARE.H	
\INCLMT\SIGNAL.H	
\INCLMT\STDARG.H	
\INCLMT\STDDEF.H	
\INCLMT\STDIO.H	
\INCLMT\STDLIB.H	
\INCLMT\STRING.H	
\INCLMT\TIME.H	

From DYNALINK diskette	To device:
\INCLMT\SYS\LOCKING.H	\INCLUDE\MT\SYS
\INCLMT\SYS\STAT.H	
\INCLMT\SYS\TIMEB.H	
\INCLMT\SYS\TYPES.H	
\INCLMT\SYS\UTIME.H	

Setting the Environment

If you install the compiler by making all the directories and copying the files yourself, you must set the environment variables before you can compile and link programs. To learn about this, see "Environment Variables" on page 2-6.

Since you have not run SETUP, you do not have the files NEW-VARS.BAT and NEW-VARS.CMD. The commands in these files are:

```
PATH=C:\your_bin_dir(s);%PATH%
set LIB=C:\your_lib_dir;%LIB%
set INCLUDE=C:\your_include_dir;%INCLUDE%
set TMP=C:\your_temp_dir;
```

where *your x _dir(s)* means the directories you chose for the various compiler file groups.

Setting the Configuration

If you are going to run the compiler in DOS mode, see "System Configuration" on page 2-7. The commands in the NEW-CONF.SYS are:

```
files=20
buffers=10
```

If you are using OS/2 mode, add the following statement to your CONFIG.SYS so that the system will have the number of files you need to open the C runtime libraries:

```
FILES=255
```

Building Combined Libraries

To make the combined run-time libraries you must run LIB, the Library Manager, for each combination of memory model, math option, and addressing mode you expect to generate applications for. The conventional name for a combined library is *x*LIBC*yz*.LIB, where *x* is the memory model (S, M, C, or L), *y* is the math option (E, 7, or A), and *z* is the addressing mode (R or P).

To select component library files as input to the library manager do the following:

- Choose *x*LIBCz.LIB and LIBH.LIB.
- Choose one of the following depending on the math option:
  - EM.LIB and *x*LIBFP.LIB for emulation (*y*=E)
  - Choose 87.LIB and *x*LIBFP.LIB for coprocessor math (*y*=7)
  - *x*LIBFA.LIB. for alternate math (*y*=A)
- Choose GRAPHICS.LIB if you want the graphics functions (for DOS mode only- *z*=R).

Example

Use the following LIB command to produce the medium-model, alternate math, OS/2 mode combined library.

```
LIB MLIBCAP,MLIBCP+LIBH+MLIBFA;
```

Using a Numeric Coprocessor

For information on numeric coprocessors, see "Selecting the Floating-Point Options" in Chapter 2 of IBM *IBM C/2 Compile, Link, and Run*. With a numeric coprocessor, you can perform fast, efficient floating-point operations. To take maximum advantage of the capabilities of the coprocessor, select one of the 8087 options described in the "If You Have a Numeric Coprocessor" section in Chapter 2 of *IBM C/2 Compile, Link, and Run*.

Using an 80186, 80188, 80286 or 80386 Processor

To use the compiler with an 80186/80188, 80286 or 80386 processor enable the instruction set by using the **/G1** (with 80186/80188) or **/G2** (with 80286/80386) options when you compile a program. See "80186/80188, 80286 or 80386 Processors" in Chapter 2 of *IBM C/2 Compile, Link, and Run* for more information about these processors.

Installing IBM C/2 On Local Area Network

You can install IBM C/2 under the IBM PC Local Area Network Program (PC LAN) Version 1.21 or Version 1.30. You must first install DOS and the LAN programs into a structure of network directories. Refer to the IBM Local Area Network Program User's Guide for instructions on how to install applications.

For PC LAN 1.21, use the PC LAN Program Installation Aid application selection menu. Select **OTHER APPLICATION NOT LISTED BELOW** and insert the diskette labeled Compiler into Drive A.

For PCLAN 1.30, insert the diskette labeled Compiler into Drive A, then enter the following command to run the Application Installation Utility:

```
PCLPAIU
```

Follow the instructions on the screen.

After the C/2 files are copied to the network server, you can build combined run-time libraries. See "Building Combined Libraries" on page 2-30.

Chapter 3. Building a C Program

You use the elements of the C language, names, numbers, and characters, to build a C program. This chapter describes:

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens.

Character Sets

The character sets defined for C programs are the representable character set and the C character set. C programs can contain only characters from the C character set, but string literals, character constants, and comments can use any representable characters.

The *representable character set* consists of all letters, digits, and symbols that you can represent graphically with a single character. The extent of the representable character set depends on the type of display, console, or character device you use.

See Appendix C of *IBM C/2 Language Reference* for an ASCII code table of the representable character set.

The *C character set* is a subset of the representable character set. It consists of letters, digits, and punctuation marks with specific meanings to IBM C/2. You construct C programs by combining the characters of the C character set into meaningful statements. The compiler produces error messages when it finds characters not used correctly or not belonging to the C character set.

Letters and Digits

C uses the following letters and digits:

- Uppercase English letters:

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Lowercase English letters:

  a b c d e f g h i j k l m n o p q r s t u v w x y z

- Decimal digits:
  0 1 2 3 4 5 6 7 8 9

IBM C/2 treats uppercase and lowercase letters as distinct characters. This capacity is *case sensitivity*. If you specify a lowercase *a* in a C program item, you cannot substitute an uppercase **A** in its place. You must continue using the lowercase letter to refer to that program item. (Refer to the discussion of the linker in *IBM C/2 Compile, Link, and Run* for information about case sensitivity in external names.)

White-Space Characters
Space, tab, line-feed, carriage-return, form-feed, vertical tab, and newline characters are *white-space characters*. They serve the same purpose as the spaces between printed words and lines. These characters separate items within a program.

A Ctrl+Z character is an end-of-file indicator. The compiler disregards any text following the Ctrl+Z mark.

IBM C/2 ignores white-space characters unless they are separators, components of character constants, or string literals. You can use extra white-space characters to make a program readable.

Punctuation and Special Characters
You can use the punctuation and special characters in the C character set in many ways. You can use them to organize the text of a program or to define the tasks of the compiler or compiled program. The following table lists these characters:

Character	Name	Character	Name
,	Comma	|	Vertical line
.	Period		Space
;	Semicolon	/	Forward slash
?	Question mark	~	Tilde
'	Single quotation mark	_	Underscore
"	Double quotation mark	#	Number sign
(	Left parenthesis	%	Percent sign

Char-acter	Name	Char-acter	Name
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	–	Hyphen, minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	+	Plus sign
>	Right angle bracket	\	Backslash
!	Exclamation point		

This book describes how to use these special characters. You can also use punctuation characters in the representable character set that do not appear in this list.

Escape Sequences

Escape sequences are combinations of special characters. They represent blank and nongraphic characters in strings and character constants. Use them to specify actions such as carriage returns. You can use them to control movements from one tab position to the next on displays and printers. You can also use them to provide literal representations of characters that normally have special meanings.

An escape sequence consists of a backslash followed by a letter or combination of digits. The following table lists the C language escape sequences:

Escape Sequence	Name	ASCII
\n	New line	(LF)
\t	Horizontal tab	(HT)
\v	Vertical tab	(VT)
\b	Backspace	(BS)
\r	Carriage return	(CR)
\f	Form feed	(FF)

Escape Sequence	Name	ASCII
\a	Bell (alert)	(BEL)
\'	Single quotation mark	
\"	Double quotation mark	
\\	Backslash	
ddd	ASCII character in octal	
xd, *xdd* or*xddd*	ASCII character in hexadecimal	

IBM Extension

If the backslash precedes a character that is not included in the list above, the compiler ignores the backslash and represents that character literally. For example, the pattern **\c** represents the character **c** in a string literal or character constant.

End of IBM Extension

The sequence *ddd* lets you code any character in the ASCII character set as a three-digit octal character code. The sequences **\x***d*, **\x***dd*, and **\x***ddd* let you code an ASCII character as a hexadecimal character code. In all cases, *d* represents a valid digit (0-7 in octal, 0-9 and A-F in hexadecimal). For example, you can code the backspace character as \010 or \x08 and the ASCII null character as \0 or \x0.

Only octal digits can appear in an octal escape sequence. At least one digit must appear. You can code the backspace character as \10.

Similarly, a hexadecimal escape sequence must contain at least one digit. You can omit the second and third digits. For example, you can code the hexadecimal escape sequence for the backspace character as \x8. A safe practice when using octal and hexadecimal escape sequences in strings is to give all three digits of the escape sequence. If the character following a short escape sequence happens to be an octal or hexadecimal digit, the compiler interprets it as the omitted part of the sequence.

Escape sequences let you send nongraphic control characters (ASCII 0x00 - 0x1F and 0x80 - 0xFF) to a display device. For example, the

escape character **\033** often appears as the first character of a control command for a display or printer.

Always represent nongraphic characters by escape sequences. Placing a nongraphic character in a C program has unpredictable results and makes debugging difficult because you cannot see the character on the display.

The backslash character (\) also is a continuation character in strings and in preprocessor definitions. When a newline character follows the backslash, the C compiler disregards the backslash and the newline character and treats the next line as part of the previous line. For an example of this treatment, see "String Literals" on page 3-11.

Operators

Operators are special characters or combinations of special characters that specify how the compiler transforms and assigns values. The compiler interprets each combination as a unit, called a *token*.

The following table lists characters that form arithmetic and logical operators for C. It gives the name of each operator. You must specify operators exactly as they appear in the table. You cannot imbed blanks between the characters of multicharacter operators.

Operator	Name	Operator	Name
!	Logical negation	<	Less than
~	Bitwise negation	<=	Less than or equal
++	Increment	>	Greater than
--	Decrement	>=	Greater than or equal
+	Addition	==	Equal
-	Subtraction, negation	!=	Not equal
*	Multiplication, indirection	|	Bitwise OR
/	Division	&	Bitwise AND address

Operator	Name	Operator	Name
%	Remainder	^	Bitwise exclusive OR
<<	Shift left	&&	Logical AND
>>	Shift right	||	Logical OR
sizeof	Size of item		

The following table lists characters that form compound assignment operators for C. It gives the name of each operator.

Operator	Name	Operator	Name
=	Assign	+=	Increment and assign
-=	Decrement and assign	*=	Multiply and assign
/=	Divide and assign	%=	Modulus and assign
>>=	Shift right and assign	<<=	Shift left and assign
&=	Bitwise AND and assign	^=	Bitwise exclusive OR and assign
|=	Bitwise OR and assign		

See "Operators" on page 6-10 for a complete description of each operator.

Constants

A *constant* is a number, character, or character string that you can use as a value in a program. The value of a constant does not change from one run to the next.

The C language has five kinds of constants: integer constants, floating-point constants, character constants, enumeration constants and string literals. The following descriptions show the format and use of each.

Integer Constants

An *integer constant* is a decimal, octal, or hexadecimal number that represents an integer value.

The following table illustrates the form of integer constants.

Decimal	Octal	Hexadecimal
10	012	0xA, 0XA
132	0204	0x84
32179	076663	0x7db3 or 0x7DB3

No white-space or blank characters can appear between the digits of an integer constant.

A decimal constant has the following form:

digits

where *digits* is one or more decimal digits (0 through 9).

An octal constant has the form:

0odigits

where *odigits* is one or more octal digits (**0** through **7**). You must include the leading **0**.

A hexadecimal constant has the form:

0xhdigits

or

0Xhdigits

where *hdigits* is one or more hexadecimal digits (**0** through **9** and either uppercase **A** through **F** or lowercase **a** through **f**). You must include the leading **0** followed by an **x** or an **X**.

Integer constants specify positive values. If you need a negative value, use the minus sign (–) in front of the constant. IBM C/2 treats the minus sign as a unary arithmetic operator, forming an expression that the compiler evaluates to a negative value.

Every integer constant has a *data type* based on its value. The data type determines what conversions the compiler must perform when it

uses the constant. The IBM C/2 compiler considers decimal constants as signed quantities. It can give them a data type of **int** or **long**. This depends on the size of the value.

IBM C/2 can give octal and hexadecimal constants data types of **int**, **unsigned int**, **long**, or **unsigned long**. This depends on the size of the constant. If the IBM C/2 compiler can represent the constant as an integer (**int**), it gives the constant the **int** type. If the constant is larger than the maximum, positively-signed value that the compiler can represent as an **int** but small enough without the sign to be represented in the same number of bits as an **int**, the compiler assigns it a data type of **unsigned int**. Similarly, a constant too large for the compiler to represent as an **unsigned int** becomes a **long** or, if necessary, an **unsigned long**.

The following table shows ranges of values and their corresponding types for octal and hexadecimal constants. These ranges apply to a machine where the **int** data type is 16-bits long.

Hexadecimal Range	**Octal Range**	**Data Type**
0x0000 - 0x7FFF	0000000 - 0077777	int
0x8000 - 0xFFFF	0100000 - 0177777	unsigned int
0x00010000 - 0x7FFFFFFF	000000200000 - 017777777777	long
0x80000000 - 0xFFFFFFFF	020000000000 - 037777777777	unsigned long

The consequence of these rules for data types is that IBM C/2 does not extend hexadecimal and octal constants to allow for a sign when it converts them to longer types.

You can force the IBM C/2 compiler to assign any integer constant a **long** type, an **unsigned** type, or both. Do this by adding the letter *l* or *u* (or *L* or *U*) to the end of the constant. The following table illustrates *long integer constants*.

Decimal	Octal	Hexadecimal
10L	012L	0xaL or 0xAL
10LU	012LU	0xaLU or 0xALU
79l	0117l	0x4fl or 0x4Fl
79u	0117u	0x4fu or 0x4Fu

"Type Specifiers" on page 5-2 describes the different data types.
"Type Conversions" on page 6-31 describes data-type conversions.

Floating-Point Constants

A *floating-point* constant is a decimal number representing a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. Floating-point constants have the form:

digits.[*digits*][E[+|-]*digits*]

or

[*digits*].*digits*[E[+ |-]*digits*]

where *digits* is one or more decimal digits (**0** through **9**) and **E** (or **e**) is the exponent symbol. You must represent a number with either the digits before the decimal point (the integer portion of the value), the digits after the decimal point (the fractional portion), or both. The exponent consists of the exponent symbol followed by a positive or negative constant integer value. You can omit the decimal point only when you give an exponent. No blank characters can separate the digits or characters of the constant.

Floating-point constants specify non-negative values. If you need negative values, place the minus sign (—) in front of the constant to form a constant floating-point expression with a negative value. IBM C/2 treats the minus sign as an arithmetic operator.

The following example shows floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

You can omit the integer portion of the floating-point constant, as in the following example:

```
.75
.0075e2
-125
-175E-2
```

All floating-point constants have a data type of **double.**

Force the IBM C/2 compiler to assign a floating-point constant to have type **float** or type **long double** by adding the letter *F* or *L* (or *f* or *l*) to the end of the constant. For example, these constants will have type **float**:

```
3.75F
-12.f
```

and these constants will have type **long double**:

```
3.75L
-1.4E-21
```

Character Constants

A *character constant* is a letter, digit, punctuation character, or escape sequence enclosed in single quotation marks. The value of a character constant is the character itself. You cannot use character constants consisting of more than one character or escape sequence.

A character constant has the form:

'*char*'

where *char* is any character from the representable character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a newline character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash as shown in the following table. To represent a newline character, use the escape sequence **\n**. The following table shows how to code character constants.

Constant	Value
a	Lowercase a
?	Question mark
\b	Backspace
\x1B	ASCII escape character

Constant	Value
\'	Single quotation mark
\\	Backslash

Character constants have the data type **int**. IBM C/2 extends them in type conversions to allow for a sign.

IBM C/2 permits hexadecimal bit patterns as character constants. These consist of a backslash (\), the letter **x**, and up to 3 hexadecimal digits (for example, **\x12**).

IBM C/2 defines two additional escape sequences:

- The sequence **\v** represents a vertical tab (VT).
- The sequence **\"** represents the double quotation mark character.

Enumeration Constants
An *enumeration constant* is an identifier declared in an **enum** declaration. The constant has type **int**. See "Naming Classes" on page 4-9 and "Enumeration Declarations" on page 5-19 for more information on enumeration constants.

String Literals
A *string literal* is a sequence of letters, digits, and symbols enclosed in double quotation marks.

The form of a string literal is:

"characters"

where *characters* is one or more characters from the representable character set, excluding the double quotation mark, the backslash, and the newline character. To form string literals that occupy more than one line, type a backslash and press Enter. The backslash causes IBM C/2 to ignore the resulting newline character. For example, the string literal:

```
"Long strings can be bro\
ken into two pieces."
```

is identical to the string:

```
"Long strings can be broken into two pieces."
```

Escape sequences (such as \n and \") can appear in string literals. To use the double quotation mark or backslash character within a

string literal, precede it with a backslash, as in the following examples:

```
"This is a string literal."
"Enter a number between 1 and 100 \nOr press Enter"
"First\\Second"
"\"Yes, I do,\" she said."
```

These examples produce the following output:

```
This is a string literal.
Enter a number between 1 and 100
Or press Enter
First\Second
"Yes, I do," she said.
```

An easier way of representing a long string is to separate it into two string literals, written with only white space between them. The compiler joins them into a single string with one null character at the end. In this way, you can extend a string literal beyond the end of a physical line without the need for a backslash-newline combination. You can indent the second part of the string on the next line. For example:

```
"Long strings can be bro"
"ken into two pieces."
```

IBM C/2 stores the characters of a string in order at contiguous storage locations and automatically adds a null character (\0) to mark the end of the string. The compiler considers each string in a program a distinct item. If two identical strings appear in a program, the compiler reserves each distinct storage.

String literals have the data type **char**[ ] by default. The compiler stores a string literal as an array with each element having a data type of **char**. The number of elements in the array is the number of characters in the string literal plus 1, because the null character stored after the last character counts as an array element.

Identifiers

Identifiers are names you give variables, functions, and labels in a given program. You create an identifier by declaring it with the associated variable or function. You can use the identifier to refer to the item after the declaration.

An identifier is one or more letters, digits, or underscores (_). It begins with a letter or underscore. Use leading underscores with

care. Identifiers beginning with an underscore can conflict with the names of hidden system routines to produce errors.

You can use any number of characters in an identifier. Only the first 31 characters are significant to IBM C/2.

IBM Extension

External identifiers are significant to 31 characters, also.

Note: This number of significant characters is different from some C compilers. Other programs that read compiler output, such as a linker, might allow fewer characters.

End of IBM Extension

These are identifiers:

```
j
cnt
temp1
top_of_page
skip12
```

IBM Extension

The $ sign may also be used as the first character in an identifier.

End of IBM Extension

IBM C/2 is case sensitive. Uppercase and lowercase letters are separate and distinct characters. You can create distinct identifiers with the same spelling but with different cases for one or more of the letters. Each of the following identifiers is unique:

```
add ADD Add aDd
```

IBM C/2 does not allow identifiers that have the same spelling and case as a C language keyword. You can find a description of keywords in the next section.

The linker might further restrict the number and type of characters for *globally visible symbols*. Unlike the compiler, the linker might not distinguish between uppercase and lowercase letters. See *IBM C/2 Compile, Link, and Run* for information about the linker.

Keywords

Keywords are predefined identifiers with special meaning. You can use keywords only as documented in this library. The names of program items must not conflict with these keywords:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

You cannot redefine keywords. You can substitute text for keywords before compiling by using C preprocessor directives.

IBM Extension

The following identifiers can be keywords in some programs:

cdecl	**fortran**	**interrupt**	**_near**
_cdecl	**_fortran**	**_interrupt**	**pascal**
_export	**huge**	**_loadds**	**_pascal**
far	**_huge**	**near**	**_saveregs**
_far			

End of IBM Extension

Comments

A *comment* is a sequence of characters that the compiler interprets as a single white-space character. The compiler otherwise ignores comments. A comment has this form:

```
/* characters */
```

where *characters* is any combination of characters from the representable character set. This includes newline characters but excludes the combination ***/**. This means comments can occupy more than one line, but you cannot nest them. You can also set off a comment on a single line by starting it with **//** and finishing it at the end of the same line.

Use comments to document statements and actions in a C program. You can put them anywhere that C allows a blank character. Because the compiler ignores the characters of a comment, keywords in comments do not produce errors.

These are comments:

```
/* Comments separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/*****************************************
   Comments can span several lines.
 *****************************************/
```

The following example shows how the *//* signals that the rest of the line is to be treated as a comment. (See the next section for a discussion of tokens.) The next (unescaped) newline terminates the comment.

```
// this is a comment on a single line
a = b
if (a <= b)      // this also is one line
      b--;
```

Because you cannot nest comments, this is an error:

```
/* You cannot /* nest */ comments */
```

The compiler recognizes the ***/*** after **nest** as the end of the comment. It tries to process the remaining text as a statement. An error occurs when it cannot do so.

To suppress the compiling of a large part of a program that contains comments, use the **#if** preprocessor directive instead of comments as in the following example.

```
#if 0
      ...
   /*  code to be suppressed       */
      ...
#endif
```

Tokens

When the compiler processes a program, it divides the program into groups of characters known as tokens. A *token* is the smallest item of program text that has meaning to the compiler. It cannot be divided further. Operators, constants, identifiers, and keywords are tokens. Characters such as brackets ([ ]), braces ({}), less-than, greater-than (<>), parentheses (()), and commas (,) are also tokens.

Tokens are delimited by blank characters and by other tokens, such as operators and punctuation symbols. To prevent the compiler from breaking an item into two or more tokens, do not use blanks between the characters of identifiers, multicharacter operators, and keywords.

The compiler interprets text by including as many characters as possible in a single token before moving on to the next token. Because of this, the compiler can sometimes misinterpret tokens. Use white space to separate tokens.

In the following expression, the compiler makes the longest possible operator (+ +) from the three plus signs and processes the remaining plus sign as an addition operator (+).

```
i+++j
```

The compiler interprets this expression as (*i* + +) + (*j*), not (*i*) + (+ +*j*). The compiler increases *i* by 1 and adds *j*. Use blanks and parentheses to make clear what you want in such cases.

Chapter 4. Program Structuring

This chapter shows how to structure C language source programs. It introduces terms that describe the various items in IBM C/2.

Source Program

C source programs are a collection of one or more directives, declarations, pragmas, and definitions. *Directives* instruct the C preprocessor to act on the text of the program. *Declarations* establish names and define characteristics of variables, functions, and data types. *Pragmas* instruct the compiler to carry out certain functions. *Definitions* are declarations that define the name and the type of variables or functions. A variable definition can give an initial value to the variable it declares. A definition reserves storage for the variable.

A function definition specifies the function body. The *function body* is a compound statement containing declarations and statements that define what the function does. The function definition declares the function name, its parameters, and the data type of the value it returns, if any.

A source program can have any number of directives, declarations, and definitions. Each statement must have the proper syntax or format. Statements can appear in more than one order. The statement order affects how you can use variables and functions. In particular, the order affects the visibility of variables. See Chapter 7, "Using C Statements" for more information about statements.

A program must contain one function definition. This **main** function defines what action the program takes.

This is source code of a simple C program:

```
# define TWO 2          /* Preprocessor directive */

int x = 1;      /* Variable definitions */

int y = TWO;

extern int printf ( );/* Function declaration */

main ()         /* Function definition
         for main function */
{
   int z;      /* Variable declarations */
   int w;

   z = y + x;     /* Program statements */
   w = y - x;
   printf ("z=%d \n w= %d \n", z, w);
}
```

This source program defines **main** and declares a reference to the function **printf**. The program defines the variables *x* and *y* with initial values. It also defines the variables *z* and *w*.

Source Files

You can divide source programs into source files. *C source files* are text files that contain any combination of complete directives, declarations, and definitions. To compile the source program, you must compile the source files individually and then link them. With the **#include** directive, you combine source files into larger source files.

It is sometimes useful to place variable definitions in one source file and declare references to those variables in any source files that use them. This makes definitions easy to find and change, if necessary. For the same reason, you can organize manifest constants and macros into separate files and include them into source files as required.

Directives in a source file apply to that source file and its included files only. Each directive applies only to the portion of the file following the directive. If a set of directives applies throughout a source program, all the source files must include the set.

Pragmas usually act on a specific part of a source file. How you use the pragma determines the specific compiler action the pragma defines. Information for each individual pragma is in *IBM C/2 Compile, Link, and Run*.

The example that follows is a source program in two source files. The **main** and **max** functions are in separate files. The program begins with the **main** function.

```
/*****************************************
 Source file 1 - main function
 *****************************************/

#define ONE 1
#define TWO 2
#define THREE 3
 /* Function declaration */
extern int max(int, int);
 /* Function definition */
main ()
{
   int u, w, x, y,z;
   w = max(u,ONE)
   x = max(w,TWO)
   y = max(x,THREE)
   z = max(y,z)
}
/*************************************
 Source file 2 - max function
 *************************************/
 /* Function  definition */
int max (a, b)
int a, b;
{
   if ( a > b )
       return (a);
   else
       return (b);
}
```

The first source file declares the function **max**, which is defined elsewhere. This is a *forward declaration*, a reference to the function in source file 2. Four statements in **main** are *function calls* to **max**.

The lines beginning with a # sign are preprocessor directives. These direct the preprocessor to replace the identifiers ONE, TWO, and THREE with the specified numbers. The directives do not apply to the second source file.

The second source file contains the function definition for **max**. This definition answers the calls to **max** from the first source file. After you compile the source files, you can link them and run them as a single program.

Running the Program

Every program has a primary program function. Traditionally, the primary program function has the name **main**. IBM C/2 requires the explicit name **main** for the primary function.

The **main** function is the starting point for running a program. It controls the program by directing the calls to other functions. A program usually stops running at the end of the **main** function, although it can stop at other points in the program.

The source program usually has more than one function. Each function performs at least one specific task. The **main** function calls these functions to perform tasks. When the source program calls a function, it begins running the first statement in the called function. The function returns control at the **return** statement or at the end of the function.

You can declare any function to have parameters. Functions called by other functions receive values for their parameters from the arguments of the calling functions. You can declare parameters of the **main** function to receive values from outside the program. (The command prompt that starts the program can pass such values.)

The first three parameters of the **main** function have the names **argc**, **argv**, and **envp**. The **argc** parameter holds the total number of arguments passed to the **main** function. The **argv** parameter is an array of pointers. Each element points to a string representation of an argument passed to the main function.

IBM Extension

The **envp** parameter is a pointer to an array of pointers to string values. The string values in this table set up the environment in which the program runs.

End of IBM Extension

The operating system supplies values for the **argc**, **argv**, and **envp** parameters. You supply the arguments to the **main** function. The operating system, not the C language, determines the argument-passing convention that a particular system uses. See the section

about the C Calling Sequence in Chapter 6 of *IBM C/2 Compile, Link, and Run*. You must declare any parameters to functions when you define the function.

Lifetime and Visibility

Lifetime and visibility are important in understanding the structure of a C program. The lifetime of a variable or function can be global or local. An item with a *global lifetime* has storage and a defined value for the duration of the program. The compiler reserves storage for an item with a *local lifetime* each time the current point of program execution enters the program block that defines or declares that item. When the current point leaves that block, the local item loses its storage and its value.

An item is *visible* in a block or source file if the block or source file knows the data type and the declared name of the item. An item can be globally visible. This means that it is visible, or is visible after appropriate declarations, throughout all source files. For more information about visibility between source files (also known as linkage), see the discussion of storage-class specifiers in "Storage Classes" on page 5-35.

A *program block* or *block* is a compound statement. Compound statements consist of declarations and statements. The bodies of C functions are compound statements. You can nest blocks. Function bodies can contain blocks that contain other blocks.

Declarations and definitions within blocks occur at the *internal level*. Declarations and definitions outside of all blocks occur at the *external level*.

You can declare or define variables at the external or the internal level. You can declare functions at the external or the internal level, but you can define functions only at the external level. The definition or body of a function cannot be a block nested in a block.

All functions have global lifetimes, regardless of where you declare them. Variables declared at the external level have global lifetimes. Variables declared at the internal level usually have local lifetimes. You use the storage class specifiers **static** and **extern** to declare

global variables or to make references to global variables within a block.

Variables declared or defined at the external level are visible from the point at which you declare or define them to the end of the source file. You can make these variables visible in other source files with appropriate declarations, as described in "Storage Classes" on page 5-35. Variables with a **static** storage class at the external level are visible only within the source file in which you define them.

In general, variables declared or defined at the internal level are visible from the point at which you first declare them to the end of that block. These variables are local variables. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition replaces the external-level definition to the end of the block. The compiler restores the visibility of the external-level variable when the current point of execution leaves the block.

You can nest block visibility. This means that a block nested inside a block can contain declarations that redefine variables declared in the outer block. The new definition of the variable applies to the inner block. C restores the original definition when the current instruction returns to the outer block. A variable from the outer block is visible inside inner blocks that do not redefine the variable.

Functions with **static** storage class are visible only in the source file in which you define them. All other functions are globally visible.

The following table summarizes the main factors that determine the lifetime and visibility of functions and variables. The table does not cover all cases. Refer to "Storage Classes" on page 5-35 for more information.

Level	Item	Storage Class Specifier	Life time	Visibility
External	Variable declaration or definition	**static**	Global	Restricted to single source file
	Variable declaration or definition	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

The following program illustrates blocks, nesting, and visibility of variables. In this example, there are four levels of visibility: the external level and three block levels. Assuming that you have defined the function printf elsewhere, the main function prints the values 1, 2, 3, 0, 3, 2, 1.

```
/* i defined at external level */
int i = 1;

/* main function defined at external level */
main ()
{

   /* Prints 1 (value of external level i) */
   printf("%d\n", i);

   /* First nested block */
   {

      /* i and j defined at internal level */
      int i = 2, j = 3;

      /* Prints 2, 3 */
      printf("%d\n%d\n", i, j);

      /* second nested block */
      {

  /* i is redefined */
  int i = 0;

  /* Prints 0, 3 */
  printf("%d\n%d\n", i, j);

      /* End of second nested block */
      }

      /* Prints 2 (outer definition restored)*/
      printf("%d\n", i);

   /* End of first nested block */
   }

   /* Prints 1(external level definition restored)*/
   printf("%d\n", i);
}
```

Naming Classes

In any C program, identifiers refer to many items. You use identifiers for functions, variables, parameters, union members, and other items. C lets you use the same identifier for more than one class of identifier, as long as you follow the rules outlined in this section.

The compiler sets up naming classes to distinguish among classes of identifiers. You must assign unique names within each class to avoid conflict. An identical name can identify an item in more than one identifier class. This means that you can use the same identifier for two or more items if the items are identifiers in different identifier classes. The context of an identifier within a program lets the compiler resolve its class without ambiguity.

Items that you can name and the rules for naming them are:

- **Variables and Functions**: Variable and function names are in a naming class with parameters of functions, **typedef** names, and enumeration constants. Variable and function names must be distinct from other names in this class with the same visibility. You can redefine function names and variable names within program blocks as described in "Lifetime and Visibility" on page 4-5.
- **Parameters**: The names of parameters of a function are in a class with the names of the variables of a function. The parameter names must be distinct from the variable names declared at this level. Redeclaring parameters at this level causes an error, but you may redeclare parameter names in nested blocks within the function body.
- **Enumeration Constants**: Enumeration constants are in the same naming class as variable and function names. Enumeration constant names must be distinct from variable, function, and enumeration constant names with the same visibility. Like variable names, the names of enumeration constants have nested visibility. You can redefine them within blocks.
- **Tags**: Enumeration, structure, and union tags are in a single naming class. (For a description of tags see "Variable Declarations" on page 5-18.) Each enumeration, structure, or union tag must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
- **Members**: The members of an individual structure or union variable form a naming class to themselves. The name of a member

must be unique within the structure or union. It need not be distinct from a member name in another structure or union variable, or from any other variable name.

- **Statement Labels**: Statement labels form a separate naming class. A statement label must be distinct from other statement labels in the same function. Statement labels need not be distinct from other names or from label names in other functions.
- **Typedef Names**: The names of the data types that you define with the **typedef** declaration are synonyms of data-type specifiers. (For a description of data-type specifiers and the **typedef** declaration see "Type Specifiers" on page 5-2.) The type names that you define with the **typedef** declaration are in a naming class with variable and function names. The names must be distinct from all variable and function names with the same visibility, and from the names of parameters of functions and enumeration constants. Like variable names, you can use **typedef** to redefine data-type names within program blocks.

Structure tags, structure members, and variable names are in three different naming classes; no conflict occurs between the three items named *student* in the following example:

```
struct student /* structure tag */
              /*    the name of a new type */
{
   char *student; /* structure member */

   int class;
   int id;        /* two more members */

} student; /* the name of one variable */
                    /*     of the new type */

/* Next are three different contexts referring to three different 'students */

struct student *studptr;          /* declares a pointer to the new */
                                  /*   structure type called student */

studptr = &student              /* initialize studptr to point to the  */
                                  /*   variable called student */

studptr - > student = "Harry";    /* assign a value to the structure */
                                  /*    member called student */

student.student = "Harry";        /* (does exactly the same thing */
                                  /*     as the previous example */
```

Chapter 5. Declaring Variables, Functions, and Data Types

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form:

[sC specifier][type-specifier]declarator[=initializer][,declarator...]

where *sC specifier* is a storage class specifier; *type-specifier* is the name of a defined type; *declarator* is an identifier; and *initializer* gives a value or sequence of values you assign to the variable you are declaring. You must explicitly declare all C variables before using them. You can declare C functions explicitly in a function declaration or implicitly by calling the function before you declare or define it.

The C language defines a standard set of data types. You can add to that set by declaring new data types based on types already defined. You can declare arrays, data structures, and pointers to either variables or functions.

C declarations require one or more declarators. A *declarator* is an identifier that you can change with brackets, asterisks, or parentheses. It declares a variable, array, pointer, or function type. When you declare simple variables (such as character, integer, and floating-point values), or structures and unions of simple variables, the declarator is just an identifier.

The location of the declaration within the source program, and the presence or absence of other declarations of the variable are important factors in determining the visibility of variables.

The storage class specifier of a declaration affects how the declared item is stored and initialized and which portions of a program can refer to it. The four storage class specifiers are: **auto**, **extern**, **register**, and **static**.

Type Specifiers

The C language provides definitions for a set of basic data types. The following lists their names:

Integral Types

char	**long**	**unsigned short int**
signed	**long int**	**signed int**
signed char	**unsigned**	**signed long int**
int	**unsigned char**	**int**
short	**unsigned long**	**signed short**
signed short int	**unsigned int**	**unsigned short**
short int	**signed long**	

Floating-Point Types

double
float
long double

Another class of data types is known as the *enumerated* types. These consist of enumerations and **void**. For a discussion of type specifiers for enumeration types see "Structure, Union, and Enumeration Types" on page 5-46. You can use the **void** type to declare functions that return no value or as the type of a pointer that can be converted to a pointer to an object of any type. You can create additional type specifiers with **typedef** declarations.

Type specifiers are commonly abbreviated, as shown in the following figure. Integral types are signed by default. Thus, if the **unsigned** keyword is omitted from the type specifier, the integral type is signed, even if the **signed** keyword is not specified.

Type Specifier	Abbreviation
char	- -
int	- -
short int	short
long int	long
unsigned char	- -
unsigned int	unsigned
unsigned short int	unsigned short

Type Specifier	Abbreviation
unsigned long int	unsigned long
float	--
const int	const
volatile long int	volatile long

Note: This book uses the abbreviated forms in the preceding table instead of the long forms of the type specifiers and assumes that the **char** type is signed by default. Throughout this book, **char** stands for **signed char**.

IBM Extension

The /J option is available to change the default for the **char** type from signed to an unsigned type. When this is in effect, the abbreviation **char** has the same meaning as **unsigned char**. You must use the **signed** keyword to declare a signed character value.

End of IBM Extension

The **const** type specifier declares an object as nonmodifiable. The **const** keyword can be used as a modifier for any fundamental or aggregate type, or to modify a pointer to an object of any type. A **typedef** can be modified by a **const** type specifier. A declaration that includes the keyword **const**, as a modifier of an aggregate type declarator, indicates that each element of the aggregate type is unmodifiable. If an item is declared with only the **const** type specifier, its type is taken to be **const int**. A **const** object can be placed in a read-only region of storage.

The **volatile** type specifier declares an item whose value can be legitimately changed by something beyond the control of the program it appears in. The **volatile** keyword can be used in the same circumstances as **const**. An item can be both **const** and **volatile**, in which case the item cannot be modified by its own program but can be modified by some asynchronous process. The **volatile** keyword is implemented syntactically but not semantically.

The following table summarizes the storage associated with each basic type and gives the range of values that you can store in a variable of each type. Because the **void** type does not apply to variables, it is not in this table.

Type	Storage	Range of Values (Internal)
char	1 byte	−128 to 127
int and unsigned	implementation-dependent	
short	2 bytes	−32,768 to 32,767
long	4 bytes	−2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned short	2 bytes	0 to 65,535
float	4 bytes	IEEE standard notation; see discussion below
double	8 bytes	IEEE standard notation; see discussion below

The **char** type is signed. Use the **char** type to store a letter, digit, or symbol from the representable character set. The integer value of a character is the ASCII code corresponding to that character. Because the compiler interprets the **char** type as a signed 1-byte integer, values in the range −128 to 127 are permitted for **char** variables, although only the values from 0 to 127 have character equivalents. In type conversions, IBM C/2 extends a **char** value to allow for a sign.

The C language does not define the storage and range associated with the **int** and **unsigned int** types. Instead, the size of an **int** (signed or unsigned) corresponds to the natural size of an integer on a given machine. For example, on a 16-bit machine the integer size is usually 16 bits, or 2 bytes. On 8088 and 80286 processors, an **int** is 16 bits long. On a 32-bit machine, the integer type is usually 32 bits, or 4 bytes. Thus, the integer size is equal either to the **short int** or the **long int** type, depending on the use. Similarly, the **unsigned int** type is equal either to the **unsigned short** or **unsigned long** type.

C programs use the **int** and **unsigned int** type specifiers widely because they let a particular machine handle integer values in the most efficient way for that machine. However, because the size of the **int** and **unsigned int** types varies, programs that depend on a specific **int** size might not be portable. Use expressions involving the **sizeof** operator in place of hard-coded data sizes to increase the portability of the code.

Use the type specifiers **int** and **unsigned int** (or simply **unsigned**) to define certain features of the C language (for instance, in defining the **enum** type). In these cases, the definition of **int** and **unsigned int** for a particular implementation determines the actual storage.

In IBM C/2, the **short** type is 16 bits in length; the **long** type 32 bits. Characters are placed into consecutive memory locations starting with the lowest address byte, while **ints** are stored in the 80x86 hi-byte, lo-byte format (high byte at the lower address). As a result, storing two characters in an **int** would give an **int** that looked as if the characters had been stored in reverse.

Range of Values

The range of values for a variable lists the minimum and maximum values that the machine can represent internally in a given number of bits. However, because of the conversion rules of the C language, you cannot always use the maximum or minimum for a variable of a given type in an expression.

For example, the constant expression –32768 consists of the arithmetic negation operator (–) applied to the constant value 32768. Because 32768 is too large to represent as a **short**, it is given **long** type, and consequently the constant expression –32768 has **long** type. You can represent the value –32768 as a **short** only by type-casting it to the **short** type. The type cast loses no information because C can represent –32768 internally in 2 bytes of storage space.

You can represent a value such as 65000 as an **unsigned short** only by type-casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65000 in decimal notation is a signed constant and is given **long** type because 65000 does not fit into a **short**. You can then cast this **long** value to the **unsigned short** type without loss of information, because 65000 fits

into 2 bytes of storage space when it is stored as an unsigned number.

Octal and hexadecimal constants have either signed or unsigned type, but they are never sign-extended in type conversions.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, an 8-bit excess 127 binary exponent, and a 23-bit mantissa. The phrase *8-bit excess 127* means the exponent has binary 1111111 (decimal 127) added to the 8-bit binary representation to provide a method of checking for errors.

The mantissa represents a number between 1.0 and 2.0. The high-order bit of the mantissa is 1, which is not stored in the number and makes the effective precision 24 bits. This representation gives an exponent range of 10 to the (+ or—) 38th power and up to seven digits of precision. The maximum value of a **float** is normally 3.4E38.

For example, consider the real number 6.0. As the product of a number between 1.0 and 2.0 and a power of two, its unique expression is 1.5×2^2. The exponent (2) is represented as 2 + 127, or 129. The sign bit for a positive number is 0 and the mantissa is binary 1.1, of which the first 1 is not represented in the number.

The IEEE representation of 6.0 in binary is:

```
   0   10000001  10000000000000000000000
     ^         ^
sign   exponent       mantissa
```

Values with **double** type have 8 bytes. The format is similar to the **float** format, except that the exponent is 11 bits excess 1023 and the mantissa has 52 bits (plus the implied high-order 1 bit). The phrase *11 bits excess 1023* means the exponent has binary 1111111111 (decimal 1023) added to the 11-bit binary representation to provide a method of checking for errors.

Again, consider the real number 6.0. As the product of a number between 1.0 and 2.0 and a power of two, its unique expression is 1.5 x 2^2. The exponent (2) is represented as 2 + 1023, or 1025. The sign bit for a positive number is 0 and the mantissa is binary 1.1, of which the first 1 is not represented in the number. The IEEE representation of 6.0 in binary is:

```
    0  10000000001  10000000000000000000000000000000000000000000000000000
     ^            ^
sign   exponent     mantissa
```

This gives an exponent range of 10 to the (+ or −) 308th power and up to 15 digits of precision. Type **long double** is equivalent to **double** in IBM C/2.

Declarators

Another class of C data items is known as *derived types*. C lets you declare arrays of values, pointers to values, and functions returning values of specified types. To declare these items, you must use a *declarator*. A declarator is an identifier that you can change with brackets, parentheses, and asterisks to declare an array, function type, or pointer. Declarators appear in the array, function, and pointer declarations described in "Array Declarations" on page 5-27, "Function Declarations" on page 5-32, and "Pointer Declarations" on page 5-31. This section discusses the rules for forming and interpreting declarators.

Format

```
declarator[constant-expression]
*declarator
declarator()
declarator(arg-type-list)
(declarator)
```

Pointer, Array, and Function Declarators

When a declarator consists of an unchanged identifier, the item you are declaring has a *basic type*. Asterisks can appear to the left of an identifier, changing it to a pointer type. If you follow the identifier by brackets, the type changes to an array type. If you follow the identifier by parentheses, which can contain type descriptions, the type changes to a function returning a type.

A declarator is not a complete declaration; you must include a type specifier as well. The type specifier gives the type of the elements for an array type, the type of object addressed by a pointer type, and the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail. The following examples show the simplest forms of declarators. The example below declares an array of **int** values (**list**):

```
int list[20];
```

The next example declares a pointer to a **char** value (**cp**):

```
char *cp;
```

This example declares a function with no arguments returning a **double** value (**func**):

```
double func(void);
```

Complex Declarators

You can enclose any declarator in parentheses. Use parentheses to specify an interpretation of a complex declarator. A complex declarator is an identifier qualified by more than one array, pointer, or function modifier.

You can apply various combinations of the array, pointer, and function modifiers to a single identifier. Some combinations are illegal. An array cannot comprise functions, and a function cannot return an array or a function.

When the compiler interprets complex declarators, brackets and parentheses on the right of the identifier take precedence over asterisks on the left of the identifier. Brackets and parentheses have the same precedence, and the compiler interprets them left-to-right. The compiler applies the type specifier as the last step. Use parentheses to change the default order of association in a way that forces a particular interpretation.

A simple rule in interpreting complex declarators is to read them from the inside out. Start with the identifier and look to the right for brackets or parentheses. Interpret any of these, and then look to the left for asterisks. If you find a right parenthesis at any stage, go back and apply these rules to everything within the parentheses before proceeding. Finally, apply the type specifier.

To show this rule, the following example numbers the steps in order:

```
char *(*(*var)())[10];
 7   6 4 2 1  3  5
```

1. The identifier *var* is:
2. A pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

Example

The following examples provide further illustration and show how parentheses can affect the meaning of a declaration.

In Example 1, the array modifier has a higher priority than the pointer modifier, so *var* is an array. The pointer modifier applies to the type of the array elements; the elements are pointers to **int** values.

```
int *var[5];   /* Array of pointers to int values */   Example 1
```

In Example 2, parentheses change the meaning of the declaration in the first example. Now the pointer modifier is applied before the array modifier, and *var* is a pointer to an array of 5 **int** values.

```
int (*var)[5]; /* Pointer to array of int values */   Example 2
```

Function modifiers also have a higher priority than pointer modifiers. Example 3 declares *var* to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.

```
long *var(long,long); /* Function returning pointer to long */  Example 3
```

Example 4 is like Example 2. Parentheses give the pointer modifier a higher priority than the function modifier, and *var* is a pointer to a function returning a **long** value. Again, the function takes two **long** arguments.

```
long (*var)(long,long);
     /* Pointer to function returning long */          Example 4
```

The elements of an array cannot be functions. Example 5 shows how to declare an array of pointers to functions instead of declaring the functions themselves as an array. This example declares *var* as an array of pointers to functions returning structures with two members. The arguments to the functions are two structures with the **both** structure type:

```
struct both

  {                                                Example 5
    int a;
    char b;
  } ( *var[ ] )( struct both, struct both );
                  /*Array of pointers to functions
                 returning structures */
```

The parentheses surrounding ***var* [ ]** are required. Without them, the declaration is an illegal attempt to declare an array of functions:

```
struct both *var[ ](struct both, struct both);  /* ILLEGAL */
```

Example 6 shows how to declare a function returning a pointer to an array, because functions returning arrays are illegal. Here *var* is declared to be a function returning a pointer to an array of 3 **double** values. The function *var* takes one argument; the argument, like the return value, is a pointer to an array of 3 **double** values. The argument type has a complex, abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type is an array of 3 pointers to **double** values. See "Type Names" on page 5-48 for a discussion and examples of abstract declarators.

```
double ( *var( double (*)[3] ) )[3];     Example 6
                /* Function returning a pointer to
                  an array of double values */
```

A pointer can point to another pointer, and an array can contain array elements, as Example 7 shows. Here *var* is an array of 5 elements. Each element is a 5-element array of pointers to pointers to unions with two members.

```
union sign            /* Array of arrays of pointers       Example 7
                       to pointers to unions */
  {
    int x;
    unsigned y;
    } **var[5][5];
```

Example 8 shows how the placement of parentheses alters the meaning of the declaration. In this example, *var* is a 5-element array of pointers to 5-element arrays of pointers to unions.

```
union sign *(*var[5])[5];     /* Array of pointers  Example 8
                                                        to arrays of pointers
                                                        to unions */
```

Declarators with Special Keywords

IBM C/2 includes the following:

cdecl	**fortran**	**interrupt**	**_near**
_cdecl	**_fortran**	**_interrupt**	**pascal**
_export	**huge**	**_loadds**	**_pascal**
far	**_huge**	**near**	**_saveregs**
_far			

Use these keywords to change the meaning of variable and function declarations. See *IBM C/2 Compile, Link, and Run* for a full discussion of the effects of these special keywords.

When a special keyword occurs in a declarator, it changes the item immediately to the right of the keyword. You can apply more than one keyword to the same item. For example, you can change a function identifier with both the **far** and **pascal** keywords. The order of the keywords in this case does not matter. Changing the function with **far pascal** has the same effect as changing the function with **pascal far**.

You can use two or more special keywords in different parts of the declaration to change the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword.

```
int far * pascal far func(void);
```

The **pascal** and **far** keywords change the function identifier **func**. The return value of **func** is declared to be a **far** pointer to an **int** value.

As in any C declaration, you can use parentheses to change the interpretation of the declaration. The rules governing complex declarators apply to declarations using the special keywords as well.

Example

The following examples show the use of special keywords in declarations.

This example declares a **huge** array named **database** with 65000 **int** elements. The **huge** keyword changes the array declarator.

```
int huge database [65000] ;
```

In the next example, the **far** keyword changes the asterisk to its right, making *x* a **far** pointer to a pointer to **char**.

```
char * far * x;
```

You can also express this declaration as:

```
char * (far *x);
```

The following example shows two equal declarations. Both declare **calc** as a function with the **near** and **cdecl** attributes.

```
double near cdecl calc(double,double);
double cdecl near calc(double,double);
```

The next example also shows two declarations: the first declares a **far fortran** array of characters named **initlist**, and the second declares three **far** pointers, named **nextchar**, **prevchar**, and **currentchar**. You can use these pointers to store the addresses of characters in the **initlist** array. You must repeat the **far** keyword before each declarator.

```
char far fortran intlist  INITSIZE ;
char far *nextchar, far *prevchar, far *currentchar;
```

The following example shows a more complex declaration with several occurrences of the **far** keyword. The **far** keyword always changes the item immediately to its right.

```
char far *(far *getinit)(int far *);
  ^   ^     ^    ^  ^   ^
  6   5     2    1  3   4
```

The steps in interpreting this declaration are as follows:

1. This example declares the identifier **getinit** as
2. a **far** pointer
3. to a function taking
4. a single argument that is a **far** pointer to an **int** value
5. returning a **far** pointer to
6. a **char** value.

The **_saveregs** modifier causes the compiler to generate save and restore instructions for all registers except those used for the return value (AX or AX/DX), if any, on entry/exit from function. Use the **_saveregs** keyword when it is not certain what the register conventions of the caller might be. For example, **_saveregs** could be used for a general-purpose function that will reside in a dynamic link library. Since a function in a dynamic link library might be called

from any language, you may choose not to assume IBM C/2 calling conventions in some cases.

The **_loadds** modifier performs the same function as the **/Au** switch but on a per-function basis. The generated code loads DS register on function entry in the following order:

1. The last segment specified by a **data_seg** pragma.
2. The segment name specified by the **/ND** switch.
3. DGROUP.

See Chapter 2 in *IBM C/2 Compile, Link, and Run* for a discussion of the /Au compiler option.

The **_export** modifier tells the compiler to insert an EXPDEF record into the EXE file to indicate that this routine can be exported from a DLL. This does not remove the need for .DEF files, since the linker assumes that these functions have no IOPL set, have shared data, are not resident, and have no alias name. This information can be overridden with a .DEF file entry for the same function. The compiler does emit the number of parameter words for the function. This cannot be changed by a .DEF file entry with the **iopl_parmwords** field set. The value in the .DEF file must either be 0 (indicating that the linker should use the value given by the compiler) or the same as the value given by the compiler. This field is ignored if IOPL is not requested.

Interrupt Functions

Apply the **interrupt** attribute to a function to tell the compiler that the function is an interrupt handler. This causes the compiler to generate entry and exit sequences that are appropriate for an interrupt-handling function, including saving and restoring all registers and executing an IRET instruction to return.

When an **interrupt** function is called, the DS register is initialized to the C data segment. This allows you to access global variables from within an interrupt function. In addition, all registers (except SS) are saved on the stack. You can access the registers that are saved before the normal entry sequence by declaring the function with a parameter list containing a formal parameter for each register saved.

The following example illustrates such a declaration:

```
void interrupt cdecl far int_handler (unsigned
        es, unsigned ds, unsigned di, unsigned
        si, unsigned bp, unsigned sp, unsigned
        bx, unsigned dx, unsigned cx, unsigned

        ax, unsigned ip, unsigned cs, unsigned
        flags)

{
}
```

The formal parameters must appear in the opposite order from which they are pushed onto the stack. You can omit parameters from the end of the list in a declaration but not from the beginning. For example, if your handler needs to use only DI and SI, you must still provide ES and DS but not necessarily BX or DX.

The compiler always saves and restores registers in the same, fixed order. Regardless of which names you use in the formal parameter list, the first parameter in the list refers to ES, the second refers to DS, and so on.

If your interrupt handler will be called directly from C rather than by an INT instruction, pass additional arguments by declaring all register parameters and then declaring your parameter at the end of the list.

Observe the following precautions when using interrupt functions:

1. Avoid calling standard-library functions, especially functions that rely on DOS INT 21H system calls, within an **interrupt** function. (Functions that rely on INT 21H calls include stream- and low-level-I/O functions.) It may be better to use functions that do not rely on INT 21H or BIOS, such as string-handling functions. Before using a standard-library function in an interrupt function, be sure that you are familiar with the library function and what it does. **Interrupt** functions are for the DOS environment; they should not be used under OS/2.
2. If you change any of the parameters of an **interrupt** function while the function is executing, the corresponding register contains the changed value when the function returns. For example:

```
void interrupt cdecl far int_handler
 (unsigned es, unsigned ds, unsigned di,
  unsigned si)
{
                di = -1;
}
```

This causes the DI register to contain -1 when the int_handler function returns. Generally, it is not a good idea to modify the values of the parameters representing the IP and CS registers in **interrupt** functions. To modify a particular flag, such as the carry flag for certain DOS and BIOS interrupt routines, use the OR operator (|) so that other bits in the flags register are not changed.

3. When an **interrupt** function is called by an INT instruction, the interrupt enable flag is cleared. If your function needs to do significant processing, you should use the **_enable** function to set the interrupt flag so that interrupts can be handled.

interrupt functions often need to transfer control to a second interrupt routine. Do this by:

1. Calling the interrupt routine (after casting it to an **interrupt** function if necessary). Do this if you need to do further processing after the second interrupt routine finishes.

```
void interrupt cdecl new_int ()
{
   ... /* Initial processing here */
    (*old_int) ();
   ... /* Final processing here */
}
```

2. Calling **_chain_intr** with the interrupt routine as an argument. Do this if your routine is finished and you want the second interrupt routine to terminate the interrupt call.

```
void interrupt cdecl new_int ()
{
     ... /* Initial processing here */
     _chain_intr (old_int);
   /* This is never executed */
}
```

The **_export, interrupt, _loadds** and **_saveregs** modifiers have the same binding as **pascal, fortran**, and **cdecl** and can be used in combination with themselves and/or any of the existing language modifiers and **near** and **far**. Unlike the other modifiers, they are allowed on the declaration and definition of functions and pointers to functions only. The **_loadds** and **_export** keywords have no effect on the declaration of function pointers other than for type checking purposes. The compiler strictly enforces type checking for all these attributes on assignment to function pointers and when passing them as function arguments.

Example

The following example generates argument mismatch and type mismatch errors.

```
int far _export _loadds expfunc(void);
int (far _export *fp) ();
int ff(int (_loadds far *ldfp) (void));

fp = expfunc;  /* error: fp must be _loadds also
*/

ff(expfunc);   /* error: arg to ff cannot be
_export */
ff(fp);        /* error: arg to ff must be _loadds
*/
```

In the following example, *funcl* is a **far pascal** function which takes a single argument of any pointer type and does not return a value. The function loads DS on entry. The **_saveregs** attribute tells the compiler that the function saves and restores all register contents.

```
void _loadds _saveregs far pascal funcl(void *s);
```

In the following example *fp* is a far pointer to an **interrupt** function with no arguments. The **_loadds** attribute indicates that the target function loads its own DS.

```
void (far interrupt _loadds *fp) (void);
```

End of IBM Extension

The IBM C/2 compiler recognizes the keyword **volatile** as a type specifier for any object that may change in ways unknown to your system or which has other unknown side effects.

IBM C/2 does not implement the ANSI standard semantics of **volatile**, only the syntax. The compiler accepts legal declarations with **volatile** but otherwise ignores the keyword.

You can give a data object the **volatile** attribute by placing the keyword **volatile** in the declarator. For a **volatile** pointer, you must place the keyword **volatile** between the * and the identifier. For example:

```
int * volatile x;     /* x is a volatile pointer to an int */
```

For a pointer to a volatile data object, you must place the keyword **volatile** before the **identifier* sequence. For example:

```
volatile int *x; /* x is a pointer to a volatile int */
```

The IBM C/2 compiler recognizes the keyword **const** as a type specifier for any object known to be an unchanging value. The compiler may place **const** objects into the **CONST** segment because they are read-only. The compiler may not detect every illegal attempt to modify these objects.

You can give a data object the **const** attribute by placing the keyword **const** in the declarator. For a **const** pointer, you must place the keyword **const** between the * and the identifier. For example:

```
int * const d;        /*  d is a const pointer to an int */
```

The preceding examples show that **const** and **volatile** attach to the object to the left of the keyword unless the keyword begins the declaration, in which case it applies to the identifier itself. If **const** or **volatile** appears in the type specifier, it applies to all declarators in the list.

Variable Declarations

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following variable types:

Simple	Single-value variables with integral or floating-point type.
Enumeration	Simple variables with integral type that hold one value from a set of named integer constants.
Structures	Variables composed of a collection of values that can have different types.
Unions	Variables composed of several values of different types occupying the same storage space.
Arrays	Variables composed of a collection of elements with the same type.
Pointers	Variables that point to other variables. These variables contain the locations of variables, in the form of addresses, instead of the values of variables.

The variable declarations discussed in this section have the general form:

[sc-specifier] type-specifier declarator[,declarator...];

The *sc-specifier* gives the storage class of the variable. In some contexts, you can initialize variables as you declare them. For more

information about initializing variables, see "Initialization" on page 5-42.

The *type-specifier* gives the data type of the variable, and *declarator* is the name of the variable, which you can change to declare an array or a pointer type. You can define more than one variable in the declaration by giving multiple declarators, separated by commas.

Simple Variable Declarations

A declaration for a simple variable defines the name and type of the variable. It can also define the storage class of the variable. The name of the variable is the *identifier* given in the declaration. The *type-specifier* gives the name of a defined data type, as described below.

Format

[sc-specifier]type-specifier identifier[,identifier...];

You can define several variables in the same declaration by giving a list of identifiers separated by commas. Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Example

The following example defines a simple variable *x*. This variable can hold any value in the set defined by the **int** type.

```
int x;
```

The next example defines two variables, *reply* and *flag*. Both variables have **unsigned long** type and hold unsigned long integer values.

```
unsigned long reply, flag;
```

The following example defines a variable, *order*, that has **double** type. You can assign floating-point values to this variable.

```
double order;
```

Enumeration Declarations

An enumeration declaration gives the name of the enumeration variable and defines a set of named-integer constants called the *enumeration set.*

Format

```
enum [tag] {enum-list} identifier [,identifier...];
enum tag identifier[,identifier...];
```

A variable declared to have enumeration type stores any one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **Int** type; the storage associated with an enumeration variable is the storage required for a single **Int** value.

Enumeration declarations begin with the **enum** keyword and have two forms. In the first form, you specify the values and names of the enumeration set in the enumeration list *enum-list* described in detail next. The optional *tag* is an identifier that names the enumeration type defined by the *enum-list*. The *identifier* names the enumeration variable. You can define more than one enumeration variable in the declaration.

The second form uses an enumeration *tag* to refer to an enumeration type. The *enum-list* does not appear in this type of declaration because the enumeration type is defined elsewhere. An error occurs if the given tag does not refer to a defined enumeration type or if the named type is not currently visible.

An *enum-list* has the following form:

```
identifier [=constant-expression]
[,identifier[=constant-expression]]
   .
   .
   .
```

Each identifier names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with the value 1, and so on through the last identifier appearing in the declaration. The name of an enumeration constant is equal to its value.

The *=constant-expression* phrase cancels the default sequence of values. An identifier followed by the phrase *=constant-expression* is associated with the value given by *constant-expression*. The *constant-expression* must have **Int** type and can be negative. The next identifier in the list is associated with the value of *constant-expression* + 1, unless it explicitly has another value.

An enumeration set can contain duplicate constant values, but each identifier in an enumeration list must be different from all other enu-

meration identifiers with the same visibility. For example, the value 0 can be two different identifiers, null and 0, in the same set. The identifiers in the list must also be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists. Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Example

This example defines an enumeration type named **day** and declares a variable named *workday* with that enumeration type. The value 0 is associated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

```
enum day
   {
   saturday,
   sunday = 0,
   monday,
   tuesday,
   wednesday,
   thursday,
   friday
   } workday;
```

The next example assigns a value from the set to the variable *today*. The name of the enumeration constant assigns the value.

```
today = wednesday;
```

The following example declares a variable named *holiday* to have the enumeration type **day**. Because the example above previously declared the **day** type, only the enumeration tag is necessary in this declaration.

```
enum day holiday;
```

Structure Declarations

A structure declaration defines the name of the structure variable and specifies a sequence of variable values, called *members* of the structure, that can have different types. A variable with structure type holds the entire sequence defined by that type. No relationship exists between the members of two different structure types.

Format

```
struct [tag]{member-declaration list} declarator [,declarator...];
struct tag declarator [,declarator...];
```

Structure declarations begin with the **struct** keyword and have two forms, as shown above. In the first form, you specify the types and names of the structure members in the *member-declaration list*, described in detail below. The optional *tag* is an identifier that names the structure type defined by the *member-declaration list*.

Each *declarator* gives the name of a structure variable. The *declarator* can also change the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

The second form uses a structure *tag* to refer to a structure type. The *member-declaration list* does not appear in this type of declaration because you have defined the structure type. The structure type definition must be visible for a *tag* declaration to be used, and the definition must appear prior to the *tag* declaration, unless the *tag* declares a pointer variable or a **typedef** structure type. These declarations can have a structure *tag* before you define the structure type, as long as the structure definition is visible to the declaration.

A *member-declaration list* is a list of one or more variables or bit-field declarations. Each variable declared in the *member-declaration list* is defined as a member of the structure type. Variable declarations within *member-declaration lists* have the same form as the variable declarations discussed in this chapter, except that the declarations do not contain storage class specifiers or initializers. The structure members can have any variable type: simple, array, pointer, union, or structure.

You cannot declare a member to have the type of the structure in which it appears. However, you can declare a member as a pointer to the structure type in which it appears. This lets you create linked lists of structures.

Bit-fields

A bit-field declaration has the following form:

type-specifier [*identifier*] : *constant-expression*;

IBM Extension

The type-specifier for a bit-field declaration must specify an unsigned integral type, except *enum*. If you give a signed type for a bit-field, the compiler issues a warning message and converts the type to unsigned.

Adjacent bit-fields are stored in the same storage unit having the same size as the base type, provided:

- The base types of the adjacent bit-fields are the same.
- There is space in that storage unit for the fields.

If either requirement is not met, no other fields are stored in the current storage unit. The next bit-field is stored in the next storage unit aligned appropriately for its base type.

If the previous member was a bit-field, a zero-width bit-field forces the compiler to stop allocating subsequent bit-fields from the current storage unit, and to align to a boundary appropriate to the type of the zero-width bit-field. Allocation for subsequent fields begins in the next storage unit whose alignment is appropriate to the base type of the next member.

End of IBM Extension

Example

```
struct s1 {
 unsigned char c1 : 4;  /* offset 0 */
 unsigned char c2 : 2;  /* packs into same byte as c1 */
 unsigned char c3 : 4;  /* doesn't fit, align to offset 1 */
 };

struct s2 {
 unsigned char c1 : 4; /* offset 0 */
 unsigned int i1 : 2; /* new base type, align to offset 2 */
 unsigned int i2 : 8; /* packs into same word as i1 */
 };

struct s3 {
 unsigned char bit : 4; /* offset 0 */
 unsigned char : 0; /* force align to next char boundary */
 char c;  /* offset 1 */
 };

struct s4 {
 unsigned char bit : 4; /* offset 0 */
 unsigned int : 0; /* force align to next int boundary */
 char c;  /* offset 2 */
 };

struct s5 {
 unsigned char bit : 4; /* offset 0 */
 unsigned char : 0; /* force align to next char boundary */
 int i;   /* align int boundary, offset 2 */
 };
```

The bit-field consists of the number of bits specified by *constant-expression*. The *constant-expression* must be a non-negative integer value. The width of the bit-field cannot be greater than the number of bits in the specified type. Arrays of bit-fields, pointers to bit-fields, and functions returning bit-fields are not allowed. The optional *identifier* names the bit-field. An unnamed bit-field whose width is specified as zero has a special function; it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

The identifiers in a structure declaration list must be unique within that list. Identifiers in the list need not be distinct from ordinary variable names or from identifiers in other structure declaration lists. Structure tags must be distinct from other structure, union, and enumeration tags having the same visibility.

Structure members are stored sequentially in the same order in which you declare them. The first member has the lowest storage address and the last member has the highest. The storage for each member begins on a storage boundary appropriate to its type. There-

fore, unnamed gaps can occur between the members of a structure in storage.

Bit-fields are not stored across boundaries of their declared type. For example, a bit-field declared with **unsigned int** type is either packed into the space remaining in the previous **int** or it begins a new **int**.

Example

The following example declares a structure variable named *complex*. This structure has two members, *x* and *y*, with **float** type. The structure type is not named.

```
struct
{
   float x,y;
} complex;
```

The next example declares a structure variable named *temp*. The defined structure has three members: *name*, *id*, and *class*. The *name* member is a 20-element array, and *id* and *class* are simple members with **int** and **long** type, respectively. The identifier **employee** is the structure tag.

```
struct employee
{
   char name[20];
   int id;
   long class;
} temp;
```

The following example declares three structure variables: *student*, *faculty*, and *staff*. Each structure has the same list of three members. The members are declared to have the structure type **employee**, defined in the previous example.

```
struct employee student, faculty, staff;
```

The next example declares a structure variable named *x* and defines the structure type **sample**. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, *next*, is declared as a pointer to the structure type that is defined as **sample**.

```
struct sample
{
   char c;
   float *pf;
   struct sample *next;
} x;
```

This example declares a two-dimensional array of structures named *screen*. The array contains 2000 elements. Each element is an individual structure containing four bit-field members: *icon*, *color*, *underline*, and *blink*.

```
struct
{
   unsigned icon : 8;
   unsigned color : 4;
   unsigned underline : 1;
   unsigned blink : 1;
} screen[25][80];
```

Union Declarations

A union declaration defines the name of the union variable and specifies a set of variable values (called members of the union) that can have different types. A variable with union type stores any single value defined by that type.

Format

```
union [tag] {member-declaration-list} declarator[, declarator...];
union tag declarator[,declarator...];
```

Union declarations have the same form as structure declarations, except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bit-field members are not allowed in unions. The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.

The storage associated with a union variable is the storage required for the longest member of the union. When you use a smaller member, the union variable might contain unused storage space. All members are stored in the same storage space and start at the same address. The stored value is overwritten each time a value is assigned to a different member. When a member of a union is accessed using a member of a different type, the result is undefined if the value stored and the value accessed have different base types or different sizes.

Example

This example declares a union variable named *number*. The union *number* has two members: *svar*, a signed integer, and *uvar*, an unsigned integer. This declaration lets the current value of *number* be stored as either a signed or an unsigned value. The union type is named **sign**.

```
union sign
{
   int svar;
   unsigned uvar;
} number;
```

The following example declares a union variable named *jack*. The members of the union are, in order, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage reserved for *jack* is the storage required for the 20-element array *f*, because *f* is the largest member of the union. The union type is unnamed.

```
union
{
   char *a, b;
   float f[20];
} jack;
```

This example declares a two-dimensional array of unions named *screen*. The array contains 2000 elements. Each element is an individual union with four members: *window1*, *window2*, *window3*, and *window4*, where each member is a structure. Each union element holds one of the four possible structure members at any given time. Thus, the *screen* variable is a composite of up to four different windows.

```
union
{
   struct
   {
      char icon;
      unsigned color : 4;
   } window1, window2, window3, window4;
} screen[25][80];
```

Array Declarations

A declaration for an array defines the name of the array and the type of each element. It can also define the number of elements in the array. A variable with array type is a pointer to the type of the array elements, as described in "Identifiers" on page 3-12.

Format

type-specifier declarator[*constant-expression*];
type-specifier declarator[ ];

Array declarations have two forms, as shown above. The *declarator* gives the name of the variable and can change the type of the variable. The brackets following the *declarator* change the declarator to array type. The *constant-expression* inside the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier*. The declarator may not specify a function. The type may be **void** only for a pointer declaration.

The integral type **unsigned int** is required to hold the *constant-expression* size of arrays. For arrays declared **huge**, the **sizeof** operator must be cast to type **long** to get the correct value.

The second form omits the *constant-expression* in brackets. You can use this form only if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Define arrays of arrays (multidimensional arrays) by giving a list of bracketed *constant-expressions* following the array declarator:

type-specifier declarator [*constant-expression*] [*constant-expression*]...;

Each *constant-expression* in brackets defines the number of elements in a given dimension. Two-dimensional arrays have two bracketed expressions; three-dimensional arrays have three, and so on. When you declare a multidimensional array within a function, you can omit the first *constant-expression* if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types by using complex declarators.

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing storage locations, from the first element to the last. No gaps occur between the elements of an array in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each.

```
char A[2][3];
```

The 3 columns of the first row are stored first, followed by the 3 columns of the second row.

To refer to an individual element of an array, use a subscript expression, for example:

```
A [0][1]
```

Example

The following example defines an array variable named *scores* with 10 elements, each of which has **int** type. The variable named *game* is declared as a simple variable with **int** type.

```
int scores[10], game;
```

The next example defines a two-dimensional array named *matrix*. The array has 150 elements, each having **float** type.

```
float matrix[10][15];
```

The next example defines an array of structures. This array has 100 elements. Each element is a structure containing two members.

```
struct
   {
   float x,y;
   } complex[100];
```

The next example defines an array of pointers. The array has 20 elements. Each element is a pointer to a **char** value.

```
char *name[20];
```

Pointer Declarations

A pointer declaration defines the name of the pointer variable and the type of the object to which the variable points.

Format

type-specifier *[*modification-spec*] *declarator*;

The *declarator* defines the name of the variable and can change its type. The *type-specifier* gives the type of the object. The type can be a simple type, a structure, a union, an *enum*, or void.

Pointer variables can also point to functions, arrays, and other pointers. To declare more complex pointer types, see "Complex Declarators" on page 5-9.

The *modification-spec* can be either **const** or **volatile**, or both. The specifier **const** implies that the program itself does not change the pointer, while **volatile** implies that another process beyond control of the program might change the pointer.

You can declare a pointer to a structure, union type, or **enum** before defining the structure or union type, as long as the structure, union type, or **enum** definition is visible at the time of the declaration. Such declarations are allowed because the compiler does not need to know the size of the structure, union, or **enum** to reserve space for the pointer variable. Declare the pointer by using the structure, union, or **enum** tag. See the fourth example.

A variable declared as a pointer holds a storage address. The amount of storage required for an address and the meaning of the address depend on the memory model in use. Pointers to different types might not have the same length.

In some cases the special keywords **near**, **far**, and **huge** are available to change the size of a pointer. For more information about special keywords, see "Declarators with Special Keywords" on page 5-12.

Example

Example 1 declares a pointer variable named *message*. It points to a variable with **char** type.

```
char *message;        Example 1
```

Example 2 declares an array of pointers named *pointers*. The array has 10 elements. Each element is a pointer to a variable with **int** type. The pointers are specified as **volatile**; any of them might be changed at run time by another process.

```
int * volatile pointers[10];        Example 2
```

Example 3 declares a pointer variable named *pointer*. It points to an array with 10 elements. Each element in this array has **int** type.

```
int (*pointer)[10];        Example 3
```

Example 4 declares two pointer variables that point to the structure type **list**. This declaration can appear before the definition of the **list** structure type (see next example), as long as the **list** type definition has the same visibility as the declaration.

```
struct list *next, *previous;        Example 4
```

Example 5 declares the variable *line* to have the structure type named **list**. The **list** structure type is defined to have three members. The first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another **list** structure.

```
struct list                                Example 5
{
   char *token;
   int count;
   struct list *next;
} line;
```

Function Declarations

A *function declaration* defines the name and return type of a function and can establish the types and number of arguments to the function. Function declarations, also called *forward declarations*, do not define the function body. Instead, they permit the characteristics of the function to be known before the function is defined.

Format

```
[type-specifier]declarator([arg-typelist]) [,declarator([arg-typelist])...];
```

The *declarator* of the function declaration names the function, and the *type-specifier* gives the return type of the function. If you omit the *type-specifier* from a function declaration, the compiler assumes the return type of the function is **int**.

Function declarations can include either the **extern** or the **static** storage class specifier.

Argument Type List: The *arg-typelist* establishes the number and types of the arguments to the function and can also declare identifiers for the arguments. It has one of the following forms:

type-name-list[,...]
identifier-list[,...]

The *type-name-list* is a list of one or more type names. Separate each *type-name* from the next by a comma. The first *type-name* gives the type of the first argument to the function; the second *type-name* gives the type of the second argument, and so on.

The *identifier-list* is a list of one or more sets of type name/identifier specifications, such as **float f**. Separate each such declaration from the next by a comma. The first set gives the type of the first argument to the function and declares the identifier; the second set declares the second identifier, and so on. If either *type-list* ends with a comma followed by three periods, the number of arguments to the function is variable.

Note: To maintain compatibility with previous versions, the compiler also accepts the comma character, without trailing periods, at the end of the *arg-typelist* to indicate a variable number of arguments. Use the comma only for compatibility; it is recommended that you use three periods for new code.

A *type-name* for a fundamental, void, structure, union, or *enum* type consists of the type specifier for that type (such as **int**). Form the *type-names* for pointers, arrays, and function by combining a type specifier with an "abstract declarator," that is, a declarator without an identifier. "Type Names" on page 5-48 explains how to form and interpret abstract declarators.

You can use the special keyword **void** in place of the arg-typelist to declare a function that has no arguments. The compiler produces a warning message if a call to the function or the function definition specifies arguments.

One other construction is allowed in the *arg-typelist*. The phrase **void** * specifies an argument of any pointer type. Use this phrase in the *arg-typelist* as if it were a *type-name*.

You can omit *arg-typelist*, but you still need the parentheses after the function identifier. In this form the function declaration establishes neither the number nor the types of arguments to the function. When

you omit this information, the compiler does not perform any type-checking between the arguments in a function call and the formal parameters of the function definition.

Return Type: Functions can return values of any type except arrays and functions. Thus, the type specifier of a function declaration can specify any fundamental, structure, **void**, **enum**, or **union** type. You can change the function identifier with one or more asterisks to declare a pointer return type. Although functions cannot return arrays and functions, they can return pointers to arrays and functions. Functions that return pointers to array or function types are declared by the function identifier with asterisks, brackets, and parentheses to form a complex declarator.

Example

Example 1 declares a function named **add**, that takes two **int** arguments and returns an **int** value.

```
int add(int, int);          Example 1
```

Example 2 declares a function named **strfind**, which returns a pointer to a **char** value. The function takes at least one argument, a pointer to a **char** value. The argument type list ends with a comma, showing that the function can take more arguments.

```
char *strfind(char *,);     Example 2
```

Example 3 declares a function with **void** return type (returning no value). The argument-type-list is also **void**, meaning no arguments are expected for this function.

```
void draw(void);            Example 3
```

Example 4 **sum** is declared as a function returning a pointer to an array of three **double** values. The **sum** function takes two arguments, each a **double** value.

```
double (*sum(double, double))[3];    Example 4
```

In Example 5, the function named **select** is declared to take no arguments and return a pointer to a function. The pointer return value points to a function taking one **int** argument and returning an **int** value.

```
int (*select(void))(int)          Example 5
```

In example 6, the function **prt** is declared to take a pointer argument of any type and to return an **int**. Either the **char** pointer *p* or the **short**

pointer *q* can pass as an argument to **prt** without producing a type-mismatch warning.

```
char *p;                    Example 6
short *q;
int prt(void *);
```

Storage Classes

The storage class of a variable determines whether the item has a global or local lifetime. An item with a global lifetime exists and has a value throughout the program. All functions have global lifetimes.

New storage is reserved for variables with local lifetimes each time control passes to the block they are defined in. When flow passes out of the block, the variables no longer have meaningful values.

Although C defines only two types of storage classes, four storage class specifiers are available:

Type	**Specifiers**
Local	Auto and register
Global	Extern and static.

The four storage class specifiers have distinct meanings because storage class specifiers affect the visibility of functions and variables as well as their storage class. Visibility and the related concept of lifetime are discussed in "Lifetime and Visibility" on page 4-5.

The placement of variable or function declarations within source files also affects storage class and visibility. Declarations outside of all function definitions occur at the *external level*; declarations within function definitions occur at the *internal level*.

The meaning of each storage class specifier depends on whether the declaration occurs at the external or the internal level and whether the item declared is a variable or a function. The following sections describe the meaning of storage class specifiers in each kind of declaration. They also explain the default behavior when you omit the storage class specifier from a variable or function declaration.

Variable Declarations at the External Level

Variable declarations at the external level use the **static** and **extern** storage class specifiers or omit the storage class specifier entirely. The **auto** and **register** storage class specifiers are not allowed at the external level.

Variable declarations at the external level are either *definitions* of variables or *references* to variables defined elsewhere. An external variable declaration that also initializes the variable (implicitly or explicitly) is a definition of the variable. Definitions at the external level can take several forms:

- You can define a variable at the external level by declaring it with the **static** storage class specifier. You can explicitly initialize the **static** variable. If you omit the initializer, the variable is automatically initialized to 0 at compile time. Thus, **static int k = 16;** and **static int k;** are both definitions.
- A variable is defined when it is explicitly initialized at the external level. For example, **int j = 3;** is a variable definition.

Whenever you define a variable at the external level, it is visible throughout the remainder of the source file in which it appears. The variable is not visible above its definition in the same source file.

A variable can be defined at the external level only once within a source file. If you use the **static** storage class specifier, you can define another variable with the same name with the **static** storage class specifier in a different source file. Because each **static** definition is visible only in its own source file, no conflict occurs.

Use the **extern** storage class specifier to declare a *reference* to a variable defined elsewhere. You can use these declarations to make a definition in another source file visible or to make a variable visible above its definition in the same source file. Whenever you declare a reference to the variable at the external level, the variable is visible throughout the remainder of the source file where the declared reference occurs.

For an **extern** reference to be valid, you must define the variable to which it refers once, and only once, at the external level. The definition can be in any of the source files that make up the program. If

an **extern** declaration has an initializer, it is treated as the definition of the variable.

The rules outlined here do not cover one special case; you can omit both the storage class specifier and the initializer from a variable declaration at the external level. For example, the declaration **int n;** is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

- If you define a variable by the same name at the external level elsewhere in the program, the declaration is taken to be a reference to that variable, exactly as if you had used the **extern** storage class specifier in the declaration.
- If no such definition is present, the declared variable reserves storage at link time and initializes to 0. If more than one such declaration appears in the program, storage is reserved for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of *i* at the external level, **int i** and **char i**, storage space for an **int** is reserved for *i* at link time.

Example

```
/*************************************
            SOURCE FILE ONE
*************************************/

extern int i;
       /* Reference to i, defined below */

main()
{
   i++;
   printf("%d\n", i); /* i equals 4 */
   next();
}

int i = 3;       /* Definition of i */

next()
{
   i++;
   printf("%d\n", i); /* i equals 5 */
   other();
}
/*************************************
            SOURCE FILE TWO
*************************************/

extern int i;  /* Reference to i in first source file */

other()
{
   i++;
   printf("%d\n", i); /* i equals 6 */
}
```

The two source files contain a total of three external declarations of *i*. Only one declaration contains an initialization. The declaration **int i = 3;** defines the global variable *i* with initial value 3. The **extern** declaration of *i* at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the **main** function cannot refer to the global variable *i*. The **extern** declaration of *i* in the second source file makes the global variable visible in that source file.

All three functions perform the same task. They increase *i* and print it. The **printf** function is defined eisewhere in the program. The values printed are 4, 5, and 6.

If the variable *i* is not initialized, it automatically becomes 0 at link time. The values printed in this case are 1, 2, and 3.

Variable Declarations at the Internal Level

You can use any of the four storage class specifiers for variable declarations at the internal level. When you omit the storage class specifier from a variable declaration at the internal level, the default storage class is **auto**.

The **auto** storage class specifier declares a variable with a local lifetime. The variable is visible only in the block in which you declare it. Declarations of **auto** variables can include initializers, as discussed later in this chapter. Variables with **auto** storage class are not initialized automatically. Explicitly initialize them when you declare them or assign them initial values in statements within the block. If you do not initialize them, the values of **auto** variables are undefined.

The **register** storage class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually results in faster access time and smaller code size. Variables declared with **register** storage class have the same visibility as **auto** variables.

The number of registers for variable storage depends on the available machine. If no registers are available when the compiler meets the **register** declaration, the variable is given the **auto** storage class and stored. The compiler assigns register storage to variables in exactly the same order in which the declarations appear in the source file. Register storage, if available, is guaranteed only for **int** and in the pointer types of some memory models.

A variable declared at the internal level with the **static** storage class specifier has a global lifetime. The variable is visible only within the block in which you declare it. Unlike **auto** variables, variables declared as **static** retain their values when the block is exited.

Declarations of **static** variables can include initializers. If not explicitly initialized, a **static** variable is automatically set to 0. Initialization is performed once, at link time; the **static** variable is not reinitialized each time the block is entered.

A variable declared with the **extern** storage class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The purpose of the internal **extern** declaration is to make the external-level variable definition visible within the block. The internal **extern** declaration does not

change the visibility of the global variable in any other part of the program and may not have an initializer.

Example

The following example shows the declaration of functions at the internal level:

```
int i = 1;

main()
{       /* Reference to i, defined above */
   extern int i;

        /* Initial value is zero;
        ** a is visible only within main */
   static int a;

        /* b is stored in a register,
        ** if possible */
   register int b = 0;

        /* Default storage class is auto */
   int c = 0;

        /* Values printed are 1, 0, 0, 0 */
   printf("%d\n%d\n%d\n%d\n",
     i, a, b, c);
   other();
}

other()
{
        /* i is redefined */
   int i = 16;

        /* This a is visible only within other */
   static int a = 2;

   a += 2;
        /* Values printed are 16, 4 */
   printf("%d\n%d\n", i, a);
}
```

The variable *i* is defined at the external level with the initial value 1. A reference to the external-level *i* is declared in the **main** function with an **extern** declaration. The **static** variable *a* is automatically set to 0 because the initializer is omitted. The call to **printf** (assuming the **printf** function is defined elsewhere in the source program) prints out the values 1, 0, 0, 0.

In the **other** function, the variable *i* is redefined as a local variable with the initial value 16. This does not affect the value of the

external-level *i*. The variable *a* is declared as a **static** variable and initialized to 2. This *a* does not conflict with the *a* in **main**, because the visibility of **static** variables at the internal level is restricted to the block in which you declare them.

The variable *a* is increased by 2, giving 4 as the result. If you call the **other** function again in the same program, the initial value of *a* is 4. Internal **static** variables retain their values when the block in which they are declared is exited and reentered.

Function Declarations at the External and Internal Levels

Function declarations can use the **static** or the **extern** storage class specifier. Functions have global lifetimes.

The visibility rules for functions are different from the rules for variables. Function declarations at the internal level have the same meaning as function declarations at the external level. This means that functions cannot have block visibility, and the visibility of functions cannot be nested. A function declared to be **static** is visible only within the source file in which you define it. Any function in the same source file can call the **static** function, but functions in other source files cannot call the **static** function. You can declare another **static** function by the same name in a different source file without conflict.

Unless you compile with the /Za option, functions declared as **extern** are visible throughout all source files in the program. Any function can call an **extern** function. When you compile using /Za, functions declared as **extern** within a block are visible only within that block. See the "Advanced Topics" section in Chapter 2 of *IBM C/2 Compile, Link, and Run* book for further information about the /Za option.

Function declarations that omit the storage class specifier default to **extern**.

When a program redefines a reserved external identifier, the effect depends on the identifier being replaced. Even if the replacement meets the functional specification of the replaced function, the results may be unpredictable because the IBM C/2 library functions may have undocumented side effects that are necessary for the correct operation of the function and/or related functions.

Initialization

You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. Precede the initializer by an equal sign (=), as shown below:

= *initializer*

You can initialize variables of any type with the restrictions outlined below. Functions do not take initializers.

Declarations that use the **extern** storage class specifier cannot contain initializers.

You can initialize variables declared at the external level. If you do not explicitly initialize them, they are set to 0 at link time. You can initialize any variable declared with the **static** storage class specifier. Initializations of **static** variables are performed once, at link time. If you do not explicitly initialize **static** variables, they are set to 0.

Initializations of **auto** and **register** variables are performed each time control passes to the block in which you declared them. If you omit the initializer from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.

Initializations of **auto** aggregate types (arrays, structures, and unions) are prohibited. You can initialize only **static** aggregates and aggregates declared at the external level.

The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. Constant expressions are described in "Constant Expressions" on page 6-9. You can initialize automatic and register variables with constant or variable values.

Simple and Pointer Types

To initialize simple and pointer types, assign an expression to the variable in the following form:

Format

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

Example
In this example, *x* is initialized to the constant-expression 10.

```
int x = 10;
```

In the next example, the pointer *px* is initialized to 0, producing a null pointer.

```
register int *px = 0;
```

The following example uses a constant expression to initialize c.

```
int c = (3 * 1024);
```

The next example initializes the pointer *b* with the address of another variable, *x*.

```
int *b = &x;
```

Aggregate Types

To initialize aggregate types, assign an initializer list to the aggregate in the following form:

Format

= {*initializer-list*}

The *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or another initializer list. Any brace enclosed list can appear within an *initializer-list*. This is useful for initializing aggregate members of an aggregate, as shown in the examples that follow.

For each *initializer-list*, the values of the constant expressions are assigned in order to the members of the aggregate variable. When a **union** is initialized, the *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If fewer values are in an *initializer-list* than are in the aggregate type, the remaining members or elements are initialized to 0. Giving too many initial values for the aggregate type causes an error. These rules apply to each embedded *initializer-list*, as well as to the aggregate as a whole.

Example

The following example declares *p* as a 4 x 3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row.

```
int p[4][3] =
{
   { 1, 1, 1 },
   { 2, 2, 2 },
   { 3, 3, 3 },
   { 4, 4, 4 },
};
```

The *initializer-list* for the third and fourth rows contain commas after the last constant expression. The last *initializer-list* ({4, 4, 4},) is also followed by a comma. These extra commas are permitted but not required. The required commas are those that separate constant expressions and *initializer-lists*.

If there is not an embedded *initializer-list* for an aggregate member, values are assigned in order to each member of the subaggregate. The preceding initialization is equal to:

```
int p[4][3] =
{
   1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list. In the next example, the three **int** members of *x* are initialized to 1, 2, and 3, respectively. The three elements in the first row of *m* are initialized to 4.0; the elements of the remaining row of *m* are initialized to 0 by default.

```
struct list
{
   int i, j, k;
   float m[2][3];
} x = {
         1,
         2,
         3,
         {4.0, 4.0, 4.0}
};
```

In the final example, the union variable *y* is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list {'1'} gives values to the first row of the array. Because only one value appears in the list, the element in the first column is initialized to 1, and the remaining two elements in the row are initialized to 0 (the null character), by default. Similarly, the

first element of the second row of *x* is initialized to 4 and the remaining two elements in the row are initialized to 0.

```
union
{
   char x[2][3];
   int i, j, k;
} y = {
       {'1'},
       {'4'}
};
```

String Initializers

You can initialize an array with a string literal, for example:

```
char code[ ] = "abc";
```

This initializes *code* as a four-element array of characters. The fourth element is the null character '**\0**' that ends all string literals.

If you specify the array size and the string is longer than the specified size of the array, the extra characters are discarded. The following declaration initializes *code* as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to *code*. The character **d** and the null character are discarded.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to 0.

Type Declarations

A type declaration defines the name and members of a structure or union type or the name and enumeration set of an enumeration type. You can use the name of a declared type in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. These declarations set up shorter or more meaningful names for types already defined by C or for types declared by the user.

Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. In type declarations the variable identifier is omitted, because no variable is declared. The tag is mandatory; it names the structure, union, or enumeration type. The member declaration list or *enum* list defining the type must appear in the type declaration. The short form of variable declarations, in which a tag refers to a type defined elsewhere, is not legal for type declarations.

This example declares an enumeration type named **status**. You can use the name of the type in declarations of enumeration variables. The identifier *loss* is explicitly set to −1. Both *bye* and *tie* are associated with the value 0, and *win* is given the value 1.

```
enum status
{
   loss = -1,
   bye,
   tie = 0,
   win
};
```

The next example declares a structure type named **student**:

```
struct student
{
   char name[20];
   int id, class;
};
```

Because the structure type **student** is defined, the next example declares a variable as having **student** type:

```
struct student employee;
```

You cannot define a type within the formal list of a function. For example, the following example causes an error:

```
void func( struct s { int a, b; } st );
void func( struct s st )
{
}
```

This rule applies to union and enumeration declarations as well as structure declarations. Declare the type in conventional fashion, outside the function's formal list.

Note: The preceding code is acceptable under the ANSI standard, but the type is defined only to the closing parenthesis of the prototype and the closing brace (}) of the function definition.

Typedef Declarations

A **typedef** declaration is like a variable declaration except for the **typedef** keyword that appears in place of a storage class specifier. The declaration is interpreted in the same way as variable and function declarations. The identifier, instead of taking on the type specified by the declaration, becomes a new keyword for the type.

Format

typedef *type-specifier declarator* [,*declarator*...];

The **typedef** declaration does not create types. It creates synonyms for existing types or names for types that can be specified in other ways. You can declare any type with **typedef**, including pointer, function, and array types. A **typedef** name for a pointer to a structure or union type can be declared before the structure or union type is defined, as long as the definition has the same visibility as the declaration.

This example declares **WHOLE** to be a synonym for **int**.

```
typedef int WHOLE;
```

The following example declares **GROUP** as a structure type with three members. Because a structure tag, **club**, is also specified, either the **typedef** name, **GROUP**, or the structure tag can be used in declarations.

```
typedef struct club
{
   char name[30];
   int size, year;
} GROUP;
```

The next example uses the previous **typedef** name to declare a pointer type. The type **PG** is declared as a pointer to the **GROUP** type, which in turn is defined as a structure type.

```
typedef GROUP *PG;
```

The final example declares **DRAWF** as a function returning no value and taking two **int** arguments. This means, for example, that the declaration **DRAWF box;** is equivalent to the declaration **void box(int, int);**.

```
typedef void DRAWF(int, int);
```

Type Names

A *type name* specifies a particular data type. Use type names in the following contexts:

- In the argument-type lists of function declarations
- In type casts
- In **sizeof** operations.

See "Function Declarations" on page 8-9 for a discussion of argument-type lists.

The type names for simple, enumeration, structure, and union types are the type specifiers for those types. A type name for a pointer, array, or function type has the form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting solely of one or more pointer, array, or function modifier. The pointer modifier (*) always appears before the identifier in a declarator, while array brackets ([ ]) and function modifiers (( )) appear after the identifier. These rules define where the identifier appears in an abstract declarator and how to interpret the declarator accordingly. Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations. When you give a function type with an abstract declarator, you can include the argument type list of the function, which also consists of type names.

The abstract declarator, ( ), is not allowed alone because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (a function type). You may not cast objects to functions nor use **sizeof** to obtain the size of a function.

The type specifiers established through **typedef** declarations also qualify as type names.

Example
This example gives the type name for pointer to **long** type:

```
long *
```

The following examples show how parentheses change complex abstract declarators. This example gives the name for a pointer to an array of five **int** values:

```
int (*)[5]
```

This example names a pointer to a function taking no arguments and returning an **int** value:

```
int (*)(void)
```

Chapter 6. Forming Expressions and Making Assignments

This chapter describes how to form expressions and make assignments in the C language. An *expression* is a combination of operands and operators that yields (expresses) a single value. An *operator* specifies how to manipulate the constants or variables in an expression. An *operand* is a constant or variable value that an operator in an expression manipulates. Each operand of an expression is also an expression, because it represents a single value.

C assignments are expressions. An assignment yields a value. Its value is the value assigned. In addition to the simple assignment operator (=), C offers complex assignment operators that both transform and assign their operands.

The value resulting from the evaluation of an expression depends on the relative precedence of operators in the expression and side effects, if present. The precedence of operators determines the grouping of operands in an expression. *Side effects* are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which C evaluates the operands also affects the result of the expression.

The value represented by each operand in an expression has a type. You can convert this type to a different type in assignments, type casts, function calls, and operations.

Operands

A C operand is a constant, identifier, string, function call, subscript expression, member selection expression, or more complex expression formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a *constant expression*.

Every operand has a type. The following sections discuss the type of value that each kind of operand represents. An operand can be cast from its original type to another type by means of a *type cast* opera-

tion. A type cast expression can also form an operand in an expression.

Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant can have either **int**, **unsigned int**, **long**, or **unsigned long** type, depending on the size of the integer and how the value was specified. Floating-point constants always have **double** type. String literals are considered arrays of characters. See "Strings" on page 6-3 for a discussion of string literals.

Identifiers

An *identifier* names a variable or function. Every identifier has a type, established when you declare the identifier. The value of an identifier depends upon its type, as follows:

- Identifiers of integer and floating-point types represent values of the corresponding type.
- An identifier of **enum** type represents one constant value of a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.
- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.
- An identifier declared as a pointer represents a pointer to the specified type.
- An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer is the type of the first element of the array. For example, if you declare **series** to be a ten-element integer array, the identifier **series** expresses the address of the array, while the subscript expression **series[*n*]** (where *n* is an integer in the range 0 to 9) refers to a variable integer element of series.

  The address of an array does not change during the running of the program, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, and an array identifier cannot form the left-hand operand of an assignment operation.
- An identifier declared as a function represents the function itself; its type is a function returning a value of a specific type. Although it is declared and used in the program as though it had

the type of the value that it returns, it is only an operation. Use it optionally on one or more arguments to optionally return a value of a specific type. It is not a variable, which could be assigned a value. Therefore, function identifiers can appear only alone or in expressions as right-hand sides of assignments, never as left-hand sides.

Strings

A string literal consists of a list of characters enclosed in double quotes, as shown below:

```
"string"
```

A string literal is stored as an array of elements with **char** type. The string literal represents the address of the first element of the array. The address of the first element of the string is a constant, so the value represented by a string expression is a constant.

Because string literals are effectively arrays, you can use them in contexts that allow array values. String literals are subject to the same restrictions as arrays. String literals have two additional restrictions: they are not variables and cannot be left-hand operands in assignment operations.

The last character of a string is the null character \0. The null character is not visible in the string expression, but the system adds it as the last element when it stores the string. Thus, the string "abc" actually has four characters instead of three.

Function Calls

Use a function call expression to call a function and to pass arguments to a function.

Format

expression (*expression-list*)

A function call consists of an *expression* followed by an *expression-list* in parentheses, where *expression* evaluates to a function address (for example, a function identifier), and *expression-list* is a list of expressions whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

A function call expression has the value and type of the return value of the function. If the return type of the function is **void**, the function

call expression also has **void** type. If control returns from the called function without running a **return** statement, the value of the function call is undefined, unless the type is **void**.

Subscript Expressions

Use a subscript expression to access an element of an array.

Format

expression1[*expression2*]

A *subscript expression* represents the value at the address that is *expression2* positions beyond *expression1*. The value of *expression1* is any pointer value (such as an array identifier) and the value of *expression2* is an integer value. The value of *expression2* must be enclosed in brackets.

You can use subscript expressions to refer to array elements, or you can apply a subscript to any other pointer.

According to the conversion rules of the addition operator, the compiler converts the integer value to an address offset by multiplying it by the length of the type addressed by the pointer. For a one-dimensional array, the following four expressions are equivalent, assuming that *a* is a pointer and *b* is an integer.

```
a[b]
*(a + b)
*(b + a)
b[a]
```

In each case the address computes to

(b*sizeof(*type pointed to*))+a

For example, suppose the identifier line refers to an array of **int** values. Consider the following expression:

```
int line[15] ;
```

The compiler evaluates the expression by multiplying the integer value *i* by the length of an **int**. The converted value of *i* represents *i* **int** positions. The sum of this converted value and the original pointer value (*line*) yields an address that is offset *i* **int** positions from *line*.

As the last step in evaluating the subscript expression, the system applies the indirection operator to the new address. The result is the value of the array element at that position.

```
line [i]
```

The subscript expression:

```
line[0]
```

represents the value of the first element of *A*, because the offset from the address represented by *A* is 0. An expression such as:

```
line[5]
```

refers to the element offset five positions from *line* or the sixth element of the array.

Multidimensional Array References

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate left to right. The leftmost subscript expression, *expression1[expression2],* is evaluated first. The address that results from adding *expression1* and *expression2* forms the pointer expression to which *expression3* is added. The indirection operator (*) is applied after the last subscripted expression is evaluated. The indirection operator is not applied at all if the final pointer value addresses an array type.

Expressions with more than one subscript refer to elements of *multidimensional arrays*. A multidimensional array is an array whose elements are arrays. The first element of a three-dimensional array, for example, is an array with two dimensions.

Example

In the following examples, the array named *prop* has 3 elements, each of which is a 4-by-6 array of **Int** values.

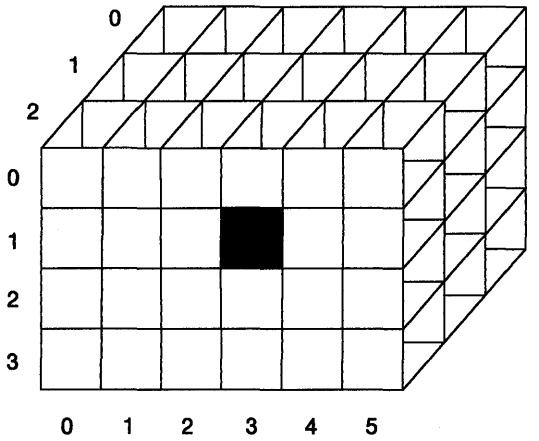
```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

The first example shows how to refer to the second individual **Int** element of *prop*. Arrays are stored by row, so the last subscript varies most quickly for accessing each element of the array.

```
i = prop[0][0][1];
```

The second example shows a more complex reference to an individual element of *prop*. To evaluate the expression, the compiler

multiplies the first subscript *2* by the size of the 4-by-6 **Int** array and adds it to the pointer value *prop*. The result points to the third 4-by-6 **Int** array.



Next the compiler multiplies the second subscript, *1*, by the size of the 6-element **Int** array and adds it to the address represented by *prop*[2]. The result points to a 6-element array, the second of four such arrays in the third 4-by-6 **Int** array.

Each element of the 6-element array is an **Int** value. The compiler multiplies the final subscript *3* by the size of an **Int** data element and adds it to *prop*[2][1]. The resulting pointer addresses the fourth element of the second 6-element array of the third 4-by-6 element array.

The last step in evaluating the expression *prop*[2][1][3] is applying the indirection operator to the pointer value. The result is the **Int** element at that address.

```
i = prop[2][1][3];
```

The next example does not apply the indirection operator. The expression:

```
prop [2][1]
```

is a valid reference to the 3-dimensional array *prop*. The result of the expression is a pointer value that addresses an array with one dimension. Because the pointer value addresses an array type, the indirection operator is not applied.

```
ip = prop[2][1];
```

The next example is a pointer array addressing an array with two dimensions. Again, because the pointer addresses an array, the indirection operator is not applied.

```
ipp = prop[2];
```

Member Selection Expressions

Member selection expressions refer to members of structures and unions. A member selection expression has the value and type of the selected member.

Format

expression.identifier
expression->identifier

In the first form, *expression.identifier*, the expression represents a value of a structure or union. The *identifier* names a member of the specified structure or union.

In the second form, *expression->identifier*, the expression represents a pointer to a structure or union. The *identifier* names a member of the specified structure or union.

The two forms of member selection expressions have a similar effect. Expressions involving the pointer selection operator (->) are shorthand versions of expressions using the period (.) in cases where the expression before the period consists of the indirection operator (*) applied to a pointer value. The following two statements are equal when *expression* is a pointer value:

expression -> identifier
(expression).identifier

Example

The following conditions apply to all examples:

```
struct pair
{
   int a;
   int b;
   struct pair *sp;
} item, list[10];
```

In the following example, the address of the *item* structure is assigned to the *sp* member of the structure. This means that *item* contains a pointer to itself.

```
item.sp = &item;
```

In the next example, the pointer expression *item.sp* is used with the pointer selection operator (->) to assign a value to the member *a*.

```
(item.sp)->a = 24;
```

The final example shows how to select an individual structure member from an array of structures.

```
list[8].b = 12;
```

Expressions with Operators

Expressions with operators can be unary, binary, or ternary expressions. A *unary expression* consists of an operand prefixed by a unary operator or an operand enclosed in parentheses and preceded by the **sizeof** keyword:

unary-operator operand

or

sizeof (*operand*)

A *binary expression* consists of two operands joined by a binary operator:

operand binary-operator operand

A ternary expression consists of three operands joined by the ternary (**?** **:**) operator:

operand **?** *operand* **:** *operand*

Assignment expressions use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators. The binary assignment operators are the simple assignment operator (=) and the compound assignment operators. Each compound assignment operator is a combination of another binary operator with the simple assignment operator. The forms of assignment expressions are:

operand++
operand - -
++*operand*
- - *operand*
operand = *operand*
operand compound-assignment-operator operand

Expressions in Parentheses

You can enclose any operand in parentheses. The parentheses have no effect on the type or value of the enclosed expression. For example, in the expression:

```
(10 + 5) / 5
```

the parentheses around **10 + 5** mean that the value of **10 + 5** is the left operand of the **/** (division) operator. The result of **(10 + 5) / 5** is 3. Without the parentheses, **10 + 5 / 5** evaluates to 11.

Although parentheses affect the way the compiler groups operands in an expression, they cannot guarantee a particular order of evaluation for the expression.

Type-Cast Expressions

A type-cast expression has the following form:

(type-*name*) *operand*

A *type cast* is an explicit assignment of data type to an existing variable of another data type. (See "Type Specifiers" on page 5-2 for a description of data types and data-type specifiers.)

Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts to integer and floating-point types, and other constant-expressions. You can combine and change the operands using operators with some restrictions.

Constant expressions cannot use assignment operators or the binary sequential evaluation operator (,). You can use the unary address-of operator (&) only in certain initializations.

These restrictions also apply to constant expressions used to initialize variables at the external level. The compiler allows such expressions to apply the unary address-of operator (&) to other external-level variables with fundamental, structure, and union types and to external level arrays subscripted with a constant expression. In these expressions, a constant expression not involving the

address-of operator can be added to or subtracted from the address expression.

Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator).

Unary operators prefix their operand and associate right to left. The unary operators of the C language are:

- ~ !	Complement operators
* &	Indirection and address-of operators
sizeof	Size operator.

Binary operators associate left to right. The binary operators are:

* / %	Multiplicative operators
+ −	Additive operators
<< >>	Shift operators
< > <= > = = = !=	Relational operators
& | ^	Bitwise operators
&& | |	Logical operators
,	Sequential evaluation operator.

C has one ternary operator, the conditional operator (**?** **:**). It associates right to left.

Standard Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These conversions are known as standard arithmetic conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are the standard arithmetic conversions. The discussion of each operator in the following sections specifies whether the operator performs the arithmetic conversions and also specifies additional conversions, if any, that the operator performs.

"Type Conversions" on page 6-31 outlines the path of each type of conversion.

Arithmetic conversion proceeds in the following order:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **double** type, the other operand is converted to **double**.
3. Any operands of **char** or **short** type are converted to **int**.
4. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
5. If one operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. If one operand is of type **long**, the other operand is converted to **long**.
7. If one operand is of type **unsigned int**, the other operand is converted to **unsigned int**.
8. If both operands have type **int**, the result has type **int**. If the expression is the right-hand side of an assignment statement, the result is computed with type **int** before the assignment is made, even though the target variable has a wider type such as **unsigned** or **long**.

Unary Operators

Arithmetic Negation (–): Produces the value of its operand and forces the grouping of the operations enclosed in parentheses. The operand must be an integral or floating-point value. This operation performs the standard arithmetic conversions described above.

Bitwise Complement (~): Produces the bitwise complement of its operand. The operand must be of integral type. This operation performs the standard arithmetic conversions. The result has the type of the operand after conversion.

Logical NOT (!): Produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (zero). The result has **int** type. The operand must be an integral, floating-point, or pointer value.

Unary Plus (+): Produces the value of its operand and forces the grouping of the operations enclosed in parentheses. It is used with expressions involving more than one associate or commutative binary operator. Use it for considerations of overflow or loss of precision in the values of the variables in the expressions.

Example
In the following example, the new value of *x* is the negative of 987, or -987:

```
short x = 987;
    x = -x;
```

In the next example, the new value assigned to *y* is the ones' complement of the unsigned value 0xaaaa, or 0x5555:

```
unsigned short y = 0xaaaa;
    y = ~y;
```

In the next example, if *x* is greater than or equal to *y*, the result of the expression is 1 (true). If *x* is less than *y*, the result is 0 (false):

```
if (!(x < y));
```

In the next example, the compiler must honor the parenthesis grouping of assignment to e because of the presence of the unary plus operator. The assignment to *d* might be computed in the sequence: add *a* to *c*; then add *b* to the result and assign to *d*.

```
int a, b, c, d, e;
d=a + (b + c);
e=a + +(b + c);
```

Indirection and Address-of Operators

Indirection (*): Gets access to a value indirectly through a pointer. The operand must be a pointer value. The result of the operation is the value to which the operand points. The result type is the type addressed by the pointer operand. If the pointer value is null, the result is undefined.

Address-of (&): Takes the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.: You cannot apply the address-of operator to a bitfield member of a structure, nor can you apply it to an identifier declared with the **register** storage class specifier.

Example
The following declarations and initial values are true for all four examples:

```
int *pa, x;
int a[20];
double d;
```

In the first example, the address-of operator (&) takes the address of the sixth element of the array *a*. The result is stored in the pointer variable *pa*.

```
pa = &a[5];
```

The indirection operator is used in the next example to get access to the **int** value at the address stored in *pa*. The value is assigned to the integer variable *x*.

```
x = *pa;
```

The next example prints the word "True." This example tests whether applying the indirection operator to the address of *x* yields the same value as *x*.

```
if (x == *&x)
printf("True");
```

The following example shows a useful application of the rule shown in the last example. A type cast converts the address of *x* to a **pointer** to a **double**. C then applies the indirection operator, yielding the result of the expression, a **double** value.

```
d = *(double *)(&x);
```

Sizeof Operator

The **sizeof** operator determines the amount of storage associated with an identifier or a type. A **sizeof** expression has the form:

```
sizeof(name)
```

where *name* is either an identifier or a type name. The type name cannot be a **void**, function, or bit-field type.

When you apply the **sizeof** operator to an array identifier, the result is the size of the entire array in bytes rather than the size of the pointer represented by the array identifier.

When you apply the **sizeof** operator to a structure or union type name or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union. This may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the members.

Example

```
buffer = calloc(100, sizeof (int));
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The preceding example uses the **sizeof** operator to pass the size of an **int**, which varies across machines, as an argument to a function named **calloc**. The value returned by the function is stored in a buffer.

Multiplicative Operators

The *multiplicative* operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator must be integral; the multiplication and division operators take integral and floating-point operands. The types of the operands can be different. The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

The conversions performed by the multiplicative operators make no provision for overflow or underflow. Information is lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*): Specifies that its two operands are to be multiplied.

Division (/): Specifies that its first operand is to be divided by the second. When two integers are divided, the result, if not an integer, is cut off. If both operands are positive or unsigned, the result is cut off toward 0. The direction of truncation when either operand is negative can be either toward or away from 0, depending on the implementation. Division by 0 gives an error either at compile or run time.

Remainder (%): Gives the remainder when the first operand is divided by the second. The sign of the remainder is the same as the sign of the first operand.

Example
The following declarations and initial values are true for all three examples:

```
int i = 10, j = 3, n;
double x = 2.0, y;
```

In the following example, *x* is multiplied by *i* to give the value 20.0. The result has double type.

```
y = x * i;
```

In the next example, 10 is divided by 3. The result is cut off toward 0, yielding the integer value 3.

```
n = i / j;
```

In the following example, *n* is assigned the integer remainder 1 when 10 is divided by 3.

```
n = i % j;
```

Additive Operators

The additive operators perform addition and subtraction. The operands can be integral or floating-point values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator. The standard arithmetic conversions are performed on integral and floating-point operands. The type of the result is the type of the operands after conversion.

The conversions performed by the additive operators make no provision for overflow or underflow. Information is lost if C cannot represent the result of an additive operation in the data type of the operands after conversion.

Addition Operator (+): Specifies addition of its two operands. The operands can have integral or floating-point types, as described above, or one operand can be a pointer and the other an integer. When you add an integer to a pointer, C converts the integer value *i* by multiplying it by the size of the type addressed by the pointer. After conversion, the integer value represents *i* storage positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value expressing the address *i* positions from the original address. The new pointer value addresses the same type as the original pointer value.

Subtraction Operator (–): Subtracts its second operand from the first. The operands can be integral or floating-point values, as described above. The subtraction operator also allows the subtraction of an integer from a pointer value and the subtraction of two pointer values.

When an integer value is subtracted from a pointer value, the same conversions take place as with addition of a pointer and integer. The subtraction operator converts the integer value with respect to the type addressed by the pointer value. The result is the storage address *i* positions before the original address, where *i* is the integer value and each position is the length of the type addressed by the pointer value. The new pointer points to the type addressed by the original pointer value.

Two pointer values can be subtracted if they point to the same type. The difference between the two pointers is converted to a signed integer value by dividing the difference by the length of the type the pointers address. The result represents the number of storage positions of that type between the two addresses. The result is meaningful only for two elements of the same array.

Pointer Arithmetic: Additive operations involving a pointer and an integer generally give meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only storage positions of the same size lie between the original address and the address plus offset.

This assumption is valid for array members. An *array* is a series of values of the same type whose elements reside in contiguous storage locations. C does not guarantee that it stores data types without blank segments of storage except for array elements. That is, gaps can occur between storage positions, even positions of the same type. Adding to or subtracting from addresses of any values but array elements gives unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that only values of the same type, with no gaps, lie between the two addresses given by the operands. The difference of any two pointers is of type **int**. The difference between any two pointers to an array declared as **huge** is of type **long**, and the result of the difference calculation must be cast to **long** in order to get the correct value. This means that for arrays larger than 32KB and less than 64KB, you cannot take the difference of any two arbitrary pointers and get a correct value, since it may be greater than 32KB.

This can be done by casting the two pointers to be **huge** pointers, taking the difference, casting it to **long**, and assigning the result to an **unsigned int**. For example,

```
int arr[20000];
int *ip = &arr[20000];
unsigned diff;
...
diff = (long) ((int huge *)ip - (int huge *)arr);
```

will correctly set **diff** to 20000. Additive operations between pointer and integer values on machines with segmented architecture must take the segment addressing conventions into account. In some cases these operations may not be valid. See the "Working with Storage Models /A" section in Chapter 2 of *IBM C/2 Compile, Link, and Run* for more information.

Example

The following declarations and initial values are true for both examples:

```
int i = 4, j;
float x[10];
float *px;
```

The following example adds the integer *i* to the address of the fifth element of *x*, *x*[4]. It does this by first multiplying *i* by the length of a **float**, and then adding the result to the address of *x*[4]. The result is the address of *x*[8], the *i*'th element beyond *x*[4].

```
px = &x[4] + i;
```

In the next example, C subtracts the address of the third element of *x* (*x*[i-2]) from the address of the fifth element of *x* (*x*[*i*]). The difference is divided by the length of a **float**. The result is the integer value 2.

```
j = &x[i] - &x[i-2];
```

Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

For leftward shifts, the vacated right bits are filled with 0's. In a rightward shift, the method of filling left bits depends on the type (after conversion) of the first operand. If it is unsigned, vacated left bits are filled with 0's. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative or if the second operand specifies a shift count greater than or equal to the width in bits of the converted first operand.

The conversions performed by the shift operators make no provision for overflow or underflow conditions. Information is lost if C cannot represent the result of a shift operation in the data type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;

y = 0x5500;

z = (x << 8) + (y >> 8);
```

The above example shifts *x* left by 8 positions and shifts *y* right by 8 positions. It then adds the shifted values and assigns the resulting value, 0xaa55, to *z*.

Relational Operators

The binary relational operators test their first operand against the second to determine if the relation specified by the operator holds true. The result of a relational expression is either 1 (if the tested relation holds) or 0 (if it does not). The type of the result is **int**. The relational operators test the following relationships:

Operator	Relationship
<	The first operand is less than the second.
>	The first operand is greater than the second.
<=	The first operand is less than or equal to the second.
>=	The first operand is greater than or equal to the second.
==	The first operand is equal to the second.
!=	The first operand is not equal to the second.

The operands can have integral, floating-point, or pointer type. The types of the operands can be different. The usual arithmetic conversions are performed on integer and floating-point operands.

One or both operands of the equality (==) and inequality (!=) operators can have **enum** type. C converts an **enum** value in the same manner as an **int** value.

The operands of any relational operator can be two pointers to the same type. For the equality and inequality operators, the result of the comparison reflects whether the two pointers address the same storage location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects the relative position of two storage addresses.

Because the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are meaningless. Comparisons between the addresses of different elements of the same array can be useful, however, because C guarantees storage of array elements in order and without gaps from the first element to the last. The address of the first array element is less than the address of the last element.

C can compare a pointer value for equality or inequality to the constant value 0. A pointer with a value of 0 does not point to a storage location; it is called a null pointer.

Example

```
int x = 0, y = 0;

1.  x < y

2.  x > y

3.  x <= y

4.  x >= y

5.  x == y

6.  x != y
```

When *x* and *y* are equal, expressions 3, 4, and 5 have the value 1, and expressions 1, 2, and 6 have the value 0.

Bitwise Operators

The bitwise operators perform bitwise AND (&), inclusive OR (|), and exclusive OR (^) operations. The operands of bitwise operators must have one of the integral types, but their types can be different. The usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.

Bitwise AND (&): Compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, C sets the corresponding bit of the result to 1. Otherwise, it sets the corresponding result bit to 0.

Bitwise Inclusive OR (|): Compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is 1, C sets the corresponding bit of the result to 1. Otherwise, both bits are 0's, and C sets the corresponding result bit to 0.

Bitwise Exclusive OR (^): Compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is 0 and the other bit is 1, C sets the corresponding bit of the result to 1. Otherwise, C sets the corresponding result bit to 0.

Example

```
short i = 0xab00;
short j = 0xabcd;
short n;

1.  n = i & j;

2.  n = i | j;

3.  n = i ^ j;
```

The result assigned to *n* in the first example is the same as *i*, 0xab00. The bitwise inclusive OR in the second example results in the value *0xabcd*, while the bitwise exclusive OR in the third example produces *0x00cd*.

Logical Operators

The logical operators perform logical AND (&&) and OR (| |) operations. The operands of the logical operators must have integral, floating-point, or pointer type. The types of the operands can be different.

C evaluates the operands of logical AND and OR expressions from left to right. If the value of the first operand is enough to determine the result of the operation, C does not evaluate the second operand.

These operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand, comparing it to 0. A pointer has a value of 0 only if you explicitly set it to 0 through assignment or initialization.

The result of a logical operation is either 0 or 1; the type of the result is **int**.

Logical AND (&&): Produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical AND operation has a value of 0, C does not evaluate the second operand.

Logical OR (||): Performs an inclusive OR on its operands. It produces the value 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical OR operation has a nonzero value, C does not evaluate the second operand.

Example
The following declaration is true for both examples:

```
int x, y;
```

In the following example, the **printf** function prints a message if *x* is less than *y* and *y* is less than *z*. If *x* is greater than *y*, C does not evaluate *y* < *z* and prints nothing.

```
if (x < y && y < z)
   printf ("x is less than z\n");
```

In the next example, C prints a message if *x* is equal to either *y* or *z*. If *x* is equal to *y*, C does not evaluate **x = = z**.

```
if (x == y || x == z)
   printf ("x is equal to either y or z\n");
```

Sequential Evaluation Operator

The sequential evaluation operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The operands can be any type. The operator performs no conversions.

You use this operator (also called the comma operator) to evaluate two or more expressions in contexts that allow only one expression to appear.

Example
In the following example, C independently evaluates each operand of the third expression of the **for** statement. It evaluates the left operand *i* += *i* first and then the other operand, *j* --.

```
for ( i = j = 1; i + j < 20; i += i, j--);
```

As shown in the next example, the comma character can separate arguments. In the first function call, three arguments, separated by commas, are passed to the called function:

```
x, y + 2, and z.

f(x, y + 2, z);

f((x--, y + 2), z);
```

Do not confuse the use of the comma character as a separator with its use as an operator; the two functions are completely different.

In the second function call, parentheses force the compiler to interpret the first comma as the sequential evaluation operator. This func-

tion call passes two arguments to **f**. The first argument is the result of the sequential evaluation operation (*x*--, *y* + 2), which has the value and type of the expression *y* + 2; the second argument is *z*.

Conditional Operator

C has one ternary operator, the conditional operator **?** **:**.

Its form is:

operand1 ? *operand2* : *operand3*

Operand1 is evaluated in terms of its equivalence to 0. It must have integral, floating-point, or pointer type. If *operand1* has a nonzero value, *operand2* is evaluated and the result of the expression is the value of *operand2*. If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*. Either *operand2* or *operand3* is evaluated but not both.

The type of the result depends on the types of the second and third operands, as follows:

- If both the second and third operands have integral or floating-point type (their types can be different), the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.
- Both the second and third operands can have the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.
- One of the second or third operands can be a pointer and the other a constant-expression with the value 0. The type of the result is the pointer type.

Example

```
j = (i < 0) ? (-i) : (i);
```

The above example assigns the absolute value of *i* to *j*. If *i* is less than 0, *-i* is assigned to *j*. If *i* is greater than or equal to 0, *i* is assigned to *j*.

Assignment Operators

Assignment operators in C can both transform and assign values in a single operation. Using a compound assignment operator to replace two separate operations can reduce code size and improve program efficiency. The following is a list and description of assignment operators:

+ +	Unary increment operator
– –	Unary decrement operator
=	Simple assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator
+ =	Addition assignment operator
-=	Subtraction assignment operator
<<=	Left shift assignment operator
>>=	Right shift assignment operator
&=	Bitwise AND assignment operator
|=	Bitwise inclusive OR assignment operator
^=	Bitwise exclusive OR assignment operator.

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific path of the conversion depends on the two types and is outlined in detail in "Type Conversions" on page 6-31.

lvalue Expressions

An assignment operation specifies that the value of the right-hand operand is to be assigned to the storage location named by the left-hand operand. Thus, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression referring to a storage location. Expressions that refer to storage locations are *lvalue* expressions. A variable name is such an expression; the name of the variable denotes a storage location, while the value of the variable is the value at that location.

The C expressions that may be **lvalue** expressions are:

- Identifiers of character, integer, floating-point, pointer, enumeration, structure, or union type
- Subscript ([ ]) expressions, except when a subscript expression evaluates to a function or an array
- Member selection expressions (−> and .), if the selected member is one of the above expressions
- Unary indirection (*) expressions, except when such expressions refer to arrays or functions
- A **const** object, which cannot be changed
- An **lvalue** expression in parentheses.

IBM Extension

Type casts to pointer types, when the size of the object being cast does not change.

End of IBM Extension

Unary Increment and Decrement

The unary assignment operators (+ + and −−) increase and decrease their operand, respectively. The operand must have integral, floating-point, or pointer type and must be an **lvalue** expression.

Operands of integral or floating-point type are increased or decreased by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is increased or decreased by the size of the object it addresses. An increased pointer points to the next object; a decreased pointer points to the previous object.

An increment (+ +) or decrement (−−) operator can appear either before or after its operand. When the operator is a prefix to its operand, the result of the expression is the increased or decreased value of the operand. When the operator attaches to the end of its operand, the immediate result of the expression is the value of the operand before it increases or decreases. After C notes that result in context, it increases or decreases the operand.

Example

The following example compares the variable *pos* to 0, then increases it by 1. If *pos* is positive before being incremented, the next statement is run; then the contents of *q* are assigned to *p*, and both *p* and *q* are incremented.

```
if (pos++ > 0)
   *p++ = *q++;
```

The next example decreases the variable *i* before using it as a subscript to *line*.

```
if (line[--i] != '\n')
   return;
```

Simple Assignment Operator (=)

The simple assignment operator (=) assigns values to variables. The right operand is assigned to the left operand; the conversion rules for assignment apply.

Example

```
double x;
int y;

x = y;
```

C changes the value of *y* to **double** type and assigns it to *x*.

Compound Assignment Operators

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, and then assign the result to the left operand. A compound assignment expression such as:

```
expression1 += expression2
```

can be understood as:

```
expression1 = expression1 + expression2
```

However, the compound assignment expression is not equal to the expanded version because the compound assignment expression evaluates *expression1* only once, while in the expanded version, *expression1* is evaluated twice, once in the addition operation and once in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example

```
#define MASK 0x8000
n |= MASK;
```

This example performs a bitwise inclusive OR operation on *n*. The compiler assigns the result to *n*. The manifest constant MASK is defined with a **#define** preprocessor directive.

Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in an expression. The precedence of an operator is meaningful only in the presence of other operators having higher or lower precedence. Expressions involving higher precedence operators are evaluated first.

The following table summarizes the precedence and associativity of C operators. The operators are listed in order of precedence from the highest to the lowest. Where several operators appear together in a line or brace, they have equal precedence and are evaluated according to their associativity, either left to right or right to left.

Operator Symbol	Type of Operation	Associativity
() [] . – >	Expression	Left to right
– ~ ! * & ++ –– sizeof *casts*	Unary	Right to left
* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Shift	Left to right
< <= >= >	Relational (inequality)	Left to right
== !=	Relational (equality)	Left to right
&	Bitwise AND	Left to right
^	Bitwise Exclusive OR	Left to right

Operator Symbol	Type of Operation	Associativity
|	Bitwise Inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional	Right to left
= *= /= %= +=-= <<= >>= &= |= ^=	Simple and compound assignment	Right to left
,	Sequential evaluation	Left to right

As the preceding table shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member selection expression, or a parenthetical expression have highest precedence and associate left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either right to left or left to right. The result of expressions with many occurrences of multiplication, addition, or binary bitwise operators at the same level is unaffected by the direction of evaluation. The compiler can evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

Only the sequential evaluation operator and the logical AND and OR operators specify a particular order of evaluation for the operands. The sequential evaluation operator evaluates its operands from left to right.

The logical operators also evaluate their operands left to right. However, the logical operators evaluate the minimum number of operands necessary to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression *x* && *y*+ +, the second operand, *y*+ +, is evaluated only if *x* is true (nonzero). Thus, *y* is not increased when *x* is false (0).

Example
The examples below show the default grouping for several expressions:

In the first example, the bitwise AND operator has higher precedence than the logical OR operator so

```
a & b
```

forms the first operand of the logical OR operation.

```
Expression  Default Grouping
a & b || c       (a & b) || c
```

In the second example, the logical OR operator has higher precedence than the simple assignment operator, so

```
b || c
```

is grouped as the right-hand operand in the assignment. The value assigned to *a* is either 0 or 1.

```
Expression  Default Grouping

a = b || c       a = (b || c)
```

The third example shows a correctly formed expression that may produce an unexpected result. The logical AND operator has higher precedence than the logical OR operator, so

```
q && r
```

is grouped as an operand. Because the logical operators guarantee evaluation of operands from left to right,

```
q && r
```

is evaluated before *s*--. If *q* && *r* evaluates to a nonzero value, *s*-- is not evaluated, and *s* is not decreased. To correct this problem, *s*-- should appear as the first operand of the expression or should be decreased in a separate operation.

```
Expression  Default Grouping
q && r || s--       (q && r)|| s--
```

The following example shows an illegal expression that produces a program error: **Illegal Expression**

```
p == 0 ? p += 1 : p += 2
```

Default Grouping

```
(p == 0 ? p += 1 : p) += 2
```

In this example, the equality operator has the highest precedence, so *p* **= = 0** is grouped as an operand. The ternary operator (**? :**) has the next highest precedence. Its first operand is

```
p == 0
```

and its second operand is

```
p += 1.
```

However, the last operand of the ternary operator is considered to be *p* rather than

```
p+= 2,
```

because this occurrence of *p* binds more closely to the ternary operator than it does to the compound assignment operator. A syntax error occurs because the left-hand operand of:

```
+= 2
```

is not an **lvalue**.

To prevent errors of this kind and to produce more readable code, use parentheses. You can correct and clarify the above example by using parentheses, as shown here.

```
(p == 0) ? (p += 1) : (p += 2)
```

Side Effects

Side effects are changes in the state of the machine that take place as a result of evaluating an expression. Side effects occur whenever the value of a variable is changed. Any assignment operation has side effects, and any call to a function that contains assignment operations has side effects.

The order of evaluation of side effects is implementation-dependent, except where the compiler guarantees a particular order of evaluation.

For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2);
```

The arguments of a function call can be evaluated in any order. The expression

```
i + 1
```

may be evaluated before

```
i = j + 2,
```

or the compiler might interpret the second operand before the first, with different results in each case.

Unary increment and decrement operations involve assignment and can cause side effects, as shown in the following example:

```
d = 0;
a = b++ = c++ = d++;
```

The value of *a* is unpredictable. The initial value of *d* (0) could be assigned to *c*, then to *b*, and then to *a* before any of the variables are increased. In this case, *a* would be equal to 0.

A second method of evaluating this expression begins by evaluating the following operands:

```
c++ = d++
```

The initial value of *d* (0) is assigned to *c*, and then both *d* and *c* are increased. Next, the increased value of *c* (1) is assigned to *b* and *b* is increased. Finally, the increased value of *b* is assigned to *a*. In this case, the final value of *a* is 2.

Because the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be used. Statements that depend on a particular order of evaluation for side effects produce nonportable and unclear code.

Type Conversions

Type conversions are the assignment of a value to a variable of a different data type. Use type conversions when:

- A value is explicitly cast to another type.
- An operator converts the type of its operand or operands before performing an operation.
- A value is passed as an argument to a function.

The following sections outline the rules governing each kind of conversion.

Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integral and floating-point types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows.

From Signed Integer Types

C converts a signed integer to a shorter signed integer by truncating the high-order bits and converts to a longer signed integer by sign-extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a **long** value is converted to a **float.** To convert a signed integer to an unsigned integer, the signed integer is converted to the size of the unsigned integer and the result is interpreted as an unsigned value.

The following table summarizes conversions from signed integer types:

From	To	Method
char	**short**	Sign-extended.
	long	Sign-extended.
	unsigned char	Preserve pattern; high-order bit loses function as sign bit.
	unsigned short	Sign-extended to **short**; convert **short** to **unsigned short.**
	unsigned long	Sign-extended to **long**; convert **long** to **unsigned long.**
	float	Sign-extended to **long**; convert **long** to float.
	double	Signed-extended to **long**; convert **long** to **double.**
short	**char**	Preserve low-order byte.
	long	Sign-extend.
	unsigned char	Preserve low-order byte.

From	To	Method
	unsigned short	Preserve bit pattern; high-order bit loses function as sign bit.
	unsigned long	Sign-extend to **long**; convert **long** to **unsigned long**.
	float	Sign-extend to **long**; convert **long** to **unsigned long**.
	float	Sign-extend to **long**; convert **long** to **float**.
	double	Sign-extend to **long**; convert **long** to **double**.
long	**char**	Preserve low-order byte.
	short	Preserve low-order word.
	unsigned char	Preserve low-order byte.
	unsigned short	Preserve low-order word.
	unsigned long	Preserve bit pattern; high-order bit loses function as sign bit.
	float	Represent as **float**; if the **long** cannot be represented exactly, some loss of precision occurs.
	double	Represent as a **double**; if the **long** cannot be represented exactly as a **double**, some loss of precision occurs.

Note: In standard C, the **int** type is equal to either the **short** type or the **long** type, depending on the implementation. In IBM C/2, an **int** is equivalent to a **short**, and conversion of an **int** value proceeds as for a **short**.

From Unsigned Integer Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero-extending. Unsigned values are converted to floating-point values by first converting them to a signed integer of the same size, then converting that signed value to a floating-point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value changes if the sign bit is set.

The following table summarizes conversions from unsigned integer types:

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit.
	short	Zero-extend.
	long	Zero-extend
	unsigned short	Zero-extend.
	unsigned long	Zero-extend.
	float	Convert to **long**; convert **long** to **float**.
	double	Convert to **long**; convert **long** to **double**.
unsigned short	**char**	Preserve low-order byte.
	short	Preserve bit pattern; high-order bit becomes sign bit.
	long	Zero-extend.
	unsigned char	Preserve low-order byte.
	unsigned long	Zero-extend.
	float	Convert to **long**; convert **long** to **float**.
	double	Convert to **long**; convert **long** to **double**.

From	To	Method
unsigned long	**char**	Preserve low-order **byte**.
	short	Preserve low-order **word**.
	long	Preserve bit pattern; high-order bit becomes sign bit.
	unsigned char	Preserve low-order byte.
	unsigned short	Preserve low-order word.
	float	Convert to **long**; convert **long** to **float**.
	double	Convert directly to **double**.

Note: In standard C, the **unsigned int** type is equal to either the **unsigned short** type or the **unsigned long** type, depending on the implementation. In IBM C/2, an **unsigned int** is equivalent to an **unsigned short** and conversion of an unsigned **int** value proceeds as for an **unsigned short**.

From Floating-Point Types

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If C cannot exactly represent the **double** value as a **float**, the number loses precision but is rounded to the nearest value that can be represented. If the value is too large to fit into a **float**, the number is undefined.

A floating-point value is converted to an integer value by converting to a **long**. Conversions to other integer types take place as for a **long**. The decimal fraction portion of the floating-point value is discarded in the conversion to a **long**. If the result is still too large to fit into a **long**, the result of the conversion is undefined.

The following table summarizes conversions from floating-point types:

From	To	Method
float	**char**	Convert to **long**; **long** to **char**.
	short	Convert to **long**; convert **long** to **short**.
	long	Truncate at decimal point; if result is too large to be represented as a **long**, result is undefined.
	unsigned short	Convert to **long**; convert **long** to **unsigned short**.
	unsigned long	Convert to **long**; convert **long** to **unsigned long**.
	double	Change internal representation.
double	**char**	Convert to **float** to **char**.
	short	Convert to **float**; convert **float** to **short**.
	long	Truncate at decimal point; if result is too large to be represented as a **long**, result is undefined.
	unsigned short	Convert to **long**; convert **long** to **unsigned short**.
	unsigned long	Convert to **long**; convert **long** to **unsigned long**.
	float	Represent as a float; if the **double** value cannot be represented exactly as a **float**, loss of precision occurs; if the value is too large to be represented in a **float**, the result is undefined.

To and from Pointer Types

You can convert a pointer to one type of value into a pointer to a different type. The result might be undefined, however, because of the alignment requirements and sizes of different types in storage.

IBM Extension

In some implementations, the special keywords **near**, **far**, and **huge** can change the size of pointers within a program. You can convert a pointer to a pointer of a different size; the path of the conversion depends on the implementation. For example, on an 80286 processor, the compiler uses a segment-register value to convert a 16-bit pointer to a 32-bit pointer. See the "Pointer Conversion" section in Chapter 2 of *IBM C/2 Compile, Link, and Run* for information on pointer conversions.

End of IBM Extension

You can convert a pointer value to an integral value. The path of the conversion depends on the size of the pointer and the size of the integer type.

If the pointer is the same size or larger than the integer type, the pointer behaves like an unsigned value in the conversion, except that you cannot convert it to a floating-point value.

If the pointer is smaller than the integer type, C converts it to a pointer with the same size as the integer type. C then converts it to the integer type. The method of converting a pointer to a longer pointer depends on the implementation.

C can convert an integer type to a pointer type. If the integer type is the same size as the pointer type, the conversion causes C to treat the integer value as a pointer (an unsigned integer). If the size of the integer type is different from the size of the pointer type, C converts the integer type to the size of the pointer using the conversion paths given in the preceding charts. C then treats it as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, C can implicitly convert pointer values. In particular, the compiler can make assumptions about the default size of pointers and convert

passed pointer values accordingly, unless a forward declaration changes the implicit conversion.

From Other Types
An **enum** value is an **int** value, by definition of the **enum** type. Conversions to and from an **enum** value proceed as for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation. No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to **void** by assignment. However, a value can be explicitly cast to **void**, as discussed in the following section.

Type Cast Conversions

You can explicitly declare type conversions with a type cast. A type cast has the form:

(type-name) operand

where *type-name* specifies a particular type and *operand* is a value for conversion to the specified type.

The conversion of *operand* takes place as though you assigned it to a variable of the named type. The conversion rules for assignments apply to type casts as well. You can use the type name **void** in a cast operation, but you cannot assign the resulting expression to any item.

Example
The following expression contains the (double) x cast expression:

```
printf("x=%f\n," (double)x);
```

The function **printf** receives the value of x as a double. The cast does not change the value of the variable x.

The usage of pointer modifiers in a cast is illustrated next. A pointer to an object of one type may be cast to a pointer to an object of another type.

```
(long *) malloc (20*sizeof(long))
```

In this example the function **malloc** returns a pointer to **void**. The cast (long *) converts the returned value to a pointer to a **long**.

Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand and operands. Many operators perform the usual arithmetic conversions.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express storage positions. See "Subscript Expressions" on page 6-4 and "Additive Operators" on page 6-15 for details.

Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on whether a forward declaration with declared argument types is present for the called function.

If a forward declaration is present and it includes declared argument types, the compiler performs type-checking. For a description of the type-checking process, see "Arguments" on page 8-14.

If no forward declaration is present or if the forward declaration omits the argument type list, the only conversions performed on the arguments in the function call are the usual arithmetic conversions. C performs these conversions independently on each argument in the call. This means that a **float** value converts to a **double** or a **char** or **short** value converts to an **int**, and an **unsigned char** or **unsigned short** converts to an **unsigned int**.

If you use the special keywords **near**, **far**, and **huge**, the compiler makes implicit conversions on pointer values passed to functions. You can change these implicit conversions by providing argument type lists to let the compiler perform type checking. See the "Pointer Conversions" section in Chapter 2 of *IBM C/2 Compile, Link, and Run* for more information.

Chapter 7. Using C Statements

The statements of a C program control the flow of a program run. In C, several kinds of statements are available to perform loops, to select other statements to be run, and to transfer control. This chapter describes C statements in alphabetic order, as follows:

break	**do**	**goto**	**return**
compound	expression	**if**	**switch**
continue	**for**	null	**while**

C statements consist of keywords, expressions, and other statements. The keywords that appear in C statements are:

break	**default**	**for**	**return**
case	**do**	**goto**	**switch**
continue	**else**	**if**	**while**

Statements appearing within C statements can be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the body of the enclosing statement. Frequently the statement body is a compound statement, which is a single statement composed of one or more statements.

Braces begin and end a compound statement. All other C statements end with a semicolon.

You can prefix any C statement with an identifying label consisting of a name and a colon. Statement labels are recognized only by the **goto** statement.

When a C program runs, it runs the statements in the order of their appearance in the program, except where a statement explicitly transfers control to another location.

break

Purpose

The **break** statement ends the running of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. A **break** statement appearing outside any **do**, **for**, **switch**, or **while** statement causes an error.

Within nested statements, the **break** statement ends only the **do**, **for**, **switch**, or **while** statement immediately enclosing it. To transfer control out of the nested structure altogether, use a **return** or **goto** statement.

Format

```
break;
```

Example

The following example processes an array of variable length strings stored in *lines*. The **break** statement causes an exit from the inner **for** loop after the ending null character (\0) of each string is found and stored in lengths [i]. Control then returns to the outer **for** loop. The variable *i* is increased, and the process is repeated until *i* is greater than or equal to LENGTH - 1.

```
for (i = 0; i < LENGTH - 1; i++)
{
   for (j = 0; j < WIDTH - 1; j++)
   {
      if (lines[i][j] == '\0')
      {
  lengths[i] = j;
  break;
      }
   }
}
```

Compound Statement

Purpose

In a compound statement, the compiler runs the statements in the order they appear, except where a statement transfers control to another location.

Format

```
{
[declaration]

⋮
statement
[statement]

⋮
}
```

Example

```
if (i > 0)
{
    line[i] = x;
    x++;
    i--;
}
```

A compound statement can appear as the body of another statement, such as the **if** statement. In the above example, if *i* is greater than 0, all of the statements in the compound statement are run in order.

Labeled Statements: Any statement in a compound statement can carry a label. Therefore, transfer into the compound statement by means of a **goto** is possible. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Declarations in a compound statement precede the program statements, so transferring directly to a program statement within the compound statement bypasses the initializations. The results are unpredictable.

continue

Purpose

The **continue** statement passes control to the next **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. Within a **do** or a **while** statement, the next iteration begins with the reevaluation of the expression of the **do** or **while** statement. Within a **for** statement, the next iteration starts with the run of the loop-expression of the **for** statement. It proceeds with the evaluation of the conditional expression and subsequent end or repeat of the statement body.

Format

```
continue;
```

Example

In the following example, the compiler runs the statement body if *i* is greater than 0. First, it assigns *f(i)* to *x*. Then, if *x* is equal to 1, the compiler runs the **continue** statement. It ignores the rest of the statements in the body, and running resumes at the top of the loop with the evaluation of *i*-- >0.

```
while (i-- > 0)
{
   x = f(i);
   if (x == 1)
     continue;
   y = x * x;
}
```

Purpose

The compiler runs the body of a **do** statement one or more times until *expression* becomes false. First, it runs the statement body. Then, it evaluates *expression*. If *expression* is false (zero), the compiler ends the **do** statement and passes control to the next statement in the program. If *expression* is true (nonzero), the compiler runs the statement body again and tests *expression* again. The compiler runs the statement body repeatedly until *expression* becomes false.

The **do** statement can also end with the running of a **break**, **goto**, or **return** statement within the statement body.

Format

```
do
      statement
while (expression);
```

Example

In the following example, the compiler runs the two statements *y = f(x);* and *x--*; regardless of the initial value of *x*. Then, it evaluates *x > 0*. If *x* is greater than 0, the compiler runs the statement body again and reevaluates *x > 0*. The statement body is run repeatedly as long as *x* remains greater than 0. The compiler stops running the **do** statement when *x* becomes 0 or negative.

```
do
{
   y = f(x);
   x--;
} while (x > 0);
```

Expression Statement

Purpose

An expression is a sequence of operators and operands that specifies how to compute a value.

Format

expression;

Example

In C, assignments are expressions. The value of the expression is the value being assigned (sometimes called the right-hand value).

In the following example, *x* is assigned the value of *y* + *3*.

```
x = (y + 3);
```

In the next example, *x* is increased by 1.

```
x++;
```

The following example shows a function-call expression. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually includes an assignment to store the returned value when the function is called. If the return value is not assigned as in the example, the function-call is run, but the return value, if any, is not used.

```
f(x);
```

Purpose

The compiler runs the body of a **for** statement one or more times until the optional *cond-expression* becomes false. The *init-expression* and *loop-expression* are optional expressions that initialize and change values during the running of the **for** statement.

The first step in the running of the **for** statement is the evaluation of *init-expression*, if present. Next, *cond-expression* is evaluated with three possible results:

- If the conditional expression is true (nonzero), the compiler runs the *statement*. Then, it evaluates the *loop-expression*, if present. The process begins again with the evaluation of *cond-expression*.
- If the conditional expression is omitted, the conditional expression is considered true. The run proceeds exactly as described above. A **for** statement lacking *cond-expression* ends only upon the running of a **break**, **goto**, or **return** statement within the statement body.
- If the conditional expression is false, running of the *statement* ends and control passes to the next statement in the program.

A **for** statement can also end with the running of a **break**, **return**, or **goto** statement within the statement body.

Format

```
for ([init-expression];
     [cond-expression];
     [loop-expression]) statement;
```

for

Example

```
for (i = space = tab = 0; i < MAX; i++)
{
   if (line[i] == ' ')
       space++;
   if (line[i] == '\t')
   {
      tab++;
      line[i] = ' ';
   }
}
```

The example counts blank and tab (\t) characters in the array of characters named *line* and replaces each tab character with a blank. First, the compiler initializes *i*, *space*, and *tab* to 0. Then, it compares *i* to the constant MAX. If *i* is less than MAX, the compiler runs the statement body. Depending on the value of *line* [*i*], it runs the body of one or neither of the **if** statements. Then, it increases and tests *i* against MAX. The compiler runs as long as *i* is less than MAX.

Purpose

The **goto** statement transfers control directly to the statement specified by *name*. The compiler runs the labeled statement immediately after it runs the **goto** statement. An error results if no statement with the given label resides in the same function or if more than one statement in the same function has identical labels.

A statement label is meaningful only to a **goto** statement. When a labeled statement is encountered in any other context, the statement is run without regard to the label.

Format

```
goto name;
   .
   .
   .
name: statement
```

Example

```
if (errorcode > 0)
     goto exit;
 .
 .
 .
exit:
     return (errorcode);
```

In the example, a **goto** statement transfers control to the point labeled *exit* when an error occurs.

A label name is an identifier, formed by following the same rules that govern the construction of identifiers. Each statement label must be distinct from other statement labels in the same function.

if

Purpose

The compiler runs the body of an **if** statement selectively, depending on the value of *expression*. First, it evaluates *expression*. If *expression* is true (nonzero), the compiler runs the statement immediately following it. If *expression* is false, it runs the statement following the **else** keyword. If *expression* is false and the **else** clause is omitted, the compiler ignores the statement following *expression*. Control then passes from the **if** statement to the next statement in the program.

Format

```
if (expression)
        statement1
[ else
        statement2 ]
```

Example

In the following example, the statement **y = x/i;** is run if *i* is greater than 0. If *i* is less than or equal to 0, *i* is assigned to *x* and *f(x)* is assigned to *y*. The statement forming the **if** clause ends with a semicolon.

```
if (i > 0)
   y = x/i;
else
{
   x = i;
   y = f(x);
}
```

C does not offer an **else-if** statement. The same effect is achieved by nesting **if** statements. You can nest an **if** statement in either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. In the absence of braces, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Example

In the following example, the **else** is associated with the inner **If** statement. If *i* is less than or equal to 0, no value is assigned to *x*.

```
if (i > 0)     /* Without braces */
   if (j > i)
      x = j;
   else
      x = i;
```

Changing the indentation has no effect on the operation of this else clause. The following is equivalent to the first example:

```
if (i>0)   /*  Without braces  */
  if(j>i)
      x=j;
else        /* Faulty indentation  */
  x=i;
```

In the next version, the braces surrounding the inner **If** statement make the **else** clause part of the outer **If** statement. If *i* is less than or equal to 0, *i* is assigned to *x*.

```
if (i > 0)
{         /* With braces */
   if (j > i)
      x = j;
}
else
   x = i;
```

Purpose

The **return** statement ends the running of the function in which it appears and returns control to the calling function. Running resumes in the calling function at the point just after the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. In this case, the function should have been given type **void**.

Format

```
return [expression];
```

Example

```
main()
{
  .
  .
  .
   y = sq(x);
   draw(x, y);
  .
  .
  .
}

sq(x)
int x;
{
   return (x * x);
}

void draw(x,y)
int x, y;
{
  .
  .
  .
   return;
}
```

Purpose

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose case *constant-expression* matches the value of the *expression* in parentheses. Running of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The default statement is run if no case *constant-expression* is equal to the value of the switch *expression*. If the default statement is omitted and no case match is found, none of the statements in the switch body are run.

The switch *expression* is an integer value that must be the size of an **int** or shorter. It can also be an enumerated value. If the *expression* is shorter than an **int**, the compiler widens it to an **int** value. The compiler then casts each case *constant-expression* to the type of the switch *expression*. The value of each case *constant-expression* must be unique within the statement body.

The case and default labels of the **switch** statement body are significant only in the initial test that determines the starting point for running of the statement body. All statements appearing between the statement at which running starts and the end of the body run regardless of their labels, unless a statement transfers control out of the body entirely.

Declarations can appear at the head of the compound statement forming the switch body, but the compiler does not perform initializations included in the declarations. The effect of the **switch** statement is to transfer control directly to a program statement within the body, bypassing the lines that contain initializations.

statements. Similarly, if *i* is equal to 0, only *z* is increased. If *i* is equal to 1, only *p* is increased. The final **break** statement is not necessary because control passes out of the body at the end of the compound statement. The final break is included for consistency.

```
switch (i)
{
   case -1:
    n++;
    break;
   case 0 :
    z++;
    break;
   case 1 :
    p++;
    break;
}
```

A statement can carry multiple case labels as the following sample shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although you can label any statement within the body of the **switch** statement, no statement must be labeled. You can intermingle statements without labels and statements with labels. Keep in mind, however, that when the **switch** statement passes control to a statement within the body, all succeeding statements in the block are run, regardless of their labels.

Chapter 8. Using IBM C/2 Functions

A *function* is an independent collection of declarations and statements usually designed to perform a specific task. C programs have at least one function (which must be named **main**) and can have other functions. This chapter describes how to define, declare, and call C functions.

A *function definition* specifies the name of the function, the return type, its storage class, its formal parameters, and the declarations and statements that define its action.

A *function declaration* establishes the name, return type, storage class, and number and types of formal parameters of a function whose explicit definition is at another point in the program. This lets the compiler compare the types of the arguments and the types of the formal parameters of a function. Function declarations are optional for functions whose return type is **int**. To ensure the correct matching or conversion of data types, you must declare, before the program calls them, functions that have return types other than **int**.

A function call passes control from the calling function to the called function. The function call also passes the values for the arguments, if any, to the called function. Performing a return statement in the called function returns control to the calling function. It also may return a value from the function.

Function Definitions

A function definition specifies the name, return type, storage class, formal parameters, and body of a function.

Format

A function definition has the following form:

```
[sC specifier][type-specifier]declarator([parameter-list])
[parameter-declarations]
function-body
```

The *sC specifier* gives the storage class of the function, which must be either **static** or **extern**.

Return Type

The *return type* of a function defines the size and type of value that the function returns. The type declaration has the form:

[*type-specifier*] *declarator*

where the *type-specifier*, together with the *declarator*, defines the return type and name of the function. If you do not specify a *type-specifier*, the IBM C/2 compiler assumes that the return type is **int**.

The *type-specifier* can specify any fundamental, structure, or union type. The *declarator* consists of the function identifier, possibly modified to declare a pointer type. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. You need not declare functions with an **int** return type before you call them. But you must define or declare functions with other return types before you call them.

The IBM C/2 compiler uses the type of a return value of a function only when the function returns a value. A function returns a value when the program performs a **return** statement containing an expression. The program evaluates the expression, converts the expression to the type of the return value, if necessary, and returns the value to the point of call. If the program does not perform a **return** statement, or if the **return** statement performed does not contain an expression, the return value of the function remains undefined. If the calling function expects a return value, the behavior of your program is also undefined.

Example

In the following example, the return type of *add* is **int** by default. The function has a storage class of **static**. You can call it only with functions in the same source file.

```
 /* Return type is int */
static add (x, y)
int x, y;
{
   return (x+y);
}
```

The parameter list is enclosed in parentheses and can take either of two forms:

```
[ identifier[,identifier] ...]
```

where each *identifier* names a parameter. You must include the parentheses.

The preferred parameter list form is:

```
[sC-specifier] type-specifier declarator[,...]
```

Parameter declarations define the type and size of values stored in the formal parameters. These declarations have the same form as other variable declarations. A formal parameter can have any fundamental, structure, union, pointer, or array type.

A parameter can be in only the **auto** or **register** storage class. If you do not specify a storage class, the IBM C/2 compiler assumes that the storage class is **auto**. If you name a formal parameter in the parameter list but do not declare it, the IBM C/2 compiler assumes that the parameter is of the **int** type. You can declare formal parameters in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. You cannot use these identifiers for variable declarations within the function body.

Make sure that the type of the formal parameter corresponds to the type of the argument and to the type of the corresponding argument in the argument type list for the function, if present. If the function has a variable number of arguments, you must determine the number of arguments passed and have the program retrieve additional arguments from the stack within the body of the function.

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary.

warning rather than an error, the resulting code will not work correctly because ANSI C considers old- and new-style function definitions to be different. (Where *old-style* means before the Proposed ANSI Standard for C, and *new-style* means conforming to the Proposed ANSI Standard for C.) In particular, old-style definitions, such as the definition in the preceding code fragment, are forced to widen type **float** arguments to type **double**. Thus, the fragment generates the parameter-mismatch error because the prototype specifies a type **float** argument and the definition specifies a type **double** argument. However a new-style definition, such as

```
void takesfloat(float f)
{
⋮
}
```

accepts the narrower argument type.

If you have code of this kind that causes parameter-mismatch errors, you have two possible solutions, depending on whether the desired argument type is actually **float** or **double**:

1. If you want type **double** arguments, then change argument types in the prototype to **double**.
2. If you want type **float** arguments, change the definition to use the new-style format illustrated above.

Example
The following example shows the use of parameter lists in functions.

The array *name* [20] given as the second argument in the call evaluates to a **char** pointer. The example declares the corresponding formal parameter to be a **char** pointer and uses it in subscripted expressions as though it were an array identifier. Because an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as **char** **n* is the same as declaring it **char** *n*[].

Within the function, the local variable *i* is a definite value and keeps track of the current position in the array. The function returns the *id* structure member if the *name* member matches the array *n*. Otherwise, it returns zero.

Function Body

The function body is a compound statement. The compound statement contains the statements that define the action of the function. It can also contain declarations of variables that these statements use.

All variables declared in the function body have **auto** storage type unless otherwise specified. When a program calls the function, the function automatically creates storage space for the local variables and gives them their initial values. Control passes to the first statement in the compound statement and continues sequentially until the program meets a **return** statement or the end of the function body. Control then returns to the point of call.

You must code an expression in the **return** statement if the function is to return a value. The return value of a function remains undefined if no **return** statement occurs or if the **return** statement does not include an expression.

Function Declarations

A function declaration defines the name, return type, and storage class of a given function, as well as the type of its arguments. Such a declaration is known as a function prototype. You can declare functions implicitly or with forward declarations. The return type of a function declared either implicitly or with a forward declaration must agree with the return type specified in the function definition.

An implicit declaration occurs whenever you call a function without previously defining or declaring it. When you make an implicit declaration, the compiler constructs a default prototype for the function. In

Example

In this example, the function **intadd** is implicitly declared to return an **int** value, because it is called before it is defined. The compiler does not check the types of the arguments in the call because no argument-type list is available.

```
main()
{
   int a = 0, b = 1;
   double x = 2.0, y = 3.0;
   double realadd(double x, double y);

   a = intadd (a, b);
   x = realadd(x, y);
}

intadd(a, b)
int a, b;
{
   return (a + b);
}

double realadd(x, y)
double x, y;
{
   return (x + y);
}
```

The function **realadd** returns a **double** value instead of an **int.** The forward declaration of **realadd** in the **main** function lets the program call the **realadd** function before it defines **realadd.** The definition of **realadd** matches the forward declaration by specifying the **double** return type.

The forward declaration of **realadd** also establishes the type of its two arguments. The arguments match the types given in the forward declaration and also match the types of the formal parameters.

Function Calls

A function call is an expression that passes control and zero or more arguments to a function. A function call has the form:

expression (*expression-list*)

where the *expression* evaluates to a function address, and the *expression-list* is a list of expressions whose values, the arguments, are passed to the function. The *expression-list* can be empty.

Example

This example assigns the return value, a pointer to a **double**, to *rp*.

```
double *realcomp(double, double);
double a, b, *rp;
      .
      .
      .
rp = realcomp(a, b);
```

In the next example, the function call, **work (count, lift);** in **main** passes an integer variable and the address of the function **lift** to the function **work**.

```
   long lift(int), step(int), drop(int);
   void work (int, long (*)(int));
main ()
{

   int select, count;
     .
     .
     .
   select = 1;
   switch ( select )
   {
      case 1: work(count, lift);
       break;

      case 2: work(count, step);
       break;

      case 3: work(count, drop);
       break;

      default:
       break;
   }
}

void work ( n, func )
int n;
long (*func)(int);
{
   int i;
   long j;

   for (i = j = 0; i < n; i++)
    j += (*func)(i);
}
```

This example passes the function address with the function identifier because a function identifier evaluates to a pointer expression. To use a function identifier in this way, you must declare the function or

does not perform a conversion but issues warning messages as if it had assigned the expressions to the formal parameters.

If you use the **near**, **far**, and **huge** keywords, you can also perform conversions on pointer arguments.

The number of expressions given in the expression list must match the number of parameters of the function, unless the forward declaration of the function explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the argument type list and converts them, if necessary, as described above. If there are additional arguments in the function call, each additional argument undergoes the usual arithmetic conversions, but the compiler does not otherwise convert it or check it.

If the parameter-type list contains the special type name **void**, the compiler produces a warning message if it also contains arguments.

If the argument-type list is empty (omitted) or the called function has no forward declaration, the compiler performs no type-checking, either for type or for number of arguments. In this case, the arguments in the function call, if any, undergo the usual arithmetic conversions independently before the compiler places them on the stack.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the compiler interprets the arguments on the stack. If the type of the formal parameter does not match the type of the argument, the compiler can misinterpret the data on the stack.

Note: Type mismatches between arguments and formal parameters can produce serious errors, particularly when the mismatches entail size differences. The compiler does not detect these errors unless you provide an argument type list in the forward declaration of the function.

Example

The following example declares that the **swap** function in **main** has two arguments, both pointers to integers. The example declares the formal parameters *a* and *b* as pointers to integer variables. In the function call:

```
swap (&x, &y)
```

```
int func1(int a,...);
...
int func1(int b, char *b, int c)
{
...
}
```

The compiler places all arguments in the function call on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the parameters of the function. You must retrieve any additional arguments from the stack and determine how many arguments are present. See *IBM C/2 Language Reference* for information about macros that you can use to handle a variable number of arguments in a portable way.

Example

This example shows a function named **scores** that takes a variable number of arguments. The forward declaration of **scores** in **main** establishes that **scores** has at least one argument, an **int**. The comma at the end of the argument-type list means that there can be more undeclared arguments.

```
   int scores (int,...);
main()
{
   int count, average, i;

⋮
   average = scores (count, 14, 96, 82);

⋮
}

scores (number)
int number;
{
   int *ip, total = 0, i;
   ip = &number + 1;

   for (i = 1; i <= number; i++, ip++)
    total += *ip;

   if (number > 0)
    return (total/number);
   return (-1);
}
```

The function call to **scores** passes four arguments. The compiler checks the first argument for compatibility with the argument-type list

Chapter 9. Using Preprocessor Directives and Pragmas

The *C preprocessor* is a text processor that manipulates the text of a source file before compiling. The compiler ordinarily calls the preprocessor in its first pass; however, you can call the preprocessor at any time to process text before compiling. This chapter explains the main tasks that the preprocessor performs in response to preprocessor directives and describes each directive in detail.

Preprocessor directives make source programs easy to change and to compile for different systems. Directives in the source file instruct the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, and suppress the compiling of a portion of the file by removing blocks of text.

The C preprocessor recognizes the following directives:

#define	**#endif**	**#ifdef**	**#line**
#elif	**#error**	**#ifndef**	**#pragma**
#else	**#if**	**#include**	**#undef**

The # sign must be the first non-white-space character on the line containing a directive. White space or blank characters can appear between the number sign and the first letter of the directive. Some directives are followed by arguments or values, as described in the following text. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A *pragma* is a pragmatic (practical) instruction to the IBM C/2 compiler. You imbed pragmas in C source files. Pragmas control the actions of the compiler in a particular portion of a program without affecting the entire program. Each implementation defines the particular pragmas available and their meaning. See *IBM C/2 Compile, Link, and Run* for information about the use of pragmas and their effects on compiling.

#define
Define Directive

Purpose
The **#define** directive substitutes a text string for an identifier.

Format

```
#define identifier text
#define identifier(parameter-list) text
```

Comments:
The **#define** directive substitutes the given *text* for subsequent occurrences of the specified *identifier* in the source file. The preprocessor replaces the *identifier* only when it forms a token. For instance, the preprocessor does not replace an *identifier* when it occurs within strings or as part of a longer identifier. Defined identifiers are conventionally given in all-capital letters, although this is not required on the **#define** directive.

If a *parameter-list* appears after the *identifier*, the **#define** directive replaces each occurrence of *identifier (argument-list)* with a version of *text* modified by substituting the arguments for the formal parameters. There must be no character between the identifier and the left parenthesis.

The *parameter-list*, when given, consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and you must enclose the list in parentheses.

The *text* consists of a series of tokens (such as keywords), constants, or complete statements. One or more white space characters must separate the *text* from the *identifier* (or from the closing parenthesis of the *parameter-list)*. If the text is longer than one line, you can continue it onto the next line by preceding the new-line character with a backslash.

The *text* can also be empty. The effect of this option is to remove instances of the given *identifier* from the source file. However, the preprocessor still considers the *identifier* to be defined; it yields the value 1 when you test the identifier with the **defined** operator of the **#if** directive (discussed later in this chapter).

REG3. It defines REG1 and REG2 as the keyword **register**. The definition of REG3 is empty; the preprocessor removes each occurrence of REG3 from the source file. These directives ensure that the most important variables (declared with REG1 and REG2) receive **register** storage.

```
#define REG1        register
#define REG2        register
#define REG3
```

The next example defines a macro named MAX. The preprocessor replaces each occurrence of the identifier MAX following the definition in the source file with the expression **((x) > (y)) ? (x) : (y)**, where actual values replace the parameters *x* and *y*. The occurrence: **MAX(1,2)** is replaced with **((1) > (2)) ? (1) : (2)**.

The occurrence **MAX(i,s[i])** is replaced with **((i) > (s[i])) ? (i) : (s[i])**.

This macro is easier to read than the corresponding expression, making the source program easier to understand.

Arguments with side effects can cause this macro to produce unexpected results. The occurrence **MAX(i, s[i++])** is replaced with **((i) > (s[i++])) ? (i) : (s[i++])**.

The expression **(s[i++])** might be evaluated twice. If **i<=s[i++]**, then **i** will have been increased by 2 at the end of the evaluation of the expression.

```
#define MAX(x,y)    ((x) > (y)) ? (x) : (y)
```

The next example defines the macro MULT. After you define this macro, an occurrence such as **MULT(3, 5)** is replaced by **(3) * (5)**. The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro. For instance, the occurrence **MULT(3 + 4, 5 + 6)** is replaced by **(3 + 4) * (5 + 6)**, which evaluates to 77. Without the parentheses, the result is **3 + 4 * 5 + 6**, which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

```
#define MULT(a,b)   ((a) * (b))
```

As the preprocessor progresses through the source file, the references to **show** are expanded as follows:

```
show ( x + z ) ;
```

produces

```
printf("x + z") ;

show (n /* comment */ + p) ;
```

produces

```
printf ("n + p") ;

show (GREETING) ;
```

produces

```
printf ("GREETING") ;

show ('\x') ;
```

produces

```
printf (" '\\x' ") ;
```

When the program runs, the following is displayed:

```
x + z
n + p
GREETING
'\x'
```

Comments:
The extension to the ANSI C standard that previously enabled expansion of macro formal arguments appearing in string literals and character constants is no longer supported. You must rewrite the code that relied on this extension using the stringizing operator.

#undef
Undefine Directive

Purpose
The **#undef** directive removes the current definition of the *identifier*.

Format

```
#undef  identifier
```

Comments:

The **#undef** directive, paired with a **#define** directive, creates a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive to control compilation of portions of the source program. The preprocessor will no longer expand occurrences of *identifier*. To remove a macro definition using **#undef**, give only the macro *identifier*. Do not give a parameter list, even if the macro is defined with one.

Example
In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Only the identifier of the macro is given. The **#undef** directive can also be applied to an identifier that has no previous definition. This ensures that the identifier is undefined.

```
#define WIDTH  80
#define ADD(X,Y) (X) + (Y)
  .
  .
  .
#undef WIDTH
#undef ADD
```

If the file specification does not give a complete pathname and you enclose the file specification in double quotation marks, the preprocessor searches for the file in the same directory that the source file including this file resides in. It then searches directories specified in the compiler command prompt and finally searches in a set of standard directories. The preprocessor stops searching as soon as it finds a file with the given name.

If you enclose the file specification in less-than, greater-than symbols, the preprocessor does not search the current working directory. It begins by searching for the file in directories specified in the compiler command line and then searches the standard directories.

Example
The first example adds the contents of the file named stdio.h to the source program. The less-than, greater-than symbols cause the preprocessor to search the standard directories for stdio.h, after searching directories specified in the command line.

```
#include <stdio.h>
```

The next example adds the contents of the file specified by defs.h to the source program. The double quotation marks mean that the preprocessor is to search the directory containing the current source file first.

```
#include "defs.h"
```

Conditional Compiling

This section describes the syntax and use of directives that control conditional compiling. These directives allow for suppressing the compiling of portions of a source file. They test a constant expression or an identifier to determine which text blocks the preprocessor passes on to the compiler and which it removes from the source file during preprocessing.

#if, #elif, #else, #endif
If, Else-if, Else, and End-if Directives

Purpose
The **#if** directive, together with the **#elif**, **#else**, and **#endif** directives, controls the compiling of portions of a source file.

Format

```
#if restricted-constant-expression
[text]
[#elif restricted-constant-expression
text]
[#elif restricted-constant-expression
text]

⋮
[#else
text]
#endif
```

Comments:
Each **#if** directive in a source file must match a closing **#endif** directive. Zero or more **#elif** directives can appear between the **#if** and **#endif** directives, but you can have no more than one **#else** directive. The **#else** directive, if present, must be the last conditional directive before **#endif**.

The combination **#if defined**(*identifier*) supplants the **#ifdef** and **#ifndef** directives.

Use the **#elif** (else – if) directive in **#if** and **#if defined** blocks.

The preprocessor selects one of the given blocks of text for further processing. A text block is any sequence of text. It can occupy more than one line.

The preprocessor processes the selected text and passes it to the compiler. If the text contains preprocessor directives, the preprocessor carries out those directives.

The preprocessor removes from the file any text blocks not selected with **#if** directives and does not compile them.

```
#if defined(CREDIT)
 credit( );
#elif defined(DEBIT)
 debit( );
#else
 printerror( );
#endif
```

The next two examples assume a previously defined manifest constant, DLEVEL. The second example shows two sets of nested **#if**, **#else**, and **#endif** directives. The preprocessor processes the first set of directives only if DLEVEL>5 is true. Otherwise, the preprocessor processes the second set of instructions.

```
#if DLEVEL > 5
   #define SIGNAL  1
   #if STACKUSE == 1
      #define STACK  200
   #else
      #define STACK  100
   #endif
#else
   #define SIGNAL  0
   #if STACKUSE == 1
      #define STACK  100
   #else
      #define STACK  50
   #endif
#endif
```

In the next example, **#elif** and **#else** directives make one of four choices, based on the value of DLEVEL. The manifest constant STACK is set to 0, 100, or 200, depending on the definition of DLEVEL. If DLEVEL is undefined, the preprocessor sends **display(debugptr);** to the compiler and does not define STACK.

```
#if DLEVEL == 0
   #define STACK  0
#elif DLEVEL == 1
   #define STACK  100
#elif DLEVEL > 5
   display(debugptr);
#else
   #define STACK  200
#endif
```

The following example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns **register** storage to variables in the same order in which

#ifdef, #ifndef
Ifdef and Ifndef Directives

Purpose

The **#Ifdef** and **#Ifndef** directives accomplish the same task as the **#If** directive used with **defined(***macro***)**. You can use these directives anywhere you can use **#If**. IBM provides these directives only for compatibility with previous versions of C. For optimal results, use the **defined(***macro***)** form of **#If**. You cannot define or undefine the identifier **defined**.

When the preprocessor meets an **#Ifdef** directive, it checks to see if the *identifier* is currently defined. If so, the condition is true (nonzero). Otherwise, the condition is false (zero). The **#Ifndef** directive checks for the opposite condition checked by **#Ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (zero).

Format

```
#ifdef  identifier
#ifndef  identifier
```

Example

In the first example, the internally stored line number is set to 151 and the filename is changed to COPY.C.

```
#line 151 "copy.c"
```

In the next example, the macro ASSERT uses the predefined identifiers __LINE__ and __FILE__ to print an error message about the source file if a given assertion is not true.

```
#define ASSERT(cond)        if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__);} else ;
```

Appendix A. Differences from the Proposed ANSI Standard for C

Although the IBM C/2 product follows most of the language design as specified in the proposed draft *American National Standard for C*, a number of differences exist.

The compiler uses a number of identifiers which in a standard implementation are reserved for the user's namespace.

The following IBM C/2 functions are non-ANSI functions:

access
alloca
_arc
bdos
_bios_disk
_bios_equiplist
_bios_keybrd
_bios_memsize
_bios_printer
_bios_serialcom
_bios_timeofday
cabs
cgets
_chain_intr
chdir
chmod
chsize
_clear87
_clearscreen
close
_control87
cprintf
cputs
creat
cscanf
dieeetomsbin
_disable
_displaycursor
dmsbintoieee
dosexterr
_dos_allocmem
_dos_close
_dos_creatnew
_dos_findfirst
_dos_findnext
_dos_freemem
_dos_getdate
_dos_getdiskfree
_dos_getfileattr
_dos_getftime
_dos_gettime
_dos_getvect
_dos_keep
_dos_open
_dos_read
_dos_setblock
_dos_setdate
_dos_setdrive
_dos_setfileattr
_dos_setftime
_dos_settime
_dos_setvect
_dos_write
dup
dup2
ecvt
_ellipse
_enable
eof
execl
execle
execlp
execlpe
execv
execve
execvp
execvpe
_exit
_expand
fcloseall
fcvt
fdopen
_ffree
fgetchar
_fheapchk
_fheapset
_fheapwalk
fieeetomsbin
filelength
fileno
_floodfill
flushall
_fmalloc
fmsbintoieee
_fmsize
FP_OFF
_fpreset
FP_SEG
fputchar
_freect
fstat
ftime
gcvt
_getbkcolor
getch
getche
_getcurrentposition
getcwd
_getfillmask
_getimage
_getlinestyle
_getlogcoord
_getphyscoord
getpid
_getpixel
_gettextcolor
_gettextposition
_getvideoconfig
getw
halloc
_harderr
_hardresume
_hardretn

The following other non-ANSI identifiers exist in the IBM C/2 run-time system:

Functions	Absolutes	Variables	Constants
$i8_implicit_exp	FIARQQ	daylight	HUGE
$i8_inpbas	FICRQQ	edata	
$i8_input	FIDRQQ	end	
$i8_input_ws	FIERQQ	environ	
$i8_output	FISRQQ	sys_errlist	
$i8_tpwr10	FIWRQQ	sys_nerr	
brk	FJARQQ	timezone	
brkctl	FJCRQQ	tzname	
	FJSRQQ	complex	
	STKHQQ	exception	
		stdaux	
		stdprn	

Note that the function names that begin with an underscore or a dollar sign in the previous lists do not violate the ANSI namespace, even though they are not ANSI functions.

Appendix B. Compiler, Linker and Run-Time Limits

The tables in this appendix contain information about the limits imposed by IBM C/2.

Compiler Limits

To operate IBM C/2, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

The following table summarizes the limits imposed by C/2. If your program exceeds one of these limits, an error message informs you of the problem.

Program Item	Description	Limit
String Literals	Maximum length of a string, including the ending null character (\0).	512 bytes
Constants	Maximum size of a constant. The type determines the maximum size of a constant. See "Constants" on page 3-6 for a discussion of constants.	
Identifiers	Maximum length of an identifier.	31 bytes (discards additional characters)
Declarations	Maximum level of nesting for structure/union definitions.	10 levels
Preprocessor Directives	Maximum size of a macro definition before expansion.	2043 bytes
	Maximum size of a macro definition after expansion.	1019 bytes

Item	Description	Limit
Groups	Maximum number of groups that can be defined.	21; but the linker always defines DGROUP so the effective maximum is 20
Overlays	Maximum number of overlays that can be defined.	63
Logical Segments	Maximum number of logical segments that can be defined.	128 by default; however, can be set as high as 3072 with /SEGMENTS
Libraries	Maximum number that can be searched.	32
Physical Segments per module	Maximum number of physical segments that can be allocated.	255 per module
Stack	Maximum stack size.	64KB

Glossary

This glossary includes terms and definitions from:

- The *American National Dictionary for Information Processing Systems*, copyright 1982 by the Computer and Business Manufacturers Association (CBEMA). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Definitions are identified by the symbol (A) after the definition.
- The *ISO Vocabulary—Information Processing* and the *ISO Vocabulary—Office Machines*, developed by the International Organization for Standardization, Technical Committee 97, Subcommittee 1. Definitions of published sections of the vocabularies are identified by the symbol (I) after the definition; definitions from draft international standards, draft proposals, and working papers in development by the ISO/TC97/SC1 vocabulary subcommittee are identified by the symbol (T) after the definition, indicating final agreement has not yet been reached among participating members.

abort. To end unexpectedly.

access. (1) The manner in which files or data sets are referred to by the computer. (2) To get access to.

additive operators. The operators perform addition (+) and subtraction (–).

address. * A character or group of characters that identifies a register, a particular part of storage, or some other data source or destination.

allocate. To assign a resource.

append. In word processing, to attach a file to the end of another file. Contrasts with link.

argc. A parameter that holds the total number of arguments passed to the **main** function.

argument. Any value with a fundamental, structure, union, or pointer type that a function call passes to the formal parameters of a function or a return call returns to the program.

argv. An array of pointers such that each element points to a string representation of an argument passed to the **main** function.

arithmetic negation. An operation changing the sign of a number.

array. A series of values of the same type; its elements reside in contiguous storage locations.

assembly mode. Display mode in which CodeView displays the program as assembler language instructions.

intermediate language, assembly language, or computer language.

compound assignment operators. Perform the operation specified by the additional operator, then assign the result to the left operand.

concatenate. To link.

concatenation. The operation that joins two strings in the order specified, thus forming one string whose length is equal to the sum of the lengths of the two strings.

configuration. The arrangement of a computer system or network as defined by the nature, number, and the chief characteristics of its functional units. The term may refer to a hardware configuration or a software configuration.

configure. To describe the devices, optional features, and programs installed on the system.

constant. A number, character, or string of characters that you can use as a value in a program.

constant expression. An operand that yields an unchanging value.

continuation character. In the C language, the backslash (\) and Enter keys cancel the effects of the newline escape sequence.

C preprocessor. A text processor that manipulates the text of a source file before compiling.

C source file. A text file that contains all or part of a C language source program.

C source program. A collection of one or more directives, declarations, and definitions.

data segment. An area of memory reserved for a program's data.

deallocate. To free.

declaration. The C language source code line that establishes the names and characteristics of the functions, variables, and types used in the program.

decrement. The quantity by which a variable is decreased.

declarator. An identifier that you can modify with brackets, asterisks, or parentheses to declare a pointer, array, or function data type.

definition. A declaration that also defines variables and functions.

denormal. For the numeric coprocessor, a special form of floating-point number, produced when an underflow occurs. A denormal is a number with a biased exponent that is zero. By providing a significand with leading zeros, the range of possible negative exponents can be extended by the number of bits in the significand. Each leading zero is a bit of lost accuracy, so the extended exponent range is obtained by reducing significance.

derived type. The type of an item whose declarator consists of identifiers changed by the addition of a preceding asterisk, trailing brackets, or trailing parentheses,

program in order to terminate the execution of that portion.

expanded. Of a macro, replaced by its definition.

expression. A combination of operands and operators that yields a single value.

expression evaluator. Allows the use of a specific programming language's syntax when entering expressions and addresses as arguments with CodeView commands. CodeView provides two expression evaluators, one using C syntax and one using BASIC syntax.

external level. Of declarations, outside of all function definitions.

fatal. Unrecoverable.

file handle. An integer value that the operating system uses to refer to the file.

floating-point constant. A decimal number representing a real, signed number.

formal parameters. Variables that receive values passed to a function by a function call.

forward declaration. Declaring a function without defining it.

flush. Referring to a buffer, writing the contents to a final location only after the buffer is full.

free. (1) Release. (2) Not in use.

function. An independent collection of declarations and statements, usually designed to perform a specific task.

function body. A compound statement containing local variable declarations and statements.

function call. An expression that passes control and zero or more arguments to a function.

function declaration. The C language source code line that establishes the name, the return type, and the storage class of a function whose explicit definition is at another point in the program.

function definition. A block of C language source code that specifies the name of the function, its formal parameters, and the declarations and statements that determine its action.

function prototype. A function declaration that also declares the types of its parameters.

generate. To produce a computer program by selecting subsets from skeletal code under the control of parameters. Also, to produce assembler language statements from the model statements of a macro definition when the definition is called by a macro instruction.

global lifetime. The property of having storage and a defined value throughout the program.

globally visible symbol. A program item (such as an identifier) coded in a context that makes it recognizable as a unique name throughout the program.

without regard to any specific implementation.

line-buffer. To flush a buffer when the compiler finds a newline character (**\n**) instead of when the buffer is full.

local lifetime. The property of having new storage reserved each time the compiler enters the block that defines or declares it.

logical AND (&&). The logical AND operator (&&) produces the value 1 if both operands have non-zero values.

logical operators. The operators logical AND (&&) and OR (||).

logical OR (||). This operator (||) performs an inclusive OR on its operands.

long integer constants. Integer constants that are 4 bytes in length.

lvalue expressions. Expressions that refer to storage locations.

macro. An identifier that represents statements or expressions.

main. The conventional name of the primary program function. Many operating systems require this name for the primary function.

manifest constant. An identifier that represents a constant.

map file. A listing file you can create during the LINK step that contains a list of segments within the load module.

members. Individual elements of a structure, union, set, or list.

memory. Program addressable storage.

memory allocation. The reserving, freeing, or reallocating of blocks of storage.

memory location. Any addressable point in storage.

mixed language programming. Making use of libraries of subroutines which may be written in different programming languages. C programs can link to these libraries and call their subroutines.

mixed mode. In sequential mode, the display mode in which CodeView displays source lines mixed with unassembled instructions. One source line for each corresponding group of assembler language instructions is displayed.

multidimensional array. An array whose elements are arrays.

multiplicative operator. Performs multiplication (*), division (/), and remainder (%) operations.

multithread applications. Uses that require a number of independent actions, known as threads. These applications use threads that can begin at any function heading in a program and can coexist within the same program or even the same module.

NaN. Not-a-number.

out certain functions prior to compiling.

process. A program running under the control of an operating system. The code and data for the program and information about the status of the running program, such as the number of open files.

process-specific feature. A characteristic of compiler implementation that is true only for a single machine or class of machines, rather than being generally true.

program segment prefix. The 256-byte header to the memory segment which DOS uses to hold the compiled and linked code of a program. When a program is called, DOS puts essential run-time information into this header.

protect mode. A method of program operation that limits or prevents access to certain instructions or areas of storage.

range of values. Values for a variable from the minimum value to the maximum value that can be represented internally in a given number of bits.

real mode. A method of program operation that does not limit or prevent access to any instructions or areas of storage. The operating system loads the entire program into storage and gives the program access to all system resources.

recursive call. To call a function by name from a point within its own definition.

reference. In programming languages, a language construct designating a declared language object.

register. A storage area commonly associated with fast-access storage, capable of storing a specified amount of data such as a bit or an address.

reinitialize. To reset.

relational operators. The binary relational operators test their first operand against the second to determine if the relation specified by the operator holds true.

replicate. To copy all or a specified portion of data.

representable character set. All letters, digits, and symbols that can represent a single character.

return type. A definition of the size and type of value that the function returns.

return values. The values of expressions specified in **return** statements of functions which are made available to the functions that called them.

routine. A program or sequence of instructions called by a program that may have some general or frequent use.

run. (1) A single performance of one or more jobs. (2) A single, continuous performance of a computer program or routine.

run-time library. A collection of functions in object code form,

storage. 1) A device or part of a device that can retain data. 2) The retention of data in a storage device.

storage class. A category of variables and functions that have similar properties of lifetime and visibility.

storage location. Any addressable point in storage.

stream. A logical sequence of data items into which any input or output can be mapped.

stream function. Functions that treat a data file or data item as a stream of individual characters.

stream pointer. A pointer returned when you open a file for stream input or output.

string. In C, an array of characters whose elements have the type **char**. The C compiler adds a null character to mark the end of a string.

string literal. A sequence of letters, digits, and symbols enclosed in double quotation marks. The C compiler treats a string literal as an array of characters. Each element of the array is a single character value.

structures. Variables composed of a collection of values that may have different types.

structure tag. An identifier used as a name for any structure of a specified form.

subtraction operator. The subtraction operator (&minus) subtracts its second operand from the first.

symbol. A name that represents a register, a segment address, an offset address, or a full 32-bit address. At the C source level, a symbol (identifier) is a variable name or the name of a function.

symbolic information. Information that CodeView uses to interpret global and local program symbols.

terminate. To stop the operation of a system or device; to stop the execution of a program.

ternary expression. Three operands joined by the ternary conditional operator (**? :**).

text mode. A form of stream mapping in which character sequences are composed into lines, and for which some characters must be altered on input or output to conform to the conventions for representing them.

token. A unit of program text that has meaning to the compiler and cannot be broken down further.

trailing zeros. Insignificant zeros appearing at the end of a decimal fraction.

twos complement. A binary number notation for negative quantities.

type. A set of values together with a set of permitted operations.

Index

– (subtraction) operator 6-16
––, decrement operators 6-8
–>, member selection
operator 6-25

A

about the SETUP and INSTAID programs 2-1
about this library 1-1
abstract declarator 5-48
addition operator (+) 6-16
addition, with pointers 6-16
additive operators 6-15
address-of (&) operator 6-13
addresses 1-5
aggregate types
array 5-27
initialization 5-42
alternate floating-point library 2-21
AND operator
bitwise (&) 6-20
logical 6-21
angle brackets (< >), in #include directive 9-10
ANSI differences A-1
ANSI functions not provided A-4
ANSI standards 1-6
argc parameter 4-4
argument type list
keyword
as arg-typelist 5-33
as type name 8-15
in argument type list 5-33
in function return type 5-34
void keyword 5-33
with abstract declarator 5-48
argument types 8-14
arguments
argument type list 8-10
arguments *(continued)*
comma (,), in 5-33
conversion of 8-14
declaring 8-10
keywords, void in 5-33
maximum number of macro arguments B-1
maximum size of macro definition B-1
passing 8-14
pointer 8-12, 8-14
type names, in 5-33
type-checking 5-33, 8-14
variable number 5-33, 8-16
argv parameter 4-4
arithmetic
conversions 6-10
negation (–) operator 6-12
with pointers 6-16
arrays
declarations 5-27
elements, referring to 6-4
format 5-27
identifiers 6-2
initialization 5-45
modifier 5-8, 5-27
multidimensional 5-28
storage 6-5
type 5-8, 5-27
variables, storage of 5-28
arrow, member selection expressions 6-7
assignment
compound 6-26
connversions 6-32
expressions 6-8
how to make 6-1
listed 3-6
operators
simple (=) 6-26

compiler limits
input files B-2
maximum length B-1
of a string B-1
of an identifier B-1
maximum level B-1
of nesting B-1
of nesting for include files B-2
maximum number of macro arguments B-1
maximum size B-1
of a constant B-1
of macro definition B-1
preprocessor directives B-1
compiler passes 2-18
compiler program, CC.EXE 2-17
compiler, linker and run-time limits B-1
compiling
conditional 9-11
suppressing 9-11
compiling and linking 2-10
compiling multi-thread programs
See IBM C/2 Language Reference
complement operators 6-12
complex declarators 5-8
component libraries 2-3
component libraries, during installation 2-3
compound assignment operators 6-26
compound statement 7-3
conditional compiling 9-11
conditional operator (? :) 6-23
conditional statements
if 7-10
switch 7-15
configuration, setting 2-29
constant expressions 6-9
permissible constant-expressions 6-9
constants
case 7-15
character 3-10
constant-expressions 6-9
decimal integer 3-7
defined 3-6, 9-14
enumeration 3-11
enumeration, naming class 4-9
expression 5-24, 6-1
expression in switch statement 7-15
floating-point 3-9
floating-point, negative 3-9
hexadecimal integer 3-7
in directives 9-14
in preprocessor directives 9-14
integer 3-7
integer constant 6-2
integer, long 3-8
integer, negative 3-7
manifest 9-2
maximum size B-1
octal integer 3-7
operand 6-2
restricted 9-14
string 3-11
const, keyword 5-18
continue statement 7-4
conventions used in this book 1-3
conversions
argument type list 8-14
arithmetic 6-10
assignment 6-32
from enumeration types 6-38
from floating-point types 6-35
from pointer types 6-38
from signed integer types 6-32
from unsigned integer types 6-34
function-call 6-39
of arguments 8-14
of formal parameters 8-15
operator 6-39
standard arithmetic 6-10

Demo program, running 2-15
description of compiler EXE files 2-17
description of P.EXE files 2-18
differences from proposed ANSI standard for C A-1
digits and letters
  hdigits 3-7
  odigits 3-7
  used by IBM C/2 3-1
directives
  define 9-3
  defined 4-1
  elif 9-13
  else 9-13
  endif 9-13
  error 9-12
  if 9-13
  ifdef 9-17
  ifndef 9-17
  include 9-10
  line 9-18
  preprocessor 9-1
  undef 9-9
directories SETUP and INSTAID ask for 2-2
disk contents 2-19
  organization, sys files 2-19
division operator (/) 6-15
do
  keyword 7-5
  statement 7-2, 7-5
    continuing running 7-4
    ending 7-2
double type 5-2
  range of values 5-3
  storage 5-3
double-precision types 1-5
dynamic linking
  See IBM C/2 Language Reference

E

^ bitwise exclusive (OR) 6-20
elements of C 3-1
elements, referring to 6-4
elif directive 9-1, 9-13
ellipses, how this book uses 1-3
else directive 9-1, 9-13
else keyword 7-10
emulator library 2-19
EM.LIB 2-19
end-of-file indicator 3-2
endif directive 9-1, 9-13
enum
  expressions and identifiers 6-2
  type specifier 5-46
  types 5-46
  type, range of values 5-3
  type, storage 5-3
enumeration
  constants 3-11
    naming class 4-9
  declarations 5-19, 5-46
  example 5-21
  format 5-19
  keyword 5-20
  list 5-20
  set 5-19
  tag 5-20
  tags 5-46
  type specifier 5-19
  types 5-2, 5-19, 5-46
  types, converting 6-38
  types, storage of 5-20
  variables, storage of 5-20
environment
  maximum size B-4
  setting 2-29
  variables, setting automatically 2-6
  verifying 2-7
envp 4-4
equality operator (==) 6-19

formal parameters *(continued)*
identifiers of 8-5
storage class 8-7
type-checking 8-5
forward declarations 5-32, 8-10
function declarations 5-32
function declarations, storage class 5-41
function-call conversions 6-39
functions
body 4-1, 8-9
call expression 6-3
calls 6-3, 8-11
calls conversions 8-14
calls, indirect 8-12
declarations
defined 5-32
extern 8-10
forward 8-10
implicit 8-9
static 8-10
storage class specifier 8-10
with variable number of arguments 5-33
definition 8-1
definitions 8-1
extern 8-2
return type 8-3
static 8-2
storage class specifier 8-2
exit from 7-13
extern 8-10
format 5-32
function calls 8-1
functions 8-1
identifiers 6-2
main arguments 4-4
main in running programs 4-4
modifier 5-8
naming class 4-9
parameters 4-4
pointers 8-12
recursive 8-18
return type 8-3
functions *(continued)*
return type, implicit 8-9
return value 7-13, 8-9
returning type 5-32
static 8-10
static and extern 8-2
storage class 8-2
type names, in 5-33
visibility 8-2, 8-10
with variable number of arguments 8-16
functions, ANSI that are not provided A-4
fundamental types
initialization 5-42
range of values 5-3
storage 5-3

G

general elements of C 3-1
global filename expansion 2-20
global lifetime 4-5, 5-35
global variables 4-6
initialization 5-42
references to 5-39
global visibility 4-5
goto keyword 7-9
goto statement 7-9

H

hex bit patterns as constants 3-11
hexadecimal escape sequences 3-3
hexadecimal integer constants 3-7
hexadecimal number representation 1-4
how SETUP and INSTAID work 2-2
huge model library files 2-19

integral types 5-2
internal declarations 5-35, 5-39
internal level declarations 4-5
internal representation 5-5
internal variable
declarations 5-35, 5-39
internal variable declarations,
default storage class 5-39
interpreting complex
declarators 5-9
int, size of 5-4
iterative statements
do 7-5
for 7-7
while 7-18

K

keywords
continue 7-4
defined 3-14
enum 5-20
far 5-12
fortran 5-12
huge 5-12
interrupt 5-12
keyword 5-31
near 5-12
pascal 5-12
signed 5-3
sizeof 6-14
special 5-12
struct 5-22
void 5-34, 8-15
void, in argument type list 5-33
_loadds 5-12
_saveregs 5-12

L

labeled statements 7-9
labels
case 7-15
default 7-15
labels *(continued)*
naming class 4-10
LAN, IBM PC, installing with 2-31
large model library files 2-19
left shift (<<) operator 6-18
letters and digits 3-1
LIBH.LIB 2-21
libraries, building special 2-8
library files
floating-point 2-19
standard C 2-20
library manager 2-18
LIB.EXE 2-18
lifetime and visibility block,
defined 4-5
lifetime, defined 4-5
lifetime, directives 4-2
lifetime, global 4-5
lifetime, local 4-5, 5-35
line control 9-18
line directive 9-1, 9-18
LINE identifier 9-18
line numbers, changing 9-18
linked lists 5-22
linker limits B-2
linking, dynamic
See IBM C/2 Language Reference
LINK.EXE 2-18
listing of IBM C non-ANSI
functions A-1
lists, linked 5-22
LLIBC.LIB 2-21
LLIBFP.LIB 2-21
local lifetime 4-5, 5-35
local variables 4-6
logical operators
AND (&&) 6-21
NOT (!) 6-12
OR (||) 6-21
order of evaluation 6-21
long type
range of values 5-3
storage 5-3

NEW-VARS.BAT and
NEW-VARS.CMD 2-29
NEW-VARS.BAT and
NEW-VARS.CMD files 2-6
non-ANSI functions, listed A-1
non-ANSI identifiers A-3
non-standard, non-portable features
See IBM extensions
nongraphic characters in C programs 3-5
nongraphic escape sequences 3-3
notations used in this book 1-3
null statement 7-12
number sign (#) character 9-1
numeric coprocessor 2-19
numeric coprocessor, using 2-30

O

octal escape sequences 3-3
octal integer constants 3-7
opening the C runtime libraries, in OS/2 2-7
operands
binary expression 6-8
constant 6-2
conversions 6-11
defined 6-1
string literals 6-3
ternary expression 6-8
unary expression 6-8
operating system
abbreviations 1-4
operator conversions 6-39
operators
addition (+) 6-16
address-of (&) 6-13
arithmetic and logical 3-5
arithmetic negation (—) 6-12
assignment 3-6, 6-24
associativity 6-27
binary 6-10
bitwise 6-10
AND (&) 6-20
exclusive OR 6-20
operators *(continued)*
bitwise *(continued)*
inclusive OR (|) 6-20
complement 6-12
compound assignment 6-26
conditional (? :) 6-23
constant 6-1
decrement 6-25
division (/) 6-15
equality (= =) 6-19
increment 6-25
indirection (*) 6-13
inequality (!=) 6-19
left shift (<<) 6-18
listed 3-5
logical 6-21
logical AND 6-21
logical not (!) 6-12
logical OR 6-21
logical, order of evaluation 6-21
multiplication (*) 6-15
multiplicative 6-14
precedence 6-27
relational 6-19
remainder (%) 6-15
right shift (>>) 6-18
sequential evaluation (,) 6-22
shift 6-10, 6-18
simple assignment (=) 6-26
sizeof 6-14
subtraction (—) 6-16
ternary 6-10
ternary (? :) 6-23
unary 6-10
unary plus (+) 6-12
with expressions 6-8
&I2@OPR.
complement 6-12
optional files 2-21
OR operator
bitwise 6-20
logical 6-21

remainder operator (%) 6-15
removing macro and manifest constant definitions 9-9
representable character set 3-1
representation, internal 5-5
required space for installation 2-4
reserved words, keywords 3-14
restricted constant expressions 6-9
  in directives 9-14
return keyword 7-13
return statement 7-13
return type
  declaring 8-10
  implicit 8-9
  in function declaration 5-34
  in function definitions 8-3
return value 7-13, 8-9
returning control 7-13
right shift (>>) operator 6-18
run-time limits B-4
  maximum length of command prompt B-4
  program limits at run-time B-4
  runtime B-4
running programs
  main function 4-4
  starting point 4-4
  the Demo program 2-15
runtime libraries, opening in OS/2 mode 2-7

S

sC specifier 5-1, 5-18
search path, include files 9-10
selecting component library files for LIB 2-30
selection statements
  if 7-10
  switch 7-15
sequential evaluation operator (,) 6-22
setting environment variables automatically 2-6
setting the configuration 2-29
setting the environment 2-29
setting up on other devices 2-22
SETUP program
  installing with 2-1
  memory used during 2-2
  what SETUP does 2-2
  /L option 2-8
shift operators 6-10, 6-18
short type
  range of values 5-3
  storage 5-3
side effects
  defined 6-30
  in expressions 8-12
  in macros 9-5
sign # character 9-1
signed types 5-2
  converting 6-32
  double 5-2
  enumeration 5-2
  float 5-2
  int 5-2
  long 5-2
  short 5-2
  signed char 5-2
  signed int 5-2
  signed short int 5-2
  unsigned char 5-2
  unsigned int 5-2
  unsigned long 5-2
  unsigned short 5-2
simple and pointer initialization 5-42
simple assignment operator (=) 6-26
simple type argument 8-14
simple variable declarations 5-19
single-precision types 1-5
sizeof operator 6-14

storage class specifiers, formal parameters 8-7
storage class specifiers, in forward declarations 8-10
storage classes
  class specifiers 5-35
  default, internal variable declarations 5-39
  defined 5-35
  in forward declarations 8-10
  in function definitions 8-2
  local 5-35
  specifiers 5-35
  specifiers with external variables 5-36
  specifiers, global 5-35
  static, with external variables 5-36
string initializers 5-45
string literals
  as initializers 5-45
  format 3-11
  storage of 3-12
stringizing operator, # 9-6
strings 6-3
struct keyword 5-22
struct type specifier 5-21, 5-46
struct types 5-21, 5-46
structure
  declarations 5-21
  example 5-25
  expressions 6-2
  format 5-21
  identifiers 6-2
  keyword 5-22
  members 5-21
  member, bit-field 5-22
  tag 5-22
  tags 4-9, 5-46
    naming class 4-9
  type argument 8-14
  types 5-21, 5-46
  types, storage of 5-24
  variables, storage of 5-24
subscript expressions 6-4
subtraction operator (—) 6-16
subtraction, with pointers 6-16
suppressing compiling 3-15, 9-11
switch expression 7-15
switch statement 7-2, 7-15
  body 7-15
  ending 7-2
SYS subdirectory 2-19
system configuration 2-7
system configuration, setting 2-29
system-level definitions 2-19

T

tags
  enumeration 5-20, 5-46
  naming class 4-9
  structure 5-22, 5-46
  union 5-46
terms used in the C/2 library 1-4
ternary expressions 6-8
ternary operator 6-10
ternary operator (? :) 6-23
text data files 2-21
text mode 2-21
token-pasting operator (##) 9-8
tokens 3-16
transfer statements
  break 7-2
  continue 7-4
  do, ending 7-2
  goto 7-9
  labeled statements 7-9
twos complement operator 6-12
type cast conversions 6-38
type conversions 6-1, 6-31
  types
    signed, converting 6-32
type declarations 5-45, 5-46
type declarations, typedef 5-45
type names 5-48
type specifiers
  abbreviations 5-3

types, array, storage of 5-28
types, enumeration 5-19
types, struct 5-21
types, structure, storage of 5-24
types, union 5-26
types, union, storage 5-26

U

unary
  increment and decrement 6-25
unary expressions 6-8
unary operators 6-10
unary plus operator 6-12
undef directive 9-1, 9-9
undefine directive 9-9
union
  declarations 5-46
  identifiers 6-2
  members, naming class 4-9
  members, referring to 6-7
  tags 5-46
    naming class 4-9
  type specifier 5-46
  types 5-46
union expression 6-2
union type argument 8-14
unsigned integer types,
 converting 6-34
unsigned types
  char 5-2
  int 5-2
  long 5-2
  short 5-2
user-defined types 5-45
using
  a numeric coprocessor 2-30
  batch files 2-16
  NEW-VARS.BAT and
    NEW-VARS.CMD during instal-
    lation 2-6
  the CC and LINK
    commands 2-11
  the CL command 2-10
utilities
  EXEMOD, See
    EXEMOD.EXE 2-18
  LIB, See LIB.EXE 2-18
  linker, See LINK.EXE 2-18

V

values, ranges 5-5
variable declarations
  array 5-27
  internal 5-39
  internal, default storage
    class 5-39
  pointer 5-31
  union 5-26
variables
  array, initialization 5-45
  auto, initialization 5-42
  declarations 4-1, 5-18
    enum 5-19
    simple 5-19
    structure 5-21
  declarations, external 5-35
  declarations, internal 5-35
  definitions 5-36
  definitions in program
    structure 4-1
  enumeration, storage of 5-20
  global 4-6
  global, initialization 5-42
  global, references to 5-39
  local 4-6
  naming class 4-9
  pointer, storage 5-31
  register, initialization 5-42
  static, initialization 5-42
  structure, storage of 5-24
  variable number 8-16
variables, array, storage of 5-28
variables, union, storage 5-26
verifying environment 2-7
verifying installed options 2-4

Continued from inside front cover.

SUCH WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

LIMITATION OF REMEDIES

IBM's entire liability and your exclusive remedy shall be as follows:

1) IBM will provide the warranty described in IBM's Statement of Limited Warranty. If IBM does not replace defective media or, if applicable, make the Program operate as warranted or replace the Program with a functionally equivalent Program, all as warranted, you may terminate your license and your money will be refunded upon the return of all of your copies of the Program.
2) For any claim arising out of IBM's limited warranty, or for any other claim whatsoever related to the subject matter of this Agreement, IBM's liability for actual damages, regardless of the form of action, shall be limited to the greater of $5,000 or the money paid to IBM, its Authorized Dealer or its approved supplier for the license for the Program that caused the damages or that is the subject matter of, or is directly related to, the cause of action. This limitation will not apply to claims for personal injury or damages to real or tangible personal property caused by IBM's negligence.
3) In no event will IBM be liable for any lost profits, lost savings, or any incidental damages or other consequential damages, even if IBM, its Authorized Dealer or its approved supplier has been advised of the possibility of such damages, or for any claim by you based on a third party claim.

Some states do not allow the limitation or exclusion of incidental or consequential damages so the above limitation or exclusion may not apply to you.

GENERAL

You may terminate your license at any time by destroying all your copies of the Program or as otherwise described in this Agreement.

IBM may terminate your license if you fail to comply with the terms and conditions of this Agreement. Upon such termination, you agree to destroy all your copies of the Program.

Any attempt to sublicense, rent, lease or assign, or, except as expressly provided herein, to transfer any copy of the Program is void.

You agree that you are responsible for payment of any taxes, including personal property taxes, resulting from this Agreement.

No action, regardless of form, arising out of this Agreement may be brought by either party more than two years after the cause of action has arisen except for breach of the provisions in the Section entitled "License" in which event four years shall apply.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.

Z125-3301-02 4/87