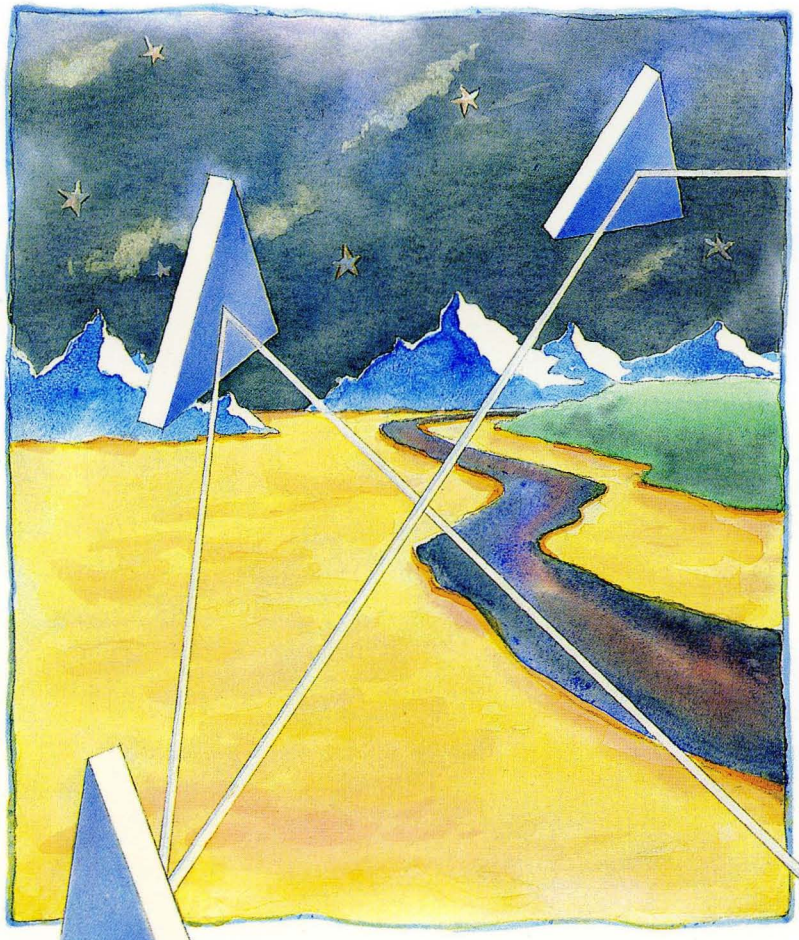


Domain C Language Reference

002093-A00

apollo

Domain C Language Reference

Order No. 002093-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: October 1982
Latest Printing: July 1988

This document was produced using the Interleaf Technical Publishing Software (TPS) and the InterCAP Illustrator I Technical Illustrating System, a product of InterCAP Graphics Systems Corporation. Interleaf and TPS are trademarks of Interleaf, Inc.

Preface

The *Domain C Language Reference* manual describes the Domain C programming language and the Domain programming environment relevant to C programmers.

We've organized this manual as follows:

Chapter 1	Presents an overview of the Domain C compiler.
Chapter 2	Describes the lexical components of a C program (such as identifiers, comments, and keywords), and describes the general organization of C programs.
Chapter 3	Describes data types and storage classes, and the syntax and semantics of declaring variables.
Chapter 4	Provides encyclopedic descriptions of all C language statements and operators, as well as descriptions of general C programming concepts.
Chapter 5	Provides details about declaring and invoking functions.
Chapter 6	Describes compiler options and the compilation/linking process.
Chapter 7	Describes how to call FORTRAN and Pascal routines from a C program, and how to share global data with routines written in other languages.
Chapter 8	Provides an overview of input and output operations that can be performed with the standard C run–time library and the UNIX system library.
Chapter 9	Describes the types of diagnostic messages that the compiler issues, and lists each message along with its probable cause.
Appendix A	Lists the ISO Latin–1 code values.
Appendix B	Lists Domain extensions to the C programming language.

Appendix C	Describes the BSD version of the **lint** utility.
Appendix D	Describes the SysV version of the **lint** utility.
Appendix E	Describes the **std_$call** keyword, which is now obsolete.

Revision History

Because this manual has been extensively revised, we have not used marginal change bars to indicate each modification. See the *C Compiler Release Document* for a list of functional changes to the C compiler.

Related Manuals

For more information about the standard C run-time library and UNIX system calls, see the *BSD Programmer's Reference* manual (005801) and the *SysV Programmer's Reference* manual (005799).

For more information about system calls see the *Domain/OS Call Reference* manual (007196) and *Programming with Domain/OS Calls* (005506).

For more information about the programming environment and software tools, see the *Domain/OS Programming Environment Reference* manual (011010).

For more information about the Domain Pascal programming language, see the *Domain Pascal Programming Language Reference* manual (000792).

For information about the Domain FORTRAN programming language, see the *Domain FORTRAN Programming Language Reference* manual (000530).

For more information about the binder (**bind**), link editor (**ld**), librarian (**lbr**), and archiver (**ar**), see the *Domain/OS Programming Environment Reference* manual (0011010).

For more information about the Domain high-level debugger, see the *Domain Distributed Debugging Environment Reference* manual (011024).

For more information about DSEE, see the *Domain Software Engineering Environment (DSEE) Reference* manual (003016).

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. To make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for comments related to hardware, software, and documentation. By using this formal channel, you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** (make apollo product report) shell command description. You can view the same description online by typing:

$ man mkapr (in the SysV environment)

% man mkapr (in the BSD environment)

$ help mkapr (in the Aegis environment)

Alternatively, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user-supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
Domain extensions	Domain-specific features of C appear in color.
sample user input	In samples, information that the user enters appears in color.
`output`	`Information that the system displays appears in this typeface.`
[ ]	Square brackets enclose optional items in formats and command descriptions.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.

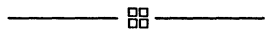
. . .

Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

.
.
.

Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Because this manual has been extensively revised, we have not used marginal change bars to indicate each modification.



This symbol indicates the end of a chapter.

Contents

Chapter 1 Overview of Domain C

Chapter 2 Program Organization

Chapter 3 Data Types and Storage Classes

Chapter 4 Code

Chapter 5 Functions

Chapter 6 Program Development

Chapter 7 Cross-Language Communication

Chapter 8 Input and Output

Chapter 9 Diagnostic Messages

Appendix A ISO Latin–1 Codes

Appendix B Domain C Extensions

Appendix C The lint Utility (BSD)

Appendix D The lint Utility (SysV)

Appendix E Using std$_call

Figures

Tables

Bug Alerts

Chapter 1

An Overview of Domain C

This manual describes Domain® C, which is our implementation of the C programming language. In this chapter, we provide an overview of the C language, list some of the key Domain extensions, and show how to compile and execute a simple C program.

1.1 History of C

The C language was first developed in 1972 by Dennis M. Ritchie at AT&T Bell Labs as a systems programming language—that is, a language to write operating systems and system utilities. Ritchie's intent in designing C was to give programmers a convenient means of accessing a machine's instruction set. This meant creating a language that was high-level enough to make programs readable and portable, but simple enough to map easily onto the underlying machine architecture.

C was so flexible, and enabled compilers to produce such efficient machine code, that in 1973 Ritchie and Ken Thompson rewrote most of the UNIX* operating system in C. Since then, C and the UNIX system have had a close association, although in recent years C has become more popular as a general-purpose programming language.

Although the power and flexibility of C is undisputed, C has also acquired the reputation for being a mysterious and messy language that promotes bad programming habits. Part of the problem is that C gives special meanings to many punctuation characters, such as asterisks, plus signs, braces, and angle brackets. Once a programmer has learned the C language, these symbols look quite commonplace, but there is no denying that a typical C program can be intimidating to the uninitiated.

The other, more serious, complaint concerns the relative dearth of rules. Other programming languages, such as Pascal, have relatively strict rules to protect programmers from

*UNIX is a registered trademark of AT&T in the USA and other countries.

making accidental blunders. It is assumed in Pascal, for instance, that if a programmer attempts to assign a floating–point number to a variable that is supposed to hold an integer, it is a mistake, and the compiler issues an error message. In C, the compiler quietly converts the floating–point value to an integer.

The C language was designed for experienced programmers. The compiler, therefore, assumes little about what the programmer does or does not intend to do. This can be summed up in the C tenet:

Trust the programmer.

As a result, C programmers have tremendous liberty to write unusual code. In many instances, this freedom allows programmers to write useful programs that would be difficult to write in other languages. However, the freedom can be abused by inexperienced programmers who delight in writing needlessly tricky code. C is a powerful language, but it requires self–restraint and discipline.

You should be somewhat familiar with C before attempting to use this manual. If you are not, please consult a good C tutorial. If you are familiar with C, you should be able to write programs in Domain C after reading this manual.

1.2 C Standards

Until recently, the only formal specification for the C language was a document written by Dennis Ritchie entitled *The C Reference Manual.* In 1977, Ritchie and Brian Kernighan expanded this document into a full–length book called *The C Programming Language* (sometimes called "the white book" because of its white cover). For years, *The C Programming Language* was the only C text and so acquired the status of a *de facto* standard. We refer to this book, and the language it defines, as the *K&R standard.*

In the early days of C, the language was used primarily on UNIX systems. Even though there were different versions of UNIX systems available, each used the same C compiler. The version of C running under a UNIX operating system is known as PCC (Portable C Compiler). Like the K&R standard, PCC became a *de facto* standard. In fact, PCC can be viewed as an implementation of the K&R standard. There are a few points about the C language, however, that the K&R standard does not define. In these cases, the PCC implementation has become the standard.

In February 1983, James Brodie of Motorola Corporation applied to the X3 Committee of the American National Standards Institute (ANSI) to draft a C standard. ANSI approved the application, and in March the X3J11 Technical Committee of ANSI was formed. X3J11 is composed of representatives from all the major C compiler developers (including Apollo), as well as representatives from several companies that program their applications in C. In the summer of 1983, the committee met for the first time, and they have been meeting four times a year since then. The final version of the C standard is expected to be approved by ANSI in 1988.

In addition to the K&R standard, the PCC implementation, and the ANSI standard, there is a new language based on C called C++. C++ was developed by Bjarne Stroustrup at AT&T. It includes many of the features in the ANSI standard, as well as further extensions to make the language object-oriented.

Except for a few rare cases, Domain C is fully compatible with the K&R standard and with PCC. Therefore, programs compiled in a UNIX environment can be ported to Apollo machines without altering the source text, and vice versa. At the same time, Domain C supports many of the newer features introduced by ANSI and C++. In particular, Domain C supports the following:

- **enum** data type
- Function prototypes
- Reference variables
- Generic pointers

Finally, Domain C includes some features that are not available in any of the existing standards. These features enable you to take full advantage of the Domain/OS environment, though use of special Domain syntaxes will make your programs less portable.

Throughout this manual, we highlight all Domain-specific features in colored text. Everything printed in black is consistent with either the K&R standard or the ANSI standard. Where the two standards differ, we explicitly state the difference in the text. Appendix D contains a detailed list of ANSI and C++ features that Domain C supports.

1.3 Two Ways to Call C

Although there is only one Domain C compiler, there are two command line interfaces to it. By default, typing **cc** in a UNIX shell gives you the **/bin/cc** interface. Typing **cc** in an Aegis shell gives you the **/com/cc** interface.

The **/com/cc** interface is always available regardless of what shell you are running and which environments are installed on your node. If you are in a UNIX shell and have Aegis installed on your node, you can access the **/com/cc** interface by typing **/com/cc** on the command line. If Aegis is not installed on your node, the **/com/cc** interface will reside in **/usr/apollo/lib/cc**. Note, however, that you can also access the **/com/cc** interface by using the **-Y0** option with the **/bin/cc** command. See Chapter 6 for more information about this compiler option.

The **/bin/cc** interface is available *only* if a UNIX environment is installed on your node. If a UNIX environment is installed but you are running an Aegis shell, you can access the **/bin/cc** interface by typing **/bin/cc** on the command line.

The **/bin/cc** command first calls the UNIX preprocessor (**cpp**); then it invokes the Domain C compiler; after compilation, it invokes the UNIX link editor (**ld**).

The **/com/cc** command only invokes the Domain C compiler (which includes the Aegis preprocessor). Unlike the **/bin/cc** command, **/com/cc** does not automatically invoke a link editor. See Chapter 6 for more information about the differences between **/com/cc** and **/bin/cc**.

1.3.1 Two Preprocessors

The C product supports two preprocessors—a UNIX preprocessor called **cpp** and an Aegis preprocessor that is bundled with the Domain C compiler. The UNIX preprocessor is automatically invoked whenever you execute the **/bin/cc** command. You can also invoke it as a stand-alone utility by executing the **/usr/lib/cpp** command. The Aegis preprocessor executes whenever you invoke the Domain C compiler. Note that when you compile in a UNIX environment, your source text is passed through both preprocessors—**cpp** first and then the Aegis preprocessor.

In general, the two preprocessors behave identically. The key differences are:

- The two preprocessors use different methods for resolving relative pathnames in **#include** directives. See the description of the **#include** directive in Chapter 4 for more information about this difference.
- The two preprocessors support different sets of command options. See Chapter 6 for details about all command options.
- The UNIX preprocessor supports the **#elif** directive; the Aegis preprocessor does not.
- The Aegis preprocessor supports many Domain-specific directives and predefined macros that **cpp** does not support.

1.3.2 Two Styles of Object Code

Both **/bin/cc** and **/com/cc** produce COFF (Common Object File Format) object files. However, the two commands produce slightly different styles of COFF. The notable differences are:

- Object files produced by **/bin/cc** have a **.o** suffix. Object files produced by **/com/cc** have a **.bin** suffix.
- If you compile with **/bin/cc**, the resulting code will not be optimized by default. If you compile with **/com/cc**, your code will be optimized at optimization level 3. You can override both of these defaults with compiler options. See Chapter 6 for more information about optimization levels.

- If you compile with **/bin/cc**, all uninitialized global variables will be placed in the **.bss** section of the object file. If you compile with **/com/cc**, all global variables will be placed in named overlay sections. This becomes an issue in cross-language communication, as explained in Chapter 7.
- Object files compiled by **/com/cc** are executable if they contain a **main()** function and do not reference externally defined objects. All object files produced by **/bin/cc** must be processed by a binder before they can be executed. Note that since **/bin/cc** automatically invokes the link editor (**ld**), this difference is usually invisible.

1.3.3 Two Command Line Syntaxes

The **/bin/cc** and **/com/cc** commands have separate syntaxes and recognize entirely different sets of command line options (although the functionality overlaps to a large extent). Chapter 6 describes these differences in detail. Here, we briefly list some of the principal differences.

- The **/bin/cc** command accepts multiple source filenames on the command line. The **/com/cc** command accepts only one filename.
- With **/bin/cc**, you can specify the names of object files, which are passed to the link editor. The **/com/cc** command accepts only source files.
- When you compile with **/bin/cc**, all source filenames must have a **.c** suffix and all object filenames must have a **.o** suffix. There are no suffix requirements with the **/com/cc** command.
- With the **/bin/cc** command, you must place compiler options before filenames. With **/com/cc**, compiler options are placed after the filename.
- The compiler options supported by **/bin/cc** are case-sensitive. The **/com/cc** compiler options are not case-sensitive.

1.4 A Sample Program

The best way to get started with Domain C is to write, compile, and execute a simple program. Here is a simple program to get you started:

```
/* Program name is "getting_started" */
#include <stdio.h>

int main( void )
{
  int x, y;
  printf( "enter an integer -- " );
  scanf( "%d", &x );
  y = x * 2;
  printf( "\n%d is twice %d\n", y, x );
}
```

1.4.1 Compiling and Executing

Suppose that you store this program in a file named **getting_started.c**. (If you use **/bin/cc**, you must enter the full name of the source file, including the .c suffix; with **/com/cc**, you may omit the **.c** suffix.) Compiling with **/com/cc** produces an executable object file named **getting_started.bin**; compiling with **/bin/cc** produces an executable binary file named **a.out**. To run these objects, just enter the name of the file. Table 1-1 summarizes the whole process.

Table 1-1. Compiling and Executing a Simple Program

With /com/cc	With /bin/cc
$ /com/cc getting_started No errors, No warnings. **$ getting_started.bin** Enter an integer -- **15** 30 is twice 15	**$ /bin/cc getting_started.c** No errors, No warnings. **$ a.out** Enter an integer -- **15** 30 is twice 15

1.5 Online Sample Programs

Many of the programs from this manual are stored online, along with sample programs from other Apollo manuals. These programs illustrate features of the C language, and demonstrate programming with Domain/OS graphics calls and system calls. There are two ways to access these online programs—with the **getcc** utility or with the Delphi system.

1.5.1 Accessing Sample Programs with getcc

The **getcc** utility enables you to extract a program from a master file that contains all sample programs. The **getcc** utility prompts you for the name of the sample program and the pathname of the file to which you want it copied.

If the online examples are stored on your node, you can access getcc directly or through a link. To access them directly, you must change your working directory before invoking getcc:

```
# In an Aegis shell
$ wd /domain_examples/cc_examples
$ getcc
```

```
# In a UNIX shell
$ cd /domain_examples/cc_examples
$ getcc
```

To access the examples through a link, you need to create the following link before invoking **getcc**:

```
#       In an Aegis shell
$ crl -/com/getcc  /domain_examples/cc_examples/getcc
$ getcc

#       In a UNIX shell
$ ln -s /domain_examples/cc_examples/getcc path_dir/getcc
$ getcc
# where "path_dir" is a name of a directory on your list of search
# pathnames.
```

If the online examples are stored on a remote node, you need to create the following two links to invoke **getcc**:

```
#       In an Aegis shell
$ crl /domain_examples/cc_examples //node/domain_examples/cc_examples
$ crl -/com/getcc  //node_name/domain_examples/cc_examples/getcc
$ getcc

#       In a UNIX shell
$ ln -s //node/domain_examples/cc_examples /domain_examples/cc_examples
$ ln -s //node/domain_examples/cc_examples/getcc  path_dir/getcc
$ getcc
# where "node" is the name of the node where the examples are
# stored, and "path_dir" is a name of a directory on your list of
# search pathnames.
```

1.5.2 Accessing Sample Programs with Domain/Delphi

All of the sample programs are available through the Delphi online documentation system. To compile and run an example, enter the name of the program in the Domain/Delphi subject field. When the source for the program appears, cut it and paste it into another file. You can then compile and execute this file as you would any other source file. See the *Retrieving Information With Domain/Delphi* manual for more information.

Chapter 2

Program Organization

This chapter describes the following subjects:

- Lexical elements of a C program
- Organization of a C program
- Constants
- Declarations

2.1 Lexical Elements

The lexical elements of the C language include the characters that may appear in a C source file, and how these characters are grouped into meaningful tokens by the Domain C compiler. In particular, we describe the following syntactic objects:

- White space and newlines
- Comments
- Identifiers
- Keywords
- Constants

2.1.1 White Space and Newlines

In C source files, blanks, newlines, vertical tabs, horizontal tabs, and formfeeds are all considered to be **white space** characters. The main purpose of white space characters is to format source files so that they are more readable to humans. In general, the compiler ignores white space characters, except when they are used to separate tokens or when they appear within string literals. The newline character also serves the special function of terminating preprocessor directives. See the "Preprocessor Directives" section in Chapter 4 for more information about preprocessor directives.

2.1.2 Comments

A comment is any series of characters beginning with /* and ending with */. The compiler ignores all comments. In the following example, a comment follows an assignment statement:

```
average = total / number_of_components; /* Find mean value. */
```

Comments may also span multiple lines, as in:

```
/*  This is a
    multi-line comment.
 */
```

Domain C allows comments to appear anywhere in the source file. Since the compiler interprets comments as nulls, this can result in unusual concatenations if you are not careful. For instance, the statement,

```
int x/* This is an example */z;
```

becomes:

```
int xz;
```

> **NOTE:** Domain C's implementation of comments conforms to the PCC implementation. The ANSI standard, however, states that comments must be replaced by a single space character.

The C language does not support nested comments. The following, for example, will produce a compile-time error:

```
/* This is an outer comment
 * /* This is an attempted inner comment -- WRONG */
 *
 *  This will be interpreted as code.
 */
```

C identifies the beginning of a comment by the character sequence /*. It then strips all characters up to, and including, the end comment sequence */. What's left gets passed to the compiler to be further processed. In the example above, therefore, the preprocessor will delete everything up to the first */ sequence, but pass the rest to the compiler. So the compiler will attempt to process:

```
*
*  This will be interpreted as code.
*/
```

Not recognizing these lines as valid C statements, the compiler will issue an error message. You can check for nested comments by compiling with the **-comchk** option (available with **/com/cc** only).

2.1.3 Spreading Source Code Across Multiple Lines

In C, you can start a statement or declaration at any column and spread it over as many lines as you want. In older versions of C, including the K&R standard, you cannot split a keyword or identifier across a line. Domain C, in conformance with the ANSI standard, defines the continuation character more generally, allowing you to use it to split identifiers and tokens as well as strings. For example, the compiler views the following two lines as the keyword **switch**:

```
swit\
ch
```

You can split a string or preprocessor directive across one or more lines. (See Chapter 3 for a definition of strings.) To split a string or preprocessor directive, however, you must use the **continuation character** (\) at the end of the line to be split; for example:

```
#define foo_macro(x,y,z) ((x) + (y))\
                         * ((z) - (x))

printf("This is an very, very, very lengthy and very, very \
uninteresting string.");
```

2.1.4 Identifiers

Identifiers, also called **names**, can consist of the following:

- Letters (ASCII decimal values 65–90 and 97–122)
- Digits
- Dollar sign ($)
- Underscore (_)

The first character must be a letter or an underscore. Identifiers that begin with an underscore are generally reserved for system use. In fact, the ANSI standard has reserved all names that begin with two underscores or an underscore followed by an uppercase letter. Note that the dollar sign is a Domain extension.

In addition, identifiers may not conflict with reserved keywords, which are listed in Figure 2–1. Table 2–1 lists some legal and illegal identifiers:

Table 2–1. Legal and Illegal Identifiers

Identifier	Legal or Illegal
meters	Legal.
green_eggs_and_ham	Legal.
system_name	Legal.
UPPER_AND_lower_case	Legal.
20_meters	Illegal, because it starts with a digit.
$name	Illegal, because it starts with a dollar sign.
name$	Legal in Domain C, but nonstandard.
int	Illegal, because **int** is a reserved keyword.
no%#@good	Illegal, because it contains illegal characters.

Identifiers are unique up to 4096 characters. Because Domain C exceeds the limits required by the K&R and ANSI standards, long names may not be portable. The ANSI standard requires compilers to support names of up to 32 characters for local variables and 6 characters for global variables.

2.1.5 Case Sensitivity

In C, identifier names are always case-sensitive; that is, an identifier written in uppercase letters is considered different from the same identifier written in lowercase. For example, the following three identifiers are all considered unique:

```
kilograms
KILOGRAMS
Kilograms
```

Some Domain/OS programming languages (such as Pascal and FORTRAN) are case-insensitive. When writing a Domain C program that calls routines from these other languages, you must be aware of this difference in sensitivity. (See Chapter 7 for details on cross-language communication.)

Note that strings (discussed in Chapter 3) are also case-sensitive. That is, the system recognizes the following two strings as distinct:

```
"THE RAIN IN SPAIN"
"the rain in spain"
```

2.1.6 Keywords

Domain C supports the list of keywords shown in Figure 2-1. You cannot use keywords as identifiers; if you do, the compiler will report an error. You cannot abbreviate a keyword and you must enter keywords in lowercase letters only.

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

Figure 2-1. Domain C Keywords

2.2 Constants

There are four types of constants in C:

- Integer constants
- Floating-point constants
- Character constants
- String constants

Every constant has two properties: **value** and **type**. For example, the constant 15 has value 15 and type **int**.

2.2.1 Integer Constants

An integer constant is a simple number like 12 as opposed to an integer variable (like **x** or **y**) or an integer expression. Whenever you use an integer constant in your source code, Domain C represents it as an **int** (32 bits). You cannot change this default. However, you can append an **l** or **L** to any constant to specify that you want it **long**. For example, **55L** is a constant with a decimal value of 55 and the storage size of a **long int**. Since **long** and **int** have the same meaning in Domain C, the **l** or **L** is redundant. You may still want to use it, though, if you are planning to port your programs to a non-Apollo machine.

If the constant value cannot fit in a **long int**, the results are unpredictable. However, the compiler will not report an error.

Domain C supports three forms of integer constants: decimal, octal, and hexadecimal. Decimal constants consist of one or more digits from 0-9 (but *not* starting with 0). Octal constants are formed by preceding the constant with a zero(0); hexadecimal constants are formed by preceding the constant with **0x** or **0X**. Hexadecimal constants consist of the digits 0-9 and the letters a-f (or A-F).

Integer constants may not contain any punctuation such as commas or periods. The following examples show some legal constants in all three forms.

Decimal	Octal	Hexadecimal
3	003	0x3
8	010	0x8
15	017	0xF
16	020	0x10
21	025	0x15
-87	-0127	-0x57
187	0273	0xBB
255	0377	0xff

Strictly speaking, constants are always positive values. A negative constant is interpreted as a positive constant preceded by the unary negation operator. In practice, this distinction is moot.

Technically, an octal constant cannot contain the digits 8 and 9 since they are not part of the octal number set. The Domain C compiler accepts 8 and 9 in octal numbers but issues a warning message. For example, the statement

```
x = 098;
```

compiles successfully, but a warning message appears. The compiler interprets this value to mean 9 eights plus 8 ones, so that 098 has a decimal value of 80. (The ANSI Standard does not support this feature.)

2.2.2 Floating-Point Constants

A floating-point constant is any number that contains a decimal point and/or exponent sign for scientific notation. All floating-point constants are of type **double** even if they can be accurately represented in four bytes. If the magnitude of a floating-point constant is too great or too small to be represented in a **double**, the C compiler will substitute a value that can be represented. This substitute value is not always predictable. See Chapter 3 for a description of the representable ranges of floating-point types.

2.2.2.1 Scientific Notation

Scientific notation is a useful shorthand for writing lengthy floating-point values. In scientific notation, a value consists of two parts: a number called the **mantissa** followed by a power of 10 called the **characteristic** (or **exponent**). The letter **e** or **E**, standing for exponent, is used to separate the two parts. The floating-point constant **3e2**, for instance, is interpreted as $3*10^2$, or 300. Likewise, the value **-2.5e-4** is interpreted as $-2.5*10^{-4}$, or -0.00025. Table 2-2 shows some legal and illegal floating-point constants.

Table 2-2. Floating-Point Constants

Constant	Legal or Illegal
3.	Legal.
35	Legal — Interpreted as an integer.
3.141	Legal.
3,500.45	Illegal — commas are illegal.
.3333333333	Legal.
4E	Illegal — the exponent sign must be followed by a number.
0.3	Legal.
3e2	Legal.
4e3.6	Illegal — the exponent must be an integer.
3.0E5	Legal.
+3.6	Illegal — Domain C doesn't support a unary plus sign.
0.4E-5	Legal.

2.2.3 Character Constants

A **character constant** is any printable character or legal escape sequence enclosed in single quotes. The value of a character constant is the integer ASCII (or ISO) value of the character. For example, the value of the constant `x' is 120.

2.2.3.1 Escape Characters

Domain C supports several predefined character constants known as escape characters. They are listed in Table 2-3.

Table 2-3. Character Escape Codes

Escape Code	Character	What It does
\b	backspace	Moves the cursor back one space.
\f	formfeed	Moves the cursor to the next logical page.
\n	newline	Prints a newline.
\r	carriage return	Prints a carriage return.
\t	horizontal tab	Prints a horizontal tab.
\v	vertical tab	Prints a vertical tab.
\'	single quote	Prints a single quote.
\"	double quote	Prints a double quote.

In addition to the escape sequences listed in Table 2–3, C also supports escape character sequences of the form:

\octal-number

and

\xhex-number

which translates into the character represented by the octal or hexadecimal number. For example, if ASCII representations are being used, the letter 'a' may be written as '\141' or '\x61' and 'Z' as '\132' or '\x5A'. This syntax is most frequently used to represent the null character as '\0'. This is exactly equivalent to the numeric constant zero (0). When you use the octal format, you do not need to include the zero prefix as you would for a normal octal constant.

2.2.3.2 Multi–Character Constants

Each character in a character constant takes up one byte of storage; therefore, you can store up to a four–byte character constant in a 32–bit integer and up to a two–byte character constant in a 16–bit integer. For example, the following assignments are quite legal (though not recommended and probably not portable):

```
{
   char      x;     /* one-byte integer */
   short int si;    /* two-byte integer */
   long int  li;    /* four-byte integer */

   x  = 'j';        /* one-byte character constant */
   si = 'ef';       /* two-byte character constant */
   li = 'abcd';     /* four-byte character constant */
}
```

The variable **si** is assigned the value of **'e'** *and* **'f'**, where each character takes up 8 bits of the 16–bit value. The Domain C compiler places the last character in the rightmost (least significant) byte. Therefore, the constant **'ef'** will have a hexadecimal value of 6566. Since the order in which bytes are assigned is machine dependent, other machines may reverse the order, assigning **f** to the more significant byte. In that case, the resulting value would be 6665. For maximum portability, we recommend that you do not use multi–character constants.

2.2.4 String Constants

A string constant is any series of printable characters or escape characters enclosed in double quotes. The compiler automatically appends a **null character** ('\0') to the end of the string so that the size of the array is one greater than the number of characters in the string. For example,

```
"A short string"
```

becomes an array with 15 elements:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A		s	h	o	r	t		s	t	r	i	n	g	\0

To span a string constant over more than one line, use the backslash character (\), also called the **continuation character**. The following, for instance, is legal:

```
string = "This is a very long string that requires more \
than one line";
```

Note that if you indent the second line, the spaces will be part of the string.

In Domain C, the length of a string constant is limited to 4095 characters including the trailing null character. This limit may differ on other implementations.

The type of a string is **array of char**, and strings obey the same conversion rules as other arrays. Except when a string appears as the operand of **sizeof** or as an initializer, it is converted to a pointer to the first element of the string. Note also that the **null string**,

```
""
```

is legal, and contains a single trailing null character.

2.3 Program Organization

When you write a Domain C program, you can put all your source code into one file or spread it across many files. Figure 2–2 shows a simplified scheme for organizing C source files. The C language permits other file organizations that are not depicted in the figure. For example, most preprocessor directives may appear anywhere in a source file, and global declarations may appear between functions. The figure, however, depicts a general organization that reflects many C programs.

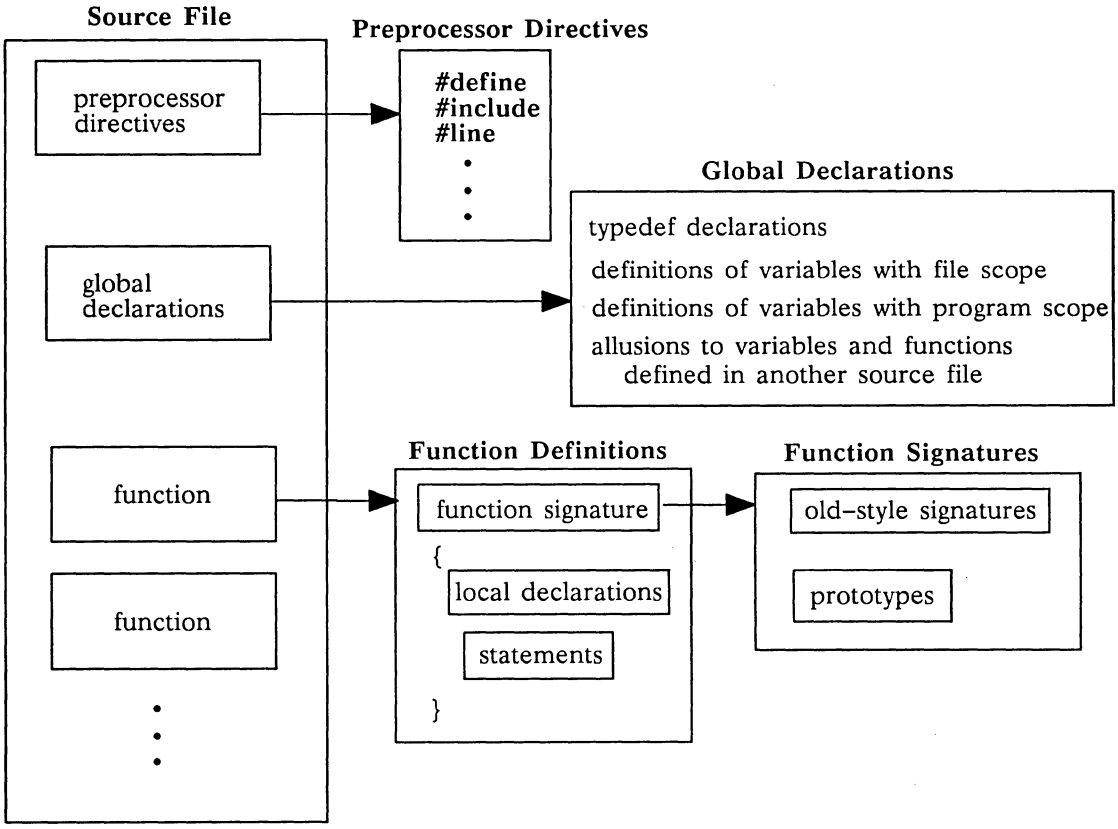


Figure 2–2. Organization of a File of C Source Code

To help illustrate this organization, we provide the following commented program:

```
/* Program name is "file_org_example */
#include <stdio.h>                      /* preprocessor directive */
#define WEIGHTING_FACTOR 0.6            /* preprocessor directive */
typedef float THIRTY_TWO_BIT_REAL;      /* global typedef decl.    */
THIRTY_TWO_BIT_REAL correction_factor = 1.15; /* global variable
                                                * decl. */

float average( float arg1, THIRTY_TWO_BIT_REAL arg2) /* prototype */
{                                       /* start of function body  */
  float mean;                           /* local variable decl.    */
  mean = (arg1 * WEIGHTING_FACTOR) +
         (arg2 * (1.0 - WEIGHTING_FACTOR)); /* assignment stmnt */
   return (mean * correction_factor);     /* return statement */
}                                       /* end of function body */
int main( void )                        /* prototype for main */
{
  float value1, value2, result;    /* local variable declarations */

  printf( "Enter two values -- " );   /* statement */
  scanf( "%f%f", &value1, &value2 );  /* statement */
  result = average( value1, value2 ); /* statement */
  printf( "The weighted average adjusted by a correction factor \
of %4.2f is %5.2f\n", correction_factor, result); /* statement */
}                                       /* end of function */
```

In the following sections, we describe the various components of a C program.

2.3.1 Functions

As shown in both the figure and the example, functions are the primary organizational unit of C. A C program must contain one or more functions.

The function called **main** has a special meaning. The C run-time system uses the first executable statement in **main** as the starting address of the entire program. Consequently, if you do not name one of the functions **main**, the program will have no starting address. Conversely, naming more than one function **main** will cause a compile-time or link-time error.

Unlike some other languages (such as Pascal), which support both procedures and functions, C supports only functions. However, a C function can emulate a Pascal procedure or a Pascal function. In other words, you can declare a C function that either returns or does not return a value to the calling program. See the description of the **void** type in Section 3.6 for more information about functions that behave like procedures.

Every function (**main** or not) adheres to the same rules of organization, and we detail these rules in Chapter 5. For now, we provide an overview.

2.3.2 The Begin and End Symbols: { }

In all structured programming languages it is necessary to mark where a *block* starts and finishes. A block is any logically distinct section of source code. In some languages, marking blocks is accomplished through keywords like **begin** and **end**. In C, you mark the beginning of a block of C code with the { symbol and the end with the } symbol. Because every function must contain at least one block, you need to specify { and } to denote the start and finish of a function.

In addition to delimiting a function, the { and } symbols serve to demarcate blocks in a variety of declarations and statements.

2.3.3 Statements

A function can contain zero or more statements. Chapter 4 describes all the statements that Domain C supports. Note that you cannot put a statement outside of a function.

2.3.4 Preprocessor Directives

Domain C supports a wide variety of preprocessor directives that serve purposes such as controlling conditional compilation, including header files, and defining program constants. Preprocessor directives begin with the # character. Although some preprocessor directives can be placed anywhere in a file, others can only be placed at specific junctures. For complete information on preprocessor directives, see the "Preprocessor Directives" listing in Chapter 4.

2.4 Declarations

With a few rare exceptions, every variable must be declared before it is referenced. A declaration serves to identify the data type and storage class of a variable, and may optionally give the variable an initial value. As Figure 2–2 shows, C supports declarations made both within a block and outside of a block. The position of the declaration affects the storage class of the variable, as explained later in this chapter.

In general, a variable declaration takes the following format:

$$[storage\_class\_specifier]\ [data\_type]\ variable\_name\ [= initial\_value];$$

where:

storage_class_specifier	is an optional keyword that we describe later in Section 3.12.
data_type	is one of the data types described in Chapter 3.
variable_name	is a legal identifier.
initial_value	is an optional initializer for the variable. (We describe variable initialization in Chapter 3.)

For example, here are a few sample variable declarations without storage class identifiers or initial values:

```
int     age;          /* an integer variable named age */
float   ph;           /* a floating-point variable named ph */
char    a_letter;     /* a character variable named a_letter */
int     values[10];   /* an array of 10 integers named values */
enum    days {mon, wed, fri}; /* an enumerated variable named
                               * days */
```

It is legal to omit the data type in certain instances, although it is considered bad practice. You may omit the data type in global declarations and in local declarations that include a storage class specifier. In all of these cases, the data type defaults to **int**. (The proposed ANSI Standard does not support omitting the data type.)

2.4.1 Typedef Declarations

The C language allows you to create your own names for data types with the **typedef** keyword. Syntactically, a typedef is exactly like a variable declaration except that the declaration is preceded by the **typedef** keyword. Semantically, the variable name becomes a synonym for the data type rather than a variable that has memory allocated for it. For example, the statement,

```
typedef long int FOUR_BYTE_INT;
```

makes the name **FOUR_BYTE_INT** synonymous with **long int**. The following two declarations are now identical:

```
long int j;
FOUR_BYTE_INT j;
```

A typedef declaration may appear anywhere a variable declaration may appear and obeys the same scoping rules as a normal declaration. You may not, however, include an initializer with a typedef. Once declared, a typedef name may be used anywhere that the type is allowed (such as in a declaration, cast operation, or **sizeof** operation). By convention, typedef names are written in all uppercase so that they are not confused with variable names.

There are a number of uses for typedefs. They are especially useful for abstracting global types that can be used throughout a program, as shown in the following structure and array declaration:

```
typedef struct {char  month[4];
                int   day;
                int   year;
               } BIRTHDAY;

typedef   char   A_LINE[80];  /* A_LINE is an array of 80
                               * characters */
```

Another use of typedefs is to compensate for differences in C compilers. For example:

```
#if SMALL_COMPUTER
 typedef int SHORTINT;
 typedef long LONGINT;
#else
  #if BIG_COMPUTER
     typedef int LONGINT;
     typdef short SHORTINT;
  #endif
#endif
```

The idea here is that you may be writing code to run on two computers, a small computer where an **int** is two bytes, and a large computer where an **int** is four bytes. Instead of using **short**, **long**, and **int**, you can use **SHORTINT** and **LONGINT** and be assured that **SHORTINT** is two bytes and **LONGINT** is four bytes regardless of the machine.

You can also use typedefs to simplify complex declarations. Consider the following example:

```
typedef float *PTRF, ARRAYF[], FUNCF();
```

This declares three new types called **PTRF** (a pointer to a **float**), **ARRAYF** (an array of **floats**), and **FUNCF** (a function returning a **float**). These typedefs could then be used in declarations such as:

```
PTRF x[5];  /* a 5-element array of pointers to floats */
FUNCF z;    /* A function returning a float */
```

Bug Alert: Using typedefs for Arrays

Be careful about using typedefs for arrays. For example, the following two examples illustrate what can happen when you mix pointers and typedefs that represent arrays. The problem with the program on the left is that **ptr** points to an array of 80 **chars**, rather than a single element of a **char** array. Because of scaling in pointer arithmetic, the increment operator (++) adds 80 bytes, not one byte, to **ptr**.

wrong

```
typedef char STR[80];
STR   string, *ptr;


main()
{
   ptr = string;
   printf("ptr = %d\n", ptr);
   ptr++;
   printf("ptr = %d\n", ptr);
}

*** Run-Time Results ***
ptr = 3997696
ptr = 3997776
```

right

```
typedef char STR[80];
STR   string;
char  *ptr;

main()
{
   ptr = string;
   printf("ptr = %d\n", ptr);
   ptr++;
   printf("ptr = %d\n", ptr);
}

*** Run-Time Results ***
ptr = 3997696
ptr = 3997697
```

2.4.2 Name Spaces

All identifiers (names) in a program fall into one of three **name spaces**. The three name spaces are:

Structure, Union, and Enumeration Tags — Tag names that immediately follow these type specifiers: **struct**, **union**, and **enum**. These types are described in Chapter 3.

Member Names — Names of members of a structure or union.

All Other Names — Any names that are not members of the preceding two classes.

Names in different name spaces never interfere with each other. That is, you can use the same name for an object in each of the three classes without these names affecting one another.

The following example uses the same name, **overuse**, in all three ways (this is an example of name spaces, not of good programming style):

```
int main( void )
{
    int overuse;            /* normal identifier */
    struct overuse          /* tag name */
      { float overuse;      /* member name */
        char *p;
      }
}
```

Note that each struct, union, or enum defines its own name space, so that different types can have the same member names without conflict. The following, for example, is legal:

```
struct A {
            int x;
            float y;
         };
struct B {
            int x;
            float y;
         };
```

The members in struct **A** are distinct from the members in struct **B**. Note that this is consistent with the ANSI standard, although it is an extension to the K&R standard.

Macro names *do* interfere with the other three name spaces. Therefore, when you specify a macro name, do not use this name in one of the other three name spaces. For example, the following program fragment is incorrect because it contains a macro named **square** and a label named **square**:

```
#define square(arg)  arg * arg

int main( void )
{
  .
  .
  .
square: .
        .
        .
}
```

Chapter 3

Data Types and Storage Classes

Every variable and expression has a data type and every function has a return data type. The type determines how the bits are to be interpreted by the computer. This chapter describes all Domain C data types in the following order:

- Integer types (**int, char, short, long, unsigned**)
- Floating-point types (**float, double**)
- Enumerated types (**enum**)
- **void**
- Pointers
- Structures and unions (**struct, union**)
- Arrays

In addition to data type, every variable has a storage class, which defines its scope and duration. The latter half of this chapter describes storage classes.

3.1 Data Type Overview

The C language offers a moderately sized and useful set of data types. There are six different types of integers and two types of floating-point objects. These types—integers and floating-points—are called **arithmetic types**. Together with pointers and enumerated types, they are known as **scalar types** because all of the values lie along a linear *scale*. That is, any scalar value is less than, equal to, or greater than another scalar value of the same type.

In addition to scalar types, there are **aggregate types**, which are built by combining one or more scalar types. Aggregate types, which include arrays, structures, and unions, are useful for organizing logically related variables into physically-adjacent groups. There is also one type—**void**—that is neither scalar nor aggregate. Figure 3-1 shows the logical hierarchy of C data types.

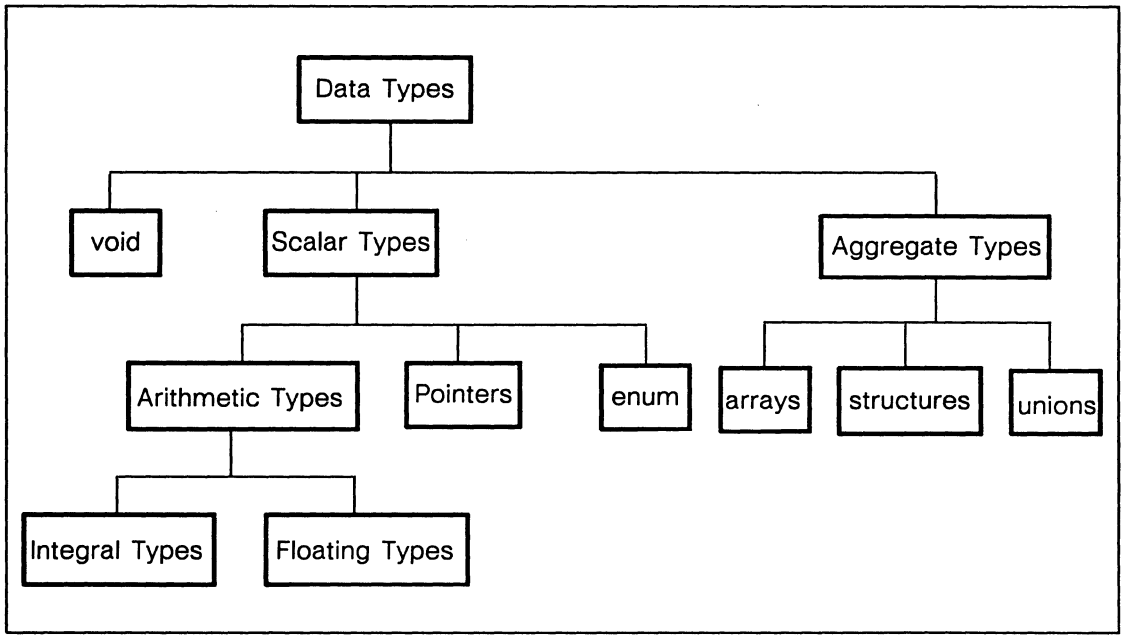


Figure 3-1. Hierarchy of C Data Types

3.1.1 Scalar Types

There are nine reserved words for scalar data types, as shown in Figure 3-2.

char	**long**	**unsigned**
short	**float**	**enum**
int	**double**	

Figure 3-2. Scalar Type Keywords

The types **char**, **int**, **float**, **double**, and **enum** are basic types. The others—**long**, **short**, and **unsigned**—are **qualifiers** that modify a basic type in some way. You can think of the basic types as nouns and the qualifiers as adjectives.

An enumerated variable consists of an ordered group of identifiers. The only value you can assign to an enumerated variable is one of those identifiers. By default, the size of an

enumeration variable is four bytes, but you can explicitly make it two bytes by using the **short** modifier. You can also use **long** to explicitly specify 4-byte enums. Applying **short** and **long** to enums is a Domain extension.

Table 3-1 shows the scalar data types supported by Domain C, their size, and their range of values. Types listed together in a group are synonymous.

Table 3-1. Domain C's Arithmetic Data Types

Data Type	Size (in bytes)	Lowest Possible Value	Highest Possible Value
int long long int	4	-2147483648	+2147483647
unsigned int unsigned long unsigned long int	4	0	4295967295
short short int	2	-32768	+32767
unsigned short unsigned short int	2	0	+65535
char	1	-128	+127
unsigned char	1	0	+255
float	4	$-0.29 * 10^{38}$	$+1.7 * 10^{38}$
double long float	8	$-1.0 * 10^{308}$	$+1.0 * 10^{308}$
short enum	2	-32768	+32767
enum long enum	4	-2147483648	+2147483647
void	none	N/A	N/A
pointers	4	N/A	N/A

3.1.2 Aggregate Types

The following briefly describes the supported aggregate data types:

arrays — An array variable consists of a fixed number of elements of the same data type. The size of an array equals the number of elements times the size of each element.

structures — A structure variable consists of one or more members, each having its own data type. For instance, a structure variable could be composed of two integers and one float. (A structure in C is similar to a fixed record in Pascal.) The size of a structure is the sum of the sizes of all the members, plus possible padding due to alignment rules.

union — A union variable consists of one or more members, each having its own data type. The difference between a structure and a union is that all the members of a structure occupy separate (unique) addresses, but all the members of a union share the same address. (A union in C is similar to a variant record in Pascal.) The size of a union is equal to the size of its largest member.

3.2 Overview of Variable Initialization

C permits you to initialize certain variables when you declare them. Throughout this chapter, we detail variable initialization for specific data types. Here in this section we provide some general guidelines about initialization.

The following variables may *not* be initialized:

- **Automatic** structures, unions, and arrays
- Variables declared with the **extern** keyword

If you do not explicitly initialize a fixed variable, the run-time system initializes it to zero for you. Members of fixed aggregate types not explicitly assigned an initalization value are automatically initialized to zero. Automatic variables do not receive a default initialization. If you do not explicitly initialize them, they will start with unpredictable values.

Fixed variables may be intialized only with **constant expressions** (defined in the "expressions" listing of Chapter 4). Automatic scalar variables may be initialized with either constant or non-constant expressions. If the data type of the initialization expression does not match the data type of the variable, the expression is converted as if a normal assignment were being made. For instance:

```
int global_int = 1;     /* Fixed duration integer initalized to 1 */
int main( void )
{
    float f = 1;       /* Initialization value is converted to 1.0 */
    char char_int = global_int/2;    /* Automatic integer initialized
                                      * to 0 (after conversion).
                                      */
}
```

Scalar initializations may optionally include surrounding braces. That is,

```
int x = 1;
```

is the same as:

```
int x = {1};
```

In practice, however, braces are generally reserved for initialization of aggregate types.

3.2.1 Old-Style Initialization

Some older compilers permit initialization without the equal sign. For example,

```
int x 1;
```

is equivalent to the current:

```
int x = 1;
```

To support programs written for these early C compilers, the Domain C compiler accepts the old-style initialization but issues a warning message. Do not use the old-style syntax for programs you are writing now.

3.3 Integer Data Types

Integers come in three different sizes and can be either signed (the default) or unsigned. With one exception, an integer declaration must include at least one of the type keywords: **unsigned, long, short, int,** or **char.** (The one exception is that a global declaration that does not contain a data type defaults to an **int.**) An integer declaration may also include combinations of these keywords.

To declare an integer variable, simply specify the name of one of the integer data types followed by the variable name. The following examples show all of the possible combinations of integer variables:

```
int a;                  /* signed 32-bit integer */
long int b;             /* same as int in Domain C */
long c;                 /* same as int in Domain C*/

unsigned int d;         /* unsigned 32-bit integer */
unsigned e;             /* same as unsigned int */
unsigned long int f;    /* same as unsigned int in Domain C */
unsigned long g;        /* same as unsigned int in Domain C */

short h;                /* signed 16-bit integer */
short int i;            /* same as short */

unsigned short j;       /* unsigned 16-bit integer */
unsigned short int k;   /* same as unsigned short */

char m;                 /* signed 8-bit integer in Domain C */
unsigned char n;        /* unsigned 8-bit integer */
```

The sizes of integer types are implementation-dependent. The K&R and ANSI standards only require that a **short** be no larger than an **int**, and an **int** be no larger than a **long**. Programs that depend on **int**s being 32 bits long, for example, may not be portable.

3.3.1 32-Bit Integers

You declare a signed 32-bit integer by specifying one of the following three data types:

- **int**
- **long int**
- **long**

Such variables can hold any integral value from -2147483648 (-0x80000000) through 214748367 (0x7FFFFFFF) inclusive.

You declare an unsigned 32–bit integer with any of the following data types:

- **unsigned int**
- **unsigned**
- **unsigned long int**
- **unsigned long**

Unsigned 32–bit variables hold values from 0 through 4295967295.

The Domain system stores 32–bit integers in four contiguous bytes as illustrated in Figure 3–3. The most significant bit in the integer is bit 31; the least significant bit is bit 0. For signed 32–bit integers, bit 31 holds the sign bit. Negative signed integers are stored in two's–complement form.

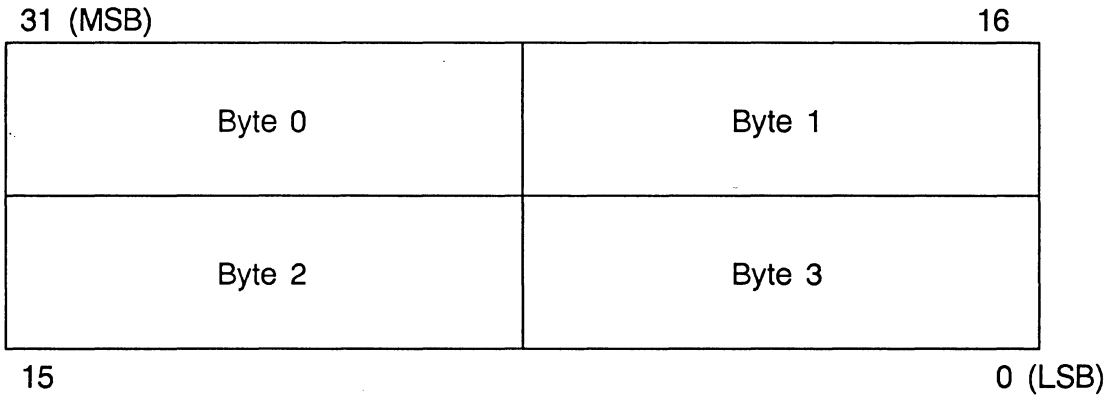


Figure 3–3. 32–Bit Integer Format

3.3.2 16–Bit Integers

You declare a signed 16-bit integer by specifying either of the following two data types:

- **short**
- **short int**

16–bit signed integer variables can hold any integral value from –32768 through +32767 inclusive.

You declare an unsigned 16-bit integer by specifying either of these two data types:

- **unsigned short**
- **unsigned short int**

Unsigned 16-bit variables can hold any value from 0 through 65535.

The Domain/OS system stores 16-bit integers in two contiguous bytes as illustrated in Figure 3-4. The most significant bit is bit 15; the least significant bit is bit 0. Negative signed integers are stored in two's-complement form. For signed 16-bit integers, bit 15 holds the sign bit.

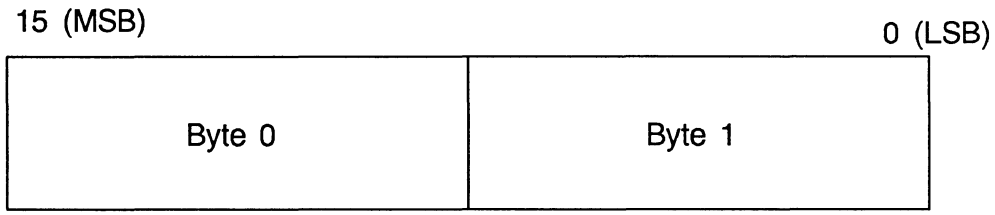


Figure 3-4. 16-Bit Integer Format

3.3.3 8-Bit Integers (Character Data Type)

In C, the distinction between characters and numbers is blurred. There is a data type called **char**, but it is really just a 1-byte integer value that can be used to hold either characters or numbers.

Domain C supports two kinds of character data types—**char** and **unsigned char**. The **char** data type holds signed 8-bit quantities ranging from -127 through +128. The **unsigned char** data type holds unsigned 8-bit quantities ranging from 0 through 255. Since the ASCII values of characters range from 0 to 127, you can use either data type to hold keyboard characters.

Here are two sample character variable definitions:

```
         char  c1;
unsigned char  c2;
```

After declaring **c1** as a **char**, you can make either of the following assignments:

```
c1 = 'A';
c1 = 65;
```

In both cases, the decimal value 65 is loaded into the variable **c1** since 65 is the ASCII code for the letter 'A'. Note that character constants are enclosed in single quotes. The

quotes tell the compiler to get the numeric code value of the character. For instance, in the following example, **a** gets the value 5, whereas **b** gets the value 53 since that is the ASCII code for the character '5'.

```
char a , b;
a = 5;
b = '5';
```

Figure 3-5 shows how the Domain/OS system stores character variables. If the variable is an **unsigned char**, then bit 7 contains the most significant bit (MSB), and bit 0 contains the least significant bit (LSB). If the variable is a **char**, then bit 7 contains the sign bit, and bit 0 contains the least significant bit. **char** variables with a negative value are stored in two's-complement form.

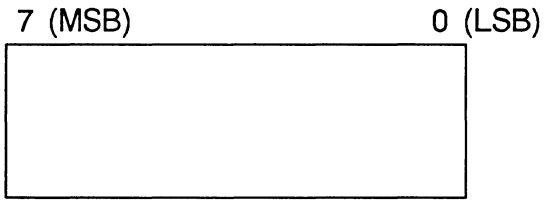


Figure 3-5. Internal Representation of Character Variables

3.3.4 Initializing Integer Variables

You may initialize integer variables with integer or floating-point values. If the initialization expression is a floating-point value, it is converted to an integer before being assigned. If the variable has fixed duration, the initializer must be a constant expression. Here are a few sample initializations:

```
{
  int                x = 50000;
  short int          y = x/2;
  unsigned long int  z = x*y;
  static int         xx = 1.5;   /* converted to 1 */
  char               yy = -20;
  unsigned char      zz = 200;
```

See Section 3.2 for details on how storage class affects initialization. See Section 4.3 for information about assignment conversions.

You can initialize character variables with integer or floating-point expressions. All of the following, for example, are legal:

```
char  zebra = 'g'; /* a character enclosed in single quotes. */
char  zebra = 103; /* a small integer */
char  zebra = 0147;/* an octal integer */
char  zebra = '\147' /* a small integer preceded by a backslash
                     * and enclosed in single quotes
                     */
```

Interestingly, all four formats produce the same results. The character constant 'g' causes the compiler to initialize **zebra** with the ASCII value of the letter **g**, which happens to be 103. By specifying the decimal integer value 103, we accomplish the same thing. The octal value 147 is also equal to 103. Finally, by preceding 147 with a backslash and enclosing it in single quotes, we tell the compiler to treat it as an octal number.

3.3.5 Integer Overflow

An overflow condition occurs whenever a value is too large to be represented in the bits allocated for it. Overflow for expressions containing unsigned objects is explicitly defined by the K&R and ANSI standards. Overflow for signed expressions, however, is implementation-dependent. Domain C handles both cases identically.

When the Domain/OS system identifies an overflow condition, it truncates the most significant bits (including the sign bit). When performing an operation on signed integers, an overflow condition may cause an unexpected change of sign in the answer. When performing an operation on unsigned integers, you can spot an overflow by recognizing an answer that is much smaller than anticipated.

Consider the following example:

```
/* Program name is "int_overflow_example" */
#include <stdio.h>

int main( void )
{
            short x = 0xFFF0;
  unsigned short y = 0xFFF0;

  printf( "x = %hd\n", x );
  printf( "y = %hd\n", y );
}
```

The results are:

```
x = -16
y = 65520
```

In both cases, the same bit pattern results:

```
11111111 11110000
```

However, **x** is interpreted as a negative number whereas **y** is interpreted as a positive value.

3.4 Floating-Point Data Types

Domain C supports three types, **float**, **long float** and **double**, for representing floating-point values. The **float** type is a single-precision floating-point type and the **double** type is double-precision. The **long float** type is a synonym for **double** (**long float** is an extension to the ANSI and K&R standards). You may *not* use the **unsigned** qualifier in a floating-point declaration. Here are a few sample declarations:

```
      float   tiger;
      double  giraffe;
long  float   elephant;
```

3.4.1 Single-Precision Floating-Point

Single-precision floating-point numbers (type **float**) occupy four contiguous bytes, as shown in Figure 3-6. The range of a **float** is approximately $-.29*10^{38}$ through $1.7*10^{38}$. It is accurate to approximately seven digits.

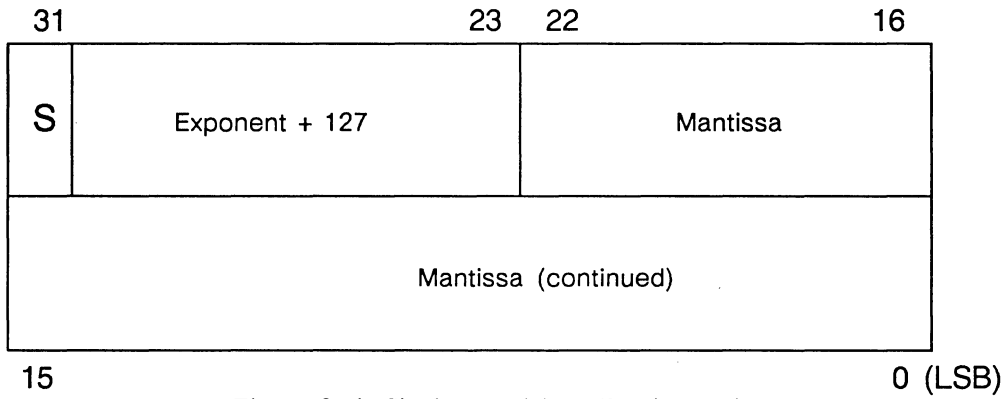


Figure 3-6. Single-Precision Floating-Point Format

The first bit (bit 31) is the sign bit. The sign bit is set (S=1) to denote a negative number, and clear (S=0) to denote a positive number. The next eight bits contain the exponent plus 127. The following 23 bits contain the mantissa of the number without the leading 1. (The mantissa is stored in magnitude, not two's-complement, form.)

The following example shows how Domain/OS stores the floating-point value +100.5. The four bytes contain the bit pattern shown in Figure 3-7.

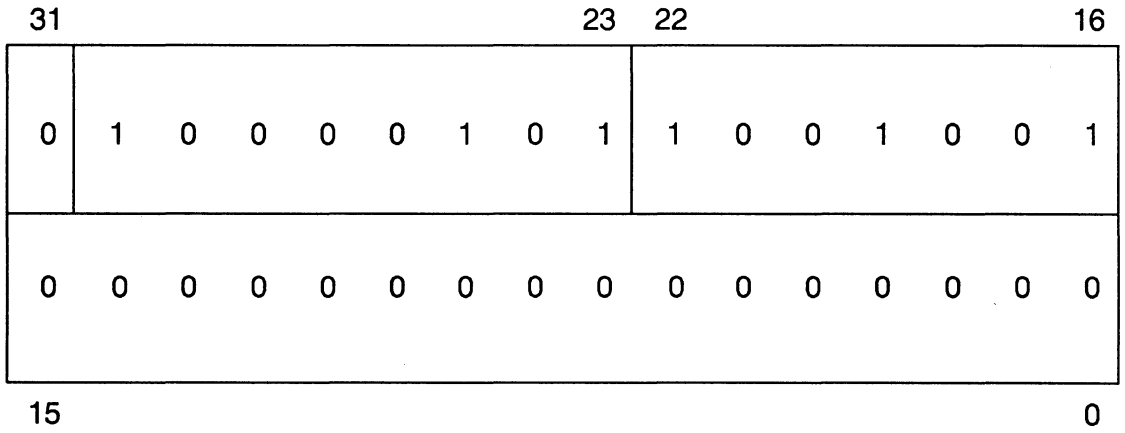


Figure 3–7. Internal Representation of +100.5

Breaking up the number into sign, exponent, and mantissa gives us the following information:

```
sign                          -- 0 (positive)
exponent                      -- 10000101 (133 in decimal)
significant part of mantissa -- 1001001
```

The exponent is 133, and 133 is equal to 127 plus 6. Therefore, we view the mantissa bits as follows:

```
bit 22 represents          2^5  * 1
bit 21 represents          2^4  * 0
bit 20 represents          2^3  * 0
bit 19 represents          2^2  * 1
bit 18 represents          2^1  * 0
bit 17 represents          2^0  * 0
bit 16 represents          2^-1 * 1
```

The quantity 100.5 is equal to $(2^6 + 2^5 + 2^2 + 2^{-1})$

3.4.2 Double–Precision Floating–Point

Double–precision floating–point numbers (type **double** and **long float**) are represented in eight bytes (64 bits). Figure 3–8 illustrates the format. A **double** has a range of approximately -10^{308} to 10^{308} and is accurate to approximately 16 decimal digits.

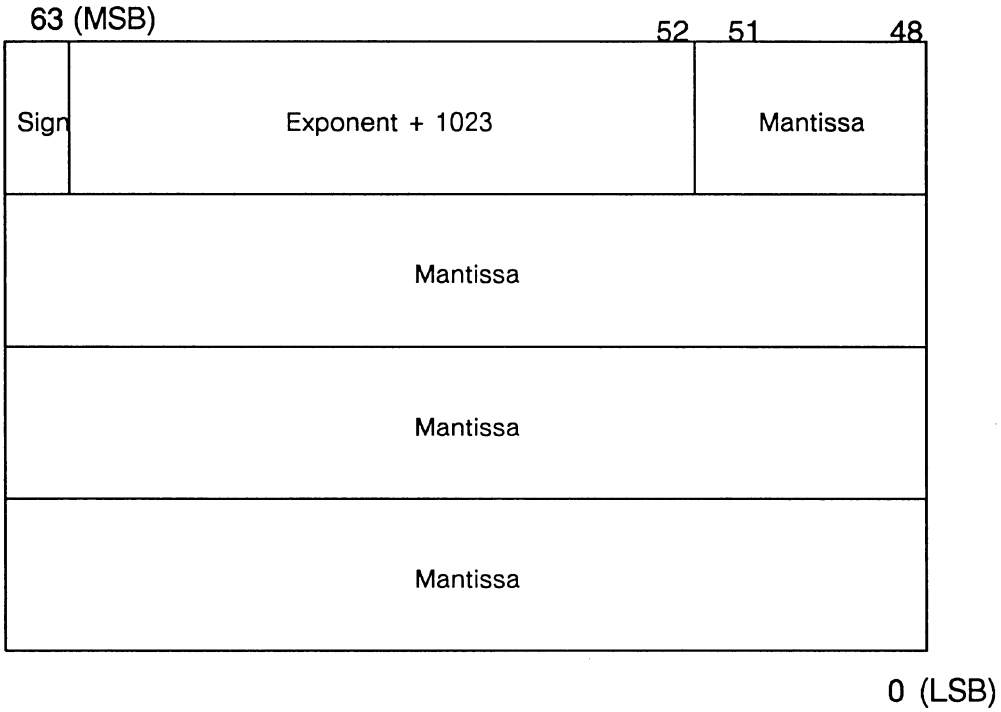


Figure 3–8. Double–Precision Floating–Point Format

The first bit of the first word is the sign bit. The next 11 bits contain the exponent plus 1023. The remaining 52 bits hold the mantissa without the leading 1. (The mantissa is stored in magnitude form, not in two's–complement form.)

3.4.3 Initializing Floating–Point Variables

You may initialize floating–point variables with either integer or floating–point data. The data is converted to the variable's type as if a normal assignment were being made. For example:

```
float     guava = 3.2;
double    pi    = 3.1415926535;
float     z     = 5;
```

3.5 Enumerated Data Types

An enumerated data type consists of an ordered group of identifiers. Enumeration types are particularly useful when you want to create a unique set of values that may be associated with a variable. The compiler reports a warning if you attempt to assign a value that's not part of the declared set of legal values to an enum variable. The possible formats of enumerated declarations are as follows:

enum [*tag_name*] { *id1* [=*val*] [{, *idN* [=*val*]}] }

enum *tag_name variable_name1* [{,*variable_nameN*}]

enum [*tag_name*] { *id1* [=*val*] [{, *idN* [=*val*]}] } *variable_name1* [{,*variable_nameN*}]

That is, to declare an enumerated variable, you must specify the keyword **enum** followed by an optional *tag_name*. The *tag_name* is not the name of the variable; rather it is the name of the enumerated type that you are declaring. After the optional *tag_name*, you optionally specify a list of identifiers separated by commas. This list of identifiers must be enclosed in braces. Each identifier may be followed by an optional constant expression that assigns a value to the enumeration constant. If no value is specified, the enumeration constant is assigned a value one greater than the value assigned to the previous enumeration constant in the list. If no values are specified for the entire list, the numbering begins at zero. Following the optional list of identifiers, you can optionally specify one or more *variable_names*. A tag name cannot be used by itself; it must be preceded by the keyword **enum**; for example, compare the right and wrong ways to use the tag name forest:

```
enum forest {maple, pine, fir} nordic;
     forest  southern; /* wrong */
enum forest  alpine;   /* right */
```

Here are five sample enumerated declarations:

```
/* These two declarations have a tag name and a variable name.    */
  enum citrus {lemon, lime, orange, carambola, grapefruit} c_fruits;
  enum beatles {John, Paul, George, Ringo} beatles_members;

/* This declaration has a tag name and two variable names. */
  enum color { red , blue , yellow }  used, not_used;

/* This declaration has a variable name, but no tag name. */
  enum {one, two, three}  cardinal_numbers;

/* This declaration has a tag name, but no variable name. */
  enum ordinal_numbers {first, second, third};
```

Consider the third declaration. It declares an enumerated type called **color** with possible values of **red**, **blue**, and **yellow**. Two variables, **used** and **not_used**, are defined to have this type. Therefore, variables **used** and **not_used** can have the values **red**, **blue**, or **yellow**. For example, you can make these assignments

```
used = red;
not_used = yellow;
used = not_used;
```

but you cannot make this assignment:

```
used = orange;    /* ILLEGAL: orange is not a value of color */
```

Because enumeration types are stored as integers, it is possible to assign integer values to an enumeration variable. However, the Domain C compiler will issue a warning message when it encounters such usages. For example, the assignment,

```
not_used = 5;
```

would produce the following warning message:

```
******** Line 6: [Warning #205]  Enumeration type clash [not_used,
5] to the = operator.
```

You can avoid this warning by casting the integer expression to the enumeration type:

```
not_used = (enum color) 5;
```

For details on how you can use enumerated variables within statements, see the "enumerated operations" listing in Chapter 4.

3.5.1 The Values of Enumerated Constants

Enumerated constants are the list of possible identifiers that an identifier can have. For example, the enumerated constants for variables **used** and **not_used** are **red**, **blue**, and **yellow**. The Domain C compiler automatically associates an integer value with each enumerated constant. By default, the integer values of enumerated constants start at zero and increment by one with each constant. For example, in the declaration of tag name **color**, the compiler assigns **red=0**, **blue=1** and **yellow=2**. Therefore:

```
(yellow > red)       /* evaluates to 1 (true)  */
(yellow == red)      /* evaluates to 0 (false) */
```

You can override this numbering scheme by explicitly assigning a number to one or more enumerated constants. For instance, the following initializations

```
enum fruits {apple=3, pear=1, orange, banana, melon=(-1)};
```

result in the following integer representations:

```
apple   =   3
pear    =   1
orange  =   2
banana  =   3
melon   =  -1
```

You can specify the values in any order and you do not have to supply consecutive integer values. If you do not explicitly assign an integer value, the system assigns a value by adding one to the previous constant's value. In our example, this means that both **apple** and **banana** have a value of 3. This is perfectly legal and means, in effect, that **apple** and **banana** are synonyms.

Some compilers allow previously defined enum constants to be used in the initializing expression, as in:

```
enum vegetables {carrots=1, celery=carrots+2};
```

However, the Domain C compiler does not allow this syntax.

Since enumerated constants have an explicit or implicit value, you can use an enumerated constant in place of an integer to subscript an array. For example:

```
{
  enum {part_number=0, order_number, quantity} num;
  int part[1000][2];

  part[0][part_number] = 1357;     /* assign part[0][0] */
  part[0][order_number] = 22567;   /* assign part[0][1] */
  part[0][quantity] = 370;         /* assign part[0][2] */
       .
       .
       .
}
```

3.5.2 Initializing Enumerated Variables

You can initialize an enumerated variable when you define it; for example:

```
enum citrus { lemon, lime, orange, carambola, grapefruit }
              c_fruits = lime;
enum beatles {John, Paul, George, Ringo} beatles_members = John;
enum eurofrancophones {France, Suisse, Belgique} la_langue =
                                                        Belgique;
enum color { red , blue , yellow }  used = red, not_used = yellow;
```

If the enumerated type has dynamic duration, it may also be initialized by a previously declared variable with the same enumerated type. The following lines, for instance, initialize **color** to **blue**, **hue** to **red**, and **shade** to **red**:

```
{
static enum rainbow {red, blue, green} color = blue, hue = red;
enum rainbow shade = hue;  /* Automatic variable initialized
                            * with previously declared
                            * variable.
                            */
}
```

3.5.3 Sized enums — Domain Extension

By default, the Domain C compiler allocates four bytes for all enumeration variables. However, if you know that the range values being assigned to an enum variable is small, you can direct the compiler to allocate only two bytes by using the **short** type specifier. You can also use the **long** type specifier to indicate four-byte enums even though this is the default. For example:

```
enum default_enum { ERR1, ERR2, ERR3, ERR4 }; /* four-byte enum
                                               * type */
long big_enum { ST0, ST1, ST2, ST3 }; /* four-byte enum type */
short enum small_enum { cats, dogs }; /* two-byte enum type */
```

When mixed in expressions, enums behave exactly like their similarly sized integer counterparts. That is, an **enum** behaves like an **int**, a **long enum** acts like a **long int**, and a **short enum** acts like a **short int**. Note, however, that you will receive a warning message when you mix enum variables or constants with integer or floating-point types, or with differently typed enums.

3.6 The void Data Type

Domain C supports the **void** data type, which has become a common feature of modern C compilers. The **void** type is not a data type in the traditional sense. You cannot declare a simple variable as being **void**; for instance, a declaration like the following will cause an error:

```
void   x;
```

The **void** data type has three important purposes. The first is to indicate that a function does not return a value. For instance, you can write a function definition such as:

```
void func( a, b )
int a, b;
{
    .
    .
}
```

This indicates that the function does not return any useful value. Likewise, on the calling side, you would declare **func()** as:

```
extern void func();
```

This informs the compiler that any attempt to use the returned value from **func()** is a mistake and should be flagged as an error. For example, you could invoke **func()** as follows:

```
func( x, y );
```

But you cannot assign the returned value to a variable:

```
num = func( x, y );  /*  This should produce an
                      *  error
                      */
```

In situations where the function returns an actual value that you want to ignore, you can use **void** in a cast operation. In the following example, for instance, function **print_line_rtn** returns an integer error code, but we explicitly discard the returned value through a cast:

```
/* Program name is "void_example2" */
#include <stdio.h>
#include <string.h>

int print_line( char *string )
{
  if (strlen( string ) > 80)    /* If line is too long, return
                                 * error */
    return -1;
  else
  {
    printf("%s\n",string);
    return 1;
  }
}

int main( void )
{
  char *string = "This is an example of a void function";

  (void) print_line( string );
}
```

In the preceding example, the **void** cast is not required since the context makes it clear that the value returned by the function should be discarded. Nevertheless, the **void** cast enables you to make this explicit. You cannot use in any way an object that has been cast to **void**. That is, you cannot cast it to another type, you cannot pass it as an argument, and you cannot assign it to a variable.

Another purpose of **void** is to declare a function that takes no arguments. This is described in Section 5.4, which discusses prototypes.

Finally, the **void** type allows you to create **generic pointers**, as described in Section 3.7.3.

3.7 Pointer Data Types

The C language allows you to create a pointer to an object of any type. To declare a pointer variable, precede the pointer variable name with an asterisk (*). The following statements show some examples of pointer declarations.

```
int *ip;        /* ip is a pointer to an int. */
char *chp;      /* chp is a pointer to a char. */
char *cp[];     /* cp is an array of pointers to chars. */
float *fp();    /* fp is a function that returns a pointer
                 *  to a float.
                 */
float (*pfp)();/* pfp is a pointer to a funtion that returns a
                * float.
                */
short **cpp;    /* cpp is a pointer to a pointer to a short. */
```

In the fifth declaration, we need to use parentheses to achieve correct binding. The rules for composing complex declarations such as this one are described in Section 3.11.

For details on using pointers in the action part of your program, see the "pointer operations" listing of Chapter 4.

3.7.1 Internal Representation of Pointers

Domain C stores pointers in the 32–bit structure shown in Figure 3–9.

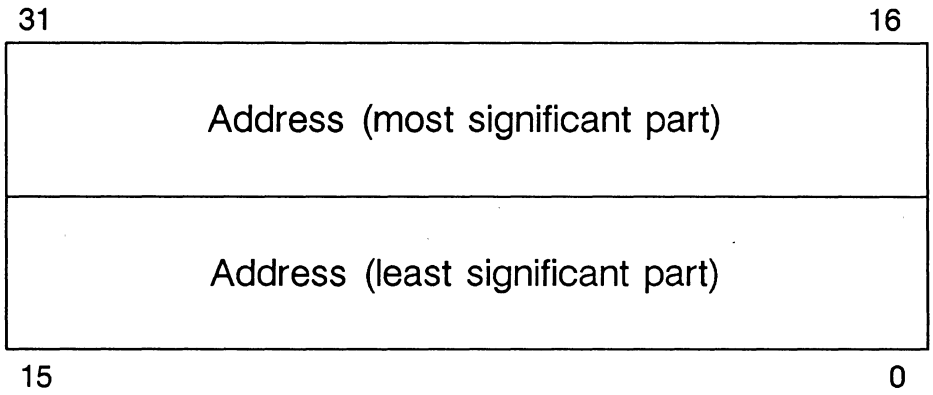


Figure 3–9. Pointer Variable Format

3.7.2 Initializing Pointers

You can initialize pointer variables with pointer expressions or with the constant zero (0). If the pointer variable has automatic duration, any pointer expression is legal. If the variable has fixed duration, the expression must be a pointer constant. Initialization with zero produces a **null pointer**. Due to dynamic conversions, it is possible to initialize a pointer with a function name, array name, string constant, or address of an object. The following examples show a variety of ways to initialize pointers.

```
float *null_point = 0;              /* Null Pointer */
int i, *pi = &i;                    /* Address of i */
static char *string="string";       /* Pointer to "string" */
float array[5], *pa = array;        /* Pointer to beginning of array
*/
float *pa1 = array+2;       /* Pointer to third element of array */

extern void f();            /* Define a function named f */
void (*pf)() = f;           /* Initialize pf to point to f */
int *p_absolute = (int *) 0xFFAABB12;   /* Pointer to absolute
                                         * address
                                         */
```

3.7.3 Generic Pointers

In accordance with the ANSI standard, the Domain C compiler now allows you to create a **generic pointer** variable by declaring a pointer to **void**:

```
void *genp;  /* genp is a generic pointer */
```

A generic pointer can be cast to any other pointer type. Moreover, a generic pointer is implicitly converted to the destination type when it is assigned a pointer value or is assigned to a pointer variable. When a generic pointer is compared to a pointer of another type, it is implicitly converted to the other pointer type. For example:

```
char *cp;
float *fp;
void * genp;

genp = cp;  /* genp is implicitly converted to pointer to char.
             */
fp = genp;  /* genp is implicitly converted to pointer to
             * float.
             */
if (cp == genp)  /* genp is implicitly converted to pointer to
                  * char.
                  */
```

It is illegal to dereference a generic pointer without first casting it to a valid pointer type.

```
float f = 2.0;
void *genp;

genp = &f;          /* ok */
f = *genp;          /* ILLEGAL */
f = *(float *)genp; /* ok */
```

Generic pointers are particularly useful for functions that can return pointers to different types of objects. The classic example is **malloc()**, which dynamically allocates memory for

different types of objects. Traditionally, **malloc()** returns a pointer to **char**, which must then be cast to the appropriate pointer type. For example:

```
struct S {
           char str[10];
           int val;
         }
int main( void )
{
  extern char *malloc();
  struct S *ps;

/*  cast returned value to pointer to struct S.  */
  ps = (struct S *) malloc( sizeof(struct S) );
}
```

By redefining **malloc()** to return a pointer to **void** rather than a pointer to **char**, you can avoid casting the returned value because it will be implicitly converted:

```
struct S {
           char str[10];
           int val;
         }
int main( void )
{
  extern void *malloc();
  struct S *ps;

/*  returned value is implicitly converted to type of ps. */
  ps = malloc( sizeof(struct S) );
}
```

3.8 Structure and Union Data Types

Because structures and unions obey most of the same syntactic rules, we describe them together.

A **structure** is an object that contains other objects. It is similar to a fixed record in Pascal. The objects within a structure, called **members** or **components**, are usually named and can be of any data type, including other structures, unions, or arrays. For instance, a structure might contain an **int**, a **float**, and a **char** as members. A **bit field** is a special member that takes up from 1 to 32 bits of memory.

A **union** is similar to a structure, but instead of holding all of the members at once, it can hold only one at a time because each member has its storage allocated at the same address. It is similar to a variant record in Pascal. The compiler makes sure that enough space is allocated to hold the largest member.

For details on using structures and unions in statements, see the "structure and union operations" listing in Chapter 4.

3.8.1 Declaring a Structure or Union

The only difference between declaring a structure and a union is in the keywords **struct** and **union.**

There are four basic types of structure and union declarations:

1. No **tag name**—If you do not specify a tag name, you should declare at least one variable. For instance, the following declares a structure variable called **struct_example,** which is a structure with three members:

```
struct { int member_one;
         float member_two;
         char member_three;
       } struct_example;
```

2. Tag name and member declaration(s), but no variable name(s)—This defines a name that can be used in place of the full structure specification in future declarations. For instance, after declaring

```
struct S1 {int i; float f;};
```

   you can declare:

```
struct S1 x,y;
```

   which declares **x** and **y** to be structures containing an **int** member named **i** and a **float** member named **f.**

3. Tag name, member declaration(s), and variable name(s)—This type of declaration serves two purposes: it defines a tag name that can be used in subsequent declarations, and it declares specific variables. For example,

```
union U { char ch[8];
          int i;
        } u1, u2, u3;
```

   defines a type called **U,** and three variables—**u1, u2,** and **u3**—that have this type.

4. Tag name and variable name(s), but no member declarations—This form of declaration may only be used if you have already defined the tag name. For example, after making the preceding declaration, we could write:

```
union U u4;
```

This would define another variable, **u4**, with type **U**. Note that you cannot use the tag name by itself; it must be preceded by the keyword **union** or **struct**.

Tag names and member names are distinct from each other and from variable names so that a tag and a variable and a member may all have the same name without a conflict arising. The following, for example, is a legal declaration:

```
struct x { int x;};
char x;
```

The compiler will not confuse the three **x**'s: their usage in the code makes it clear which one is being referenced.

A structure or union may not contain instances of itself, but it may contain pointers to itself. For example:

```
struct S { int a,b;
           float c;
/*         struct S d;      THIS IS ILLEGAL! */
           struct S *d;     /* This is legal */
         };
```

It is possible to create structures and unions that reference each other as shown in the following example:

```
union U1 { int a;
           union U2 *b;
         };

union U2 { int a;
           union U1 *b;
         };
```

Each union contains an integer as the first component and a pointer to the other union as the second component. Note that it is possible to declare a pointer to **U2** before **U2** is ever declared. This is one of the few situations in the C language where you may use an identifier before it has been declared.

3.8.2 Internal Representation of Structures

Each member of a structure takes up the same amount of space that it would require if it were an unattached variable rather than a member of a structure. For instance, an **int** requires 32 bits whether it is used as a scalar variable or used as a member of a structure. The boundary alignment rules are somewhat different, however, as explained in the next section.

3.8.2.1 Alignment of Structure Members

The **alignment** of an object identifies the set of legal addresses at which that object can be allocated. Objects that are **byte-aligned** can be allocated anywhere; objects that are

word-aligned can only be allocated at even addresses; objects that are **longword-aligned** can only be allocated at addresses that are evenly divisible by four.

Natural alignment means that an object's address is evenly divisible by its size. For example, a naturally aligned 4-byte object begins at an address that is evenly divisible by 4, and a naturally aligned 8-byte object begins at an address that is evenly divisible by 8. In general, natural alignment produces faster executable code, although the efficiency savings vary a great deal from one processor to another. Code for the 68000 family of processors runs slightly faster if objects are naturally aligned.

By default, all scalar objects are naturally aligned. The rules for structures and unions, and for structure and union members, however, are somewhat different. This section describes the default rules.

> **NOTE:** The rules described in this section do not apply to bit fields. See Section 3.8.4 for information about the alignment of bit fields.

Alignment rules affect two properties of structures:

- How members are laid out in the record (that is, whether padding is inserted between members).
- How memory for the entire record is allocated.

3.8.2.2 Layout of Structure Members

The compiler lays out structure members based on **word alignment** rules. According to **word alignment** rules, all objects longer than a byte must be aligned on shortword boundaries (even addresses). **char**s may be aligned on odd or even addresses.

As illustrated in the following examples, the default alignment rules can produce **padding** (also called "holes" or "gaps") in a structure, but the padding is never larger than one byte. Consider the following structure:

```
typedef struct { long int a;
                 char b;
                 short c;
               } S1;
```

Figure 3-10 shows how the members are laid out. Note that there is a byte of padding inserted after **b** to ensure that **c** is aligned on a word boundary.

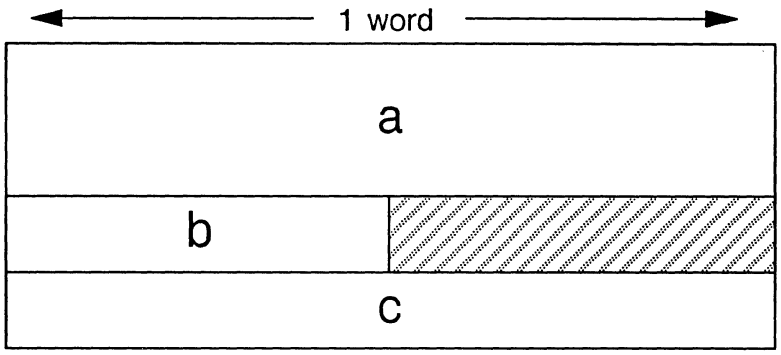


Figure 3–10. Default Layout of Structure S1

The total size of a structure must be an even multiple of two bytes. This rule can result in padding at the end of a structure. (This rule also means that the smallest possible structure is 16 bits.) Figure 3–11 shows the layout of a structure that contains a gap in the middle and a gap at the end as a result of the default alignment rules.

```
typedef struct { char c1;
                 short s1;
                 char c2;
               } S2;
```

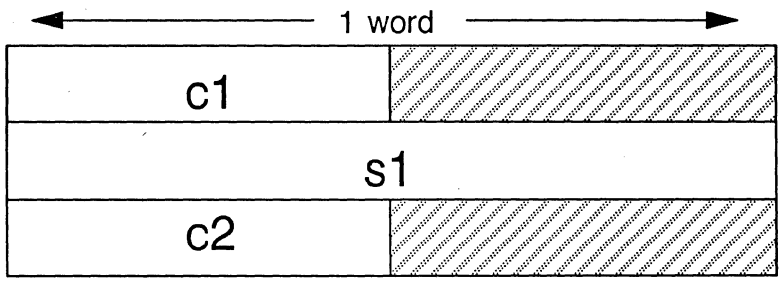


Figure 3–11. Layout of Structure S2

3.8.2.3 Memory Allocation of Structures

Structures are always allocated on even addresses (word boundaries). In addition, they may be allocated on even larger boundaries if that allocation will produce natural alignment for some of the structure's members. The actual algorithm used by the compiler to decide how to allocate structures is somewhat complex. The general steps are as follows:

1. As the compiler lays out members, it assumes that the starting address of the structure is zero.

2. The compiler then notes which members are naturally aligned.

3. After all the members have been laid out, the compiler looks for the largest member that *is* naturally aligned. The compiler then allocates the entire structure on a boundary that matches the natural alignment for this member.

These rules will be clearer if we show how they work for a couple of examples. Consider the following structure type:

```
typedef struct { float a;
                 char b;
                 short c;
               } S3;
```

The layout for this structure is shown in Figure 3–12.

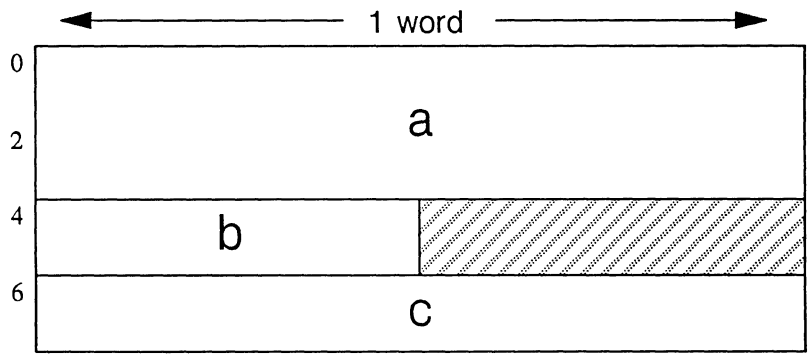


Figure 3–12. Naturally Aligned Structure S3 with 1–byte Padding

The compiler lays out the members according to word alignment rules, and assumes that the structure begins at address zero. For this structure, the alignment rules produce a layout in which all elements are naturally aligned. (Any member that starts at address zero is naturally aligned.) The compiler then searches for the largest member that is naturally aligned, which is **a**. The natural alignment of **a** is longword; therefore, structures of type **S3** will be allocated on longword boundaries.

Consider a second example:

```
typedef struct { short a;
                 float b;
               } S4;
```

The layout is shown in Figure 3–13. In this case, **a** is naturally aligned, but **b** is not naturally aligned (because the address 2 is not evenly divisible by **b**'s size, which is 4). Therefore, the compiler uses the natural alignment of **a** (word alignment) to allocate structures of type **S4**.

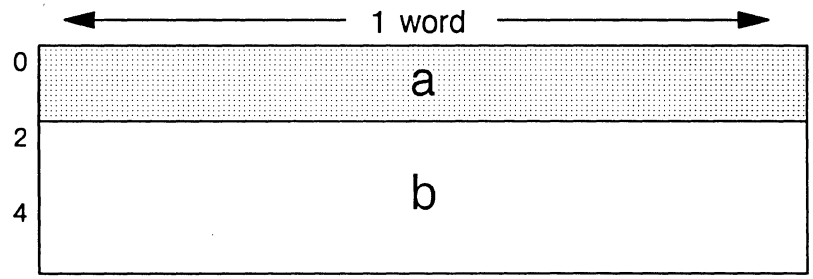


Figure 3–13. Layout of S2 Using Word Alignment Rules

You can usually guarantee that all members of a structure will be naturally aligned by arranging the members in descending order of size. This technique will always work if all the members are scalar objects. This technique may not work if one or more of the structure members is an aggregate. Arranging members in decreasing order of size also guarantees that there will be no padding between structure members. (There might still be a byte of padding at the end of the structure to make it an even number of bytes.)

In some instances, a structure that would normally be allocated on a longword or quadword boundary receives a different allocation because the structure is part of a larger aggregate type (such as a structure or array). For example, consider the declaration of **S1**:

```
typedef struct { long int x;
                 short y;
               } S1;
```

The compiler can guarantee that an individual structure of type **S1** will be allocated on a longword boundary (so that **x** and **y** will be naturally aligned), but if you declare an array of **S1** structures, only half of them will be aligned on longword boundaries

```
S1 a_of_S1[3];
```

Figure 3–14 shows the layout of an array of three **S1** structures. Note that the second element is aligned on a shortword boundary, not a longword boundary.

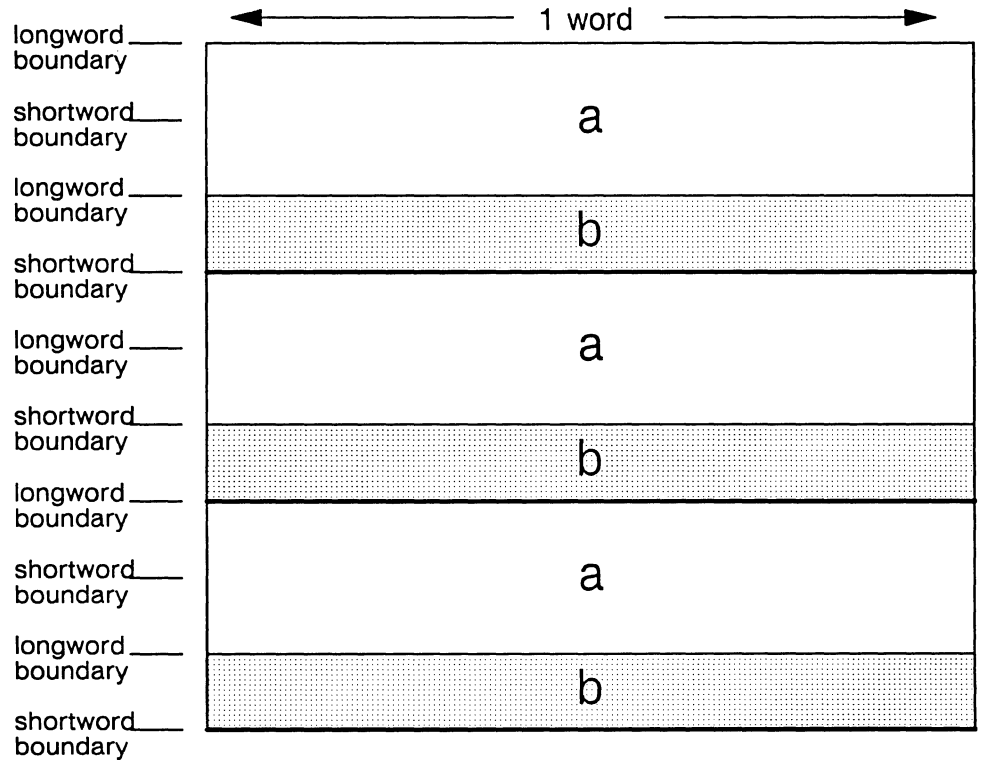


Figure 3–14. Array of S1 Structures, Not Naturally Aligned

To ensure that all elements of **a_of_S1[]** are naturally aligned, you would need to insert an additional word of padding at the end of **S1**. You could do this explicitly, as shown in the following declaration:

```
typedef struct { long int x;
                 short y;
                 short padding;
               } S1;
```

3.8.3 Internal Representation of Unions

Unions are similar to structures except that the members are overlaid one on top of another, so members share the same memory. For example, the following declaration results in the storage shown in Figure 3–15.

```
typedef union
{
  struct
  {
    char c1, c2;
  } s;
  long j;
  float x;
} U;

U example;
```

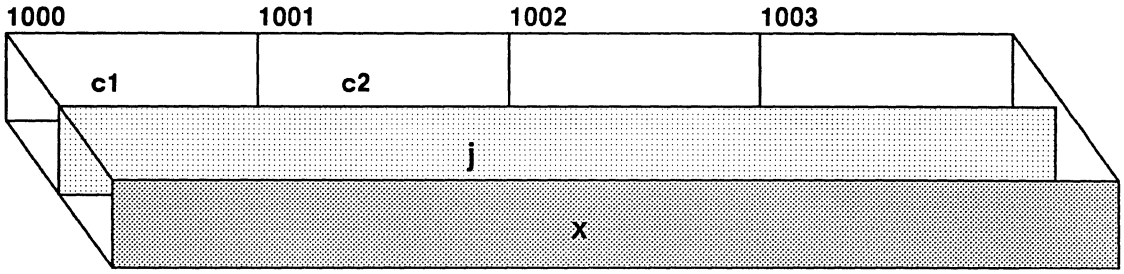


Figure 3–15. Example of Union Memory Storage

The compiler always allocates enough memory to hold the largest member and all members begin at the same address. The union is aligned so that the largest member is naturally aligned. The data stored in a union depends on which union member you use. For example, the assignments,

```
example.s.c1 = 'a';
example.s.c2 = 'b';
```

would result in the storage shown in Figure 3–16.

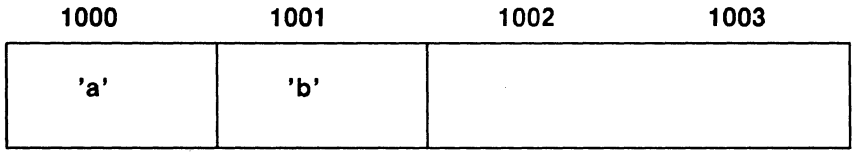


Figure 3–16. Storage in Union example *After Assignment*

But if you make the assignment,

```
example.j = 5;
```

it would overwrite the two characters, using all four bytes to store the integer value 5.

3.8.4 Bit Fields in Structures and Unions

Structures and unions can contain members known as **bit fields** that consist of a specified number of bits. Bit fields allow you to name groups of 1 to 32 bits. Bit fields are a useful construct when space is at a premium, or when you need to map an object onto a predefined structure, such as a device register.

The syntax for declaring a bit field is shown in Figure 3–17.

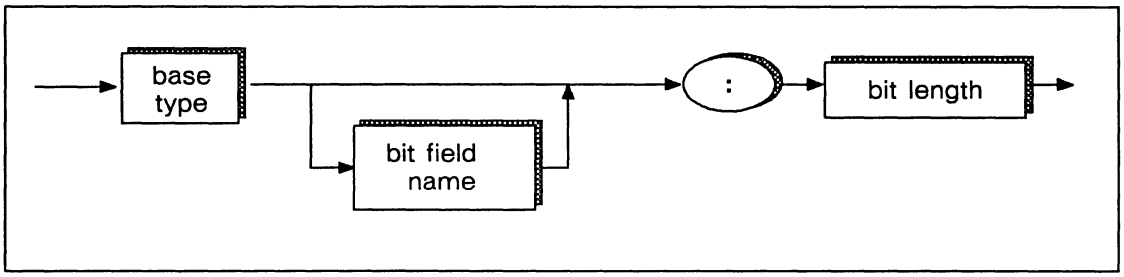


Figure 3–17. Syntax of Bit Field Declarations

Bit fields are always of type **unsigned int**. If you declare them as **int**, the compiler automatically converts them to **unsigned int**.

Bit fields may be named or unnamed. Unnamed fields cannot be accessed and are used only as padding. As a special case, an unnamed bit field with a width of zero causes the next structure member to be aligned on the next shortword boundary.

The **bit length** is an integer constant expression that may not exceed the length of an **int** (32 bits with Domain C).

The compiler assigns bit fields from left to right. The first field starts on a word boundary. After the first field, if the exact number of bits required for the next field crosses only one or zero shortword boundaries, the field starts in the next free bit. If the field would have to cross two shortword boundaries, it starts at the next shortword boundary.

You cannot declare an array of bit fields, and you may not take the address of a bit field (even if it starts on a byte).

For example, given the following declaration of structure **s1**,

```
struct  { char a;
          int  b : 3,
                 : 5; /* unnamed 5-bit field padding. */
          unsigned int c : 2,
               d : 11,
                 : 0;
          float e;
        } s1;
```

Figure 3–18 shows how Domain C represents s1:

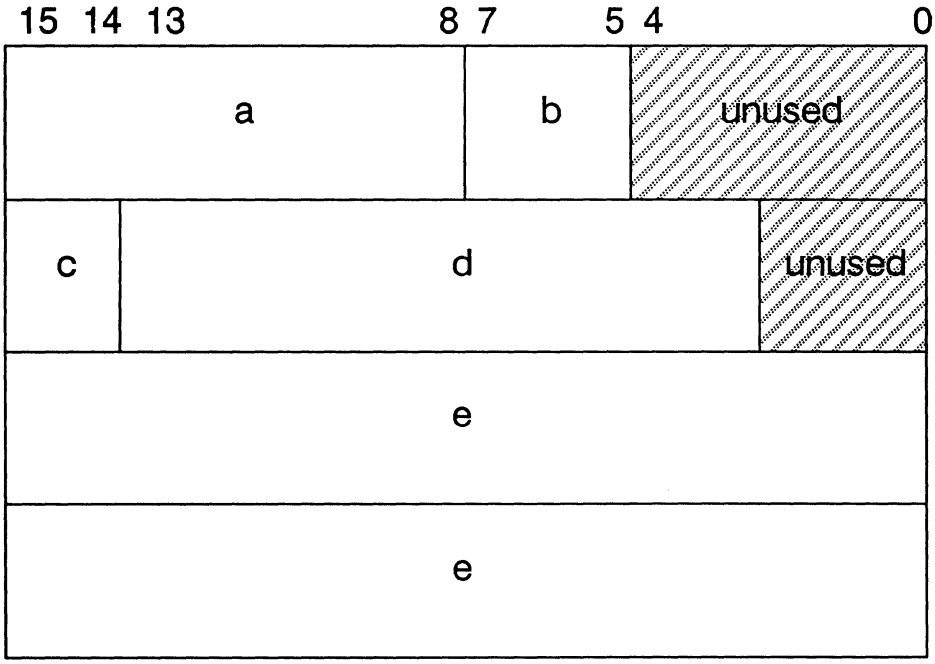


Figure 3–18. Sample Bit–Field Alignment in a Structure

3.8.5 struct and union Name Spaces

Domain C, like most recent C compilers, creates a separate overloading class for each structure and union, so that two or more structures or unions can have components with the same name. (This is consistent with the ANSI standard although it is an extension to the K&R standard.) The following declarations, for example, are legal in Domain C.

```
{
  struct first { int   i;
                 float x;
               } first_struct;

/* struct second has its own overloading class so it may also
 * contain the member names x and i.
 */
  struct second { char   x;
                  double i;
                } second_struct;
}
```

Some older compilers may require that all names be unique. The only restriction that Domain C imposes on member names is that two members of the same structure or union cannot have the same name.

3.8.6 Initializing Structures

You can only initialize structures that have fixed duration; structures with automatic duration cannot be initialized. (This is consistent with the K&R standard but not with the ANSI standard.) To initialize a structure, put the values of the members inside braces; for example:

```
static struct st2 { char c;
                    int  i;
                  } two = {'f', 4};
```

The preceding initialization sets member **two.c** to 'f' and **two.i** to 4.

There may not be more initialization values than there are members in the structure. If there are fewer initialization values than members, the remaining members are initialized to zero (0).

If a structure contains another structure nested within it, the innermost members may be initialized by enclosing them in nested braces. For instance,

```
static struct { char a,b,c,d;
                struct { float  f;
                         double df;
                       } inner;
              } outer = {'x', 'y', 'm', 'a', { 1.0 , 100.0 } };
```

results in the following initializations:

```
outer.a = 'x'
outer.b = 'y'
outer.c = 'm'
outer.d = 'a'
outer.inner.f = 1.0
outer.inner.df = 100.0
```

Note that the inner braces help program readability; however, if we had not used inner braces in the example, we still would have obtained the same results.

3.8.7 Initializing Unions

The Domain C compiler allows unions with fixed duration to be initialized by assigning the initialization value to the first union component. (This is consistent with the ANSI standard but is an extension to the K&R standard.) For example:

```
{
   union init_example { int i;
                        float f;
                      }; static union init_example test = {1};
                        /* Assigns 1 to test.i */
}
```

If you supply more than one initial value, then the last value is the only one that matters. For example, the compiler ignores the values 5 and 'a' in the following declaration:

```
union { int i;
        char a;
        float f;
      } weird = { 5, 'a', 2.3 };
```

If the first component of a union is a structure, the entire structure may be initialized as in:

```
union U { struct { int i;
                   float f;
                 } S;
          char ch[6];
        };
static union U test2 = { 1 , 1.0 };
/* Assigns 1 to test2.S.i and 1.0 to test2.S.f  */
```

Note, however, that if a union contains inner unions, the last inner union is the one that gets directly initialized. For instance, in the following example, **outer.inner2.a2** is directly initialized, not **outer.inner1.a1**. Note, however, that **outer.inner1.a1** is indirectly (and probably incorrectly) initialized.

```
union { union { int  a1;
                int  b1;
              } inner1;

        union { char a2;
                int b2;
              } inner2;

      } outer = {'b'};
```

See the beginning of this chapter for details on how storage class affects initialization.

3.9 Arrays

An array is a collection of identically typed variables stored contiguously in virtual memory. Each element of an array is accessed individually. The syntax for declaring an array is shown in Figure 3-19.

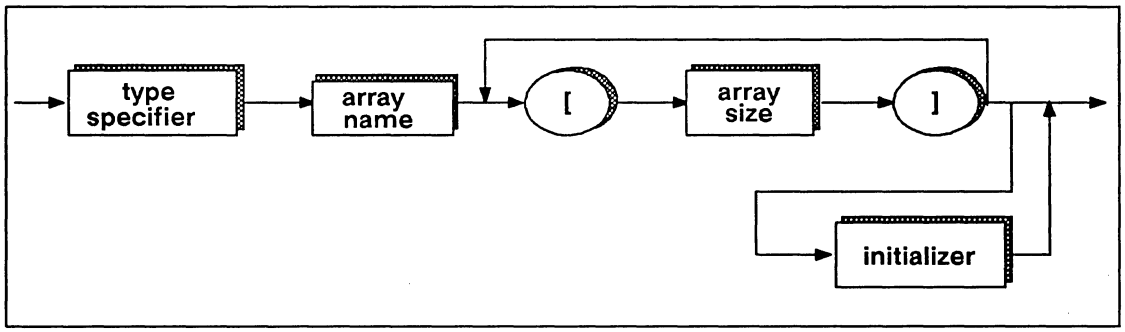


Figure 3-19. Syntax of an Array Declaration

The type specifier is any previously declared Domain C data type except **void** and "function returning...". The *array name* is any identifier. The *array size* is an optional element, but if you do include it, it must be a positive integral expression. Here are some sample array declarations:

```
int    x[5];          /* A 5-element array of ints. */
float  farray[7];     /* A 7-element array of floats. */
char   st[50];        /* A 50-element array of chars. */
short  *y[10];        /* A 10-element array of pointers to
                       * shorts. */
float  (*pf[100])();/* A 100-element array of pointers to
                       * functions that return floats. */
```

In the final example we needed to use parentheses to achieve correct binding. The composition of complex declarations such as this one is discussed in Section 3.11.

In C, arrays start at element 0, so the highest subscript is one less than the array's size. For example, an array declared as

```
char x[3];
```

contains three elements that can be referenced by **x[0]**, **x[1]**, and **x[2]**.

3.9.1 Omitting the Array Size

It is optional to specify explicitly the *array size* under any of the following conditions:

- When you specify initial values for the array. (This is described in Section 3.9.2)
- When you declare an array with the **extern** storage class specifier. If you do omit the array size, then the size of the array is determined by a global declaration of this array (in another file).
- When the array is a function parameter.

3.9.2 Initializing Arrays

Only arrays with fixed duration may be initialized. To initialize an array when you declare it, enter the initialization values separated by commas and enclosed in braces; for example:

```
static float  quatre[4] = { -1.2,  3.8,  -6.3, 10.3 };
```

If the initialization values do not match the data type of the array, the values are converted. For instance, the following line initializes all elements of **a** to 1.

```
static int a[4] = { 1, 3/2, 7-6, 1 };
```

If an initializer does not contain enough values to initialize all the elements of an array, C initializes the remaining elements to zero. For instance, in the following example, elements **a[0]**, **a[1]**, and **a[2]** are initalized to 1, 2, and 3, and elements **a[3]** and **a[4]** are initialized to 0:

```
static int a[5] = { 1, 2, 3 };
```

If an initializer contains too many initalization values, an error occurs.

Note that you can also use array initialization to establish the size of the array. In such a case, the compiler sets the size of the array so that it is just large enough to hold all the initial values. This technique is frequently used for declaring arrays of type **char** initialized with a string constant. For instance, the following three declarations are equivalent:

```
static char string[] = "Example";
static char string[8] = "Example";
static char string[] = { 'E','x','a','m','p','l','e','\0' };
```

Similarly, arrays of pointers may be initialized with string constants. For instance:

```
static char  *str[3] = {"first string" , "second string" ,
                        "third string"};
```

3.9.3 Multidimensional Arrays

An array of arrays is a **multidimensional array** and is declared with consecutive pairs of brackets. For instance:

```
/*  In the following, x is a 3-element array of
 *  5-element arrays.
 */
int x[3][5];

/*  In the following, x is a 3-element array of
 *  4-element arrays of 5-element arrays.
 */
char x[3][4][5];
```

Although a multidimensional array is stored as a 1-dimensional sequence of elements, you can treat it as an array of arrays. For example, consider the following 5x5 "magic square." It is called magic because the rows, columns, and diagonals all have the same sum.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Figure 3-20. Magic Square

To store this square in an array, we could make the following declaration:

```
static int magic[5][5] = { {17 , 24 ,  1 , 8 , 15 },
                           {23 ,  5 ,  7 , 14 , 16 },
                           { 4 ,  6 , 13 , 20 , 22 },
                           {10 , 12 , 19 , 21 ,  3 },
                           {11 , 18 , 25 ,  2 ,  9 }
                         };
```

3.9.3.1 Initializing Multidimensional Arrays

When initializing a multidimensional array, you may enclose each row in braces. If there are too few initializers, the extra elements in the row are initialized to zero. Consider the following example:

```
static int examp[5][3] = { { 1 , 2 , 3 },
                           { 4 },
                           { 5 , 6 , 7 }
                         };
```

This example declares an array with five rows and three columns, but only the first three rows are initialized, and only the first element of the second row is initialized. Pictorially, this declaration produces the following array:

```
1 2 3
4 0 0
5 6 7
0 0 0
0 0 0
```

If we do not include the inner brackets, as in:

```
static int examp[5][3] = { 1 , 2 , 3 ,
                           4 ,
                           5 , 6 , 7
                         };
```

the result is:

```
1 2 3
4 5 6
7 0 0
0 0 0
0 0 0
```

Obviously, the initializer in this example is very misleading. To enhance readability and clarity, you should always enclose each row of initializers in its own set of braces, as we did in the first example.

As with a single-dimension array, if you omit the size specification of a multidimensional array, the compiler automatically determines the size based on the number of initializers present. In the case of multidimensional arrays, however, it is important to remember that you are really declaring an array of arrays. That is, you are declaring an array where each element is itself an array. While you may omit the number of elements in the array you are declaring, you must tell the compiler the size of each element. From a syntactic point of view, this means that you may omit only the *first* size specification, but you must specify the other sizes. For example,

```
static int a_ar[][2][2] = {{{1, 1},
                            {1, 1}},

                           {{1, 1},
                            {1, 1}}};
```

results in a 2-by-2-by-2 cubic array because there are eight initializers. Each element in the array **a_ar** is itself a 2-by-2 array. If we added another initializer, the compiler would allocate space for a 3-by-2-by-2 array, initializing the extra elements to zero. The following declaration is illegal because the compiler has no way of knowing what shape the array should be:

```
/*  ILLEGAL  */
static int b_ar[][] = { 1, 2, 3, 4, 5, 6 };
```

Should the compiler create a 2-by-3 array or a 3-by-2 array? There's no way to tell. However, if you specify the size of each element, the declaration becomes unambiguous.

You can initialize arrays of structures and unions in the same manner as multidimensional arrays. For instance:

```
static struct { int i;
                float f;
              } S[3] = { {1 , 1.0},
                         {2 , 2.0},
                         {3 , 3.0}
                       };
```

Please see the beginning of this chapter for details on how storage class restricts initialization.

3.9.4 Storage of Arrays

The base type of the array establishes its storage allocation. Every element occupies the same amount of space. Each element of a 16-bit (**short**) integer array occupies two bytes; each element of a character (**char**) array occupies one byte; and so forth. The total amount of space that an array uses is equal to the size of the base type multiplied by the number of elements in the array.

For an array of structures or unions, each element is aligned on word boundaries. If it is an array of scalar types, the alignment of elements is the same as the alignment of the scalar type.

An array of arrays is a **multidimensional array**. Multidimensional arrays are stored in row-major order, which means that the last subscript varies fastest. For example, the array declared as

```
int ar[2][3]={ { 0, 1, 2 },
               { 3, 4, 5 }
             };
```

is stored as shown in Figure 3-21.

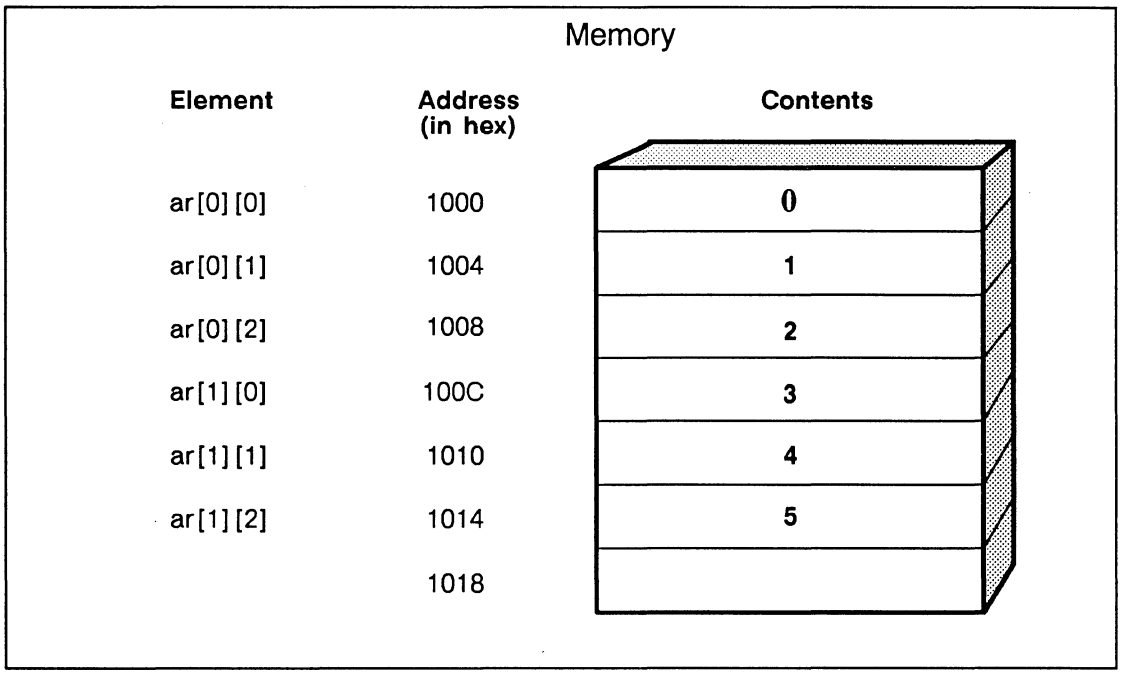


Figure 3–21. Storage of a Multidimensional Array

3.9.5 Strings

A string is an array of characters terminated by a **null character**. A null character is a character with a numeric value of zero. It is represented in C by the escape sequence: **'\0'**. String literals may not be longer than 4095 characters.

To store a string in memory, you need to declare an array of type **char**. You may initialize an array of **char**s with a string constant. For example:

```
static char str[] = "some text";
```

The array is one element longer than the number of characters in the string to accommodate the trailing null character. **str[]**, therefore, is ten characters in length. If you specify an array size, you must allocate enough characters to hold the string. In the following example, for instance, the first four elements are initialized with the characters 'y', 'e', 's', and '\0'. The remaining six elements receive the default initial value of zero:

```
static char str[10] = "yes";
```

The following statement, however, is illegal:

```
static char str[3] = "four";  /* illegal */
```

Note also that you should allocate enough space for the trailing null character, so the following, though legal, is probably not incorrect and will result in a compiler warning:

```
static char str[4] = "four";  /* illegal */
```

You may also initialize a **char** pointer with a string constant. The declaration,

```
char *ptr = "more text";
```

also creates an array of characters initialized with "more text", but it is subtly different from the preceding declaration. Both declarations allocate the same amount of storage for the string and initialize the memory locations with the same values, but the pointer declaration creates an additional 4-byte variable for the pointer.

3.10 Abstract Declarators

A declarator is the part of a declaration that does not include a storage class or initializer. An abstract declarator is the part of the declarator that does not include the variable name. For instance, in the declaration,

```
static char *p="test";
```

the declarator is:

```
char *p
```

and the abstract declarator is:

```
char *
```

There are two situations where a declarator is used without a variable name: in a cast operation and in a **sizeof** operation. The declarator in these cases obeys all the rules discussed in the previous section, except that the variable name is absent. For example,

```
x = (int *[]) y;
```

casts **y** to be an array of pointers to **int**s. Since the pointer operator always appears to the left of the variable name and the array and function operators appear to the right, there is never any ambiguity about where the variable name would be placed if it were a true declaration. To compose or decompose an abstract declarator, follow the rules discussed in Section 3.11.

3.11 Complex Declarations

Declarations in C have a tendency to become complex, making it difficult to determine exactly what is being declared. The following declaration, for instance, declares **x** to be a pointer to a function returning a pointer to a 5-element array of pointers to **int**s:

```
int *(*(*x)())[5];
```

One way to avoid complex declarations such as this one is to create intermediate typedefs, as shown below:

```
typedef int *AP[5];   /* 5-element array of pointers
                       * to ints.
                       */
typedef AP *FP();     /* Function returning pointer to
                       * 5-element array of pointers
                       * to ints.
                       */
FP *x                 /* Pointer to function returning
                       * pointer to 5-element array of
                       * pointers to ints.
                       */
```

The main reason that complex declarations look so forbidding in C is that the pointer operator is a **prefix operator**, whereas the array and function operators are **postfix operators**. As a result, the variable becomes sandwiched between operators. To compose and decipher complex declarations, you must proceed inside-out, adding asterisks to the left of the variable name, and parentheses and brackets to the right of the variable name. It is also important to remember the following three binding and precedence rules:

1. The array operator [ ] and function operator () have a higher precedence than the pointer operator (*).
2. The array and function operators group from left to right, whereas the pointer operator groups from right to left.
3. Parentheses that are not used to denote a function can alter the grouping rules of declarators as they do for expressions.

See Section 4.2.12 for more information about precedence.

3.11.1 Deciphering Complex Declarations

The best strategy for deciphering a declaration is to start with the variable name by itself and then add each part of the declaration, starting with the operators that are closest to the variable name. In the absence of parentheses to affect binding, you would add all of the function and array operators on the right side of the variable name first (since they have higher precedence), and then add the pointer operators on the left side. The declaration,

```
char *x[];
```

would be deciphered through the following steps:

1. x[] is an **array**.
2. *x[] is an **array** of **pointers**.
3. char *x[] is an **array** of **pointers** to **chars**.

Parentheses can be used to change the precedence order. For example,

```
int (*x[])();
```

would be broken down as follows:

1. x[] is an **array**.
2. (*x[ ]) is an **array** of **pointers**.
3. (*x[ ])() is an **array** of **pointers** to **functions**.
4. int (*x[ ])() is an **array** of **pointers** to **functions** returning **ints**.

If this declaration had been written without the parentheses as:

```
int *x[]();
```

it would have been translated as:

an **array** of **functions** returning **pointers** to **ints**

which is an illegal declaration since arrays of functions are invalid.

3.11.2 Composing Complex Declarations

To compose a declaration, you perform the same process. For example, to declare a **pointer** to an **array** of **pointers** to **functions** that return **pointers** to **arrays** of **structures** with tag name **S**, you could use the following steps:

1. (*x) is a **pointer**.
2. (*x)[] is a **pointer** to an **array**.
3. (*(*x)[]) is a **pointer** to an **array** of **pointers**.
4. (*(*x)[])() is a **pointer** to an **array** of **pointers** to **functions**.
5. (*(*(*x)[])()) is a **pointer** to an **array** of **pointers** to **functions** returning **pointers**.
6. (*(*(*x)[])())[] is a **pointer** to an **array** of **pointers** to **functions** returning **pointers** to **arrays**.
7. struct S (*(*(*x)[])())[] is a **pointer** to an **array** of **pointers** to **functions** returning **pointers** to **arrays** of **structures** with tag name **S**.

Note that we add parentheses for binding each time we add a new pointer operator.

Table 3–2 shows a number of legal and illegal declarations. Note that it is illegal to declare the following:

- Arrays of functions
- Functions returning functions
- Functions returning arrays

Table 3–2. Legal and Illegal Declarations in Domain C

`int i;`	An **int**
`int *p;`	A pointer to an **int**
`int a[];`	An array of **ints**
`int f();`	A function returning an **int**
`int **pp;`	A pointer to a pointer to an **int**
`int (*pa)[];`	A pointer to an array of **ints**
`int (*pf)();`	A pointer to a function returning an **int**
`int *ap[];`	An array of pointers to **ints**
`int aa[][];`	An array of arrays of **ints**
`int af[]();`	An array of functions returning **ints** (ILLEGAL)
`int *fp();`	A function returning a pointer to an **int**
`int fa()[];`	A function returning an array of **ints** (ILLEGAL)
`int ff()();`	A function returning a function returning an **int** (ILLEGAL)
`int ***ppp;`	A pointer to a pointer to a pointer to an **int**
`int (**ppa)[];`	A pointer to a pointer to an array of **ints**
`int (**ppf)();`	A pointer to a pointer to a function returning an **int**
`int *(*pap)[];`	A pointer to an array of pointers to **ints**
`int (*paa)[][];`	A pointer to an array of arrays of **ints**
`int (*paf)[]();`	A pointer to an array of functions returning **ints** (ILLEGAL)
`int *(*pfp)();`	A pointer to a function returning a pointer to an **int**
`int (*pfa)()[];`	A pointer to a function returning an array of **ints** (ILLEGAL)
`int (*pff)()();`	A pointer to a function returning a function returning an **int** (ILLEGAL)
`int **app[];`	An array of pointers to pointers to **ints**
`int (*apa[])[];`	An array of pointers to arrays of **ints**
`int (*apf[])();`	An array of pointers to functions returning **ints**
`int *aap[][];`	An array of arrays of pointers to **ints**
`int aaa[][][];`	An array of arrays of arrays of **ints**
`int aaf[][]();`	An array of arrays of functions returning **ints** (ILLEGAL)
`int *afp[]();`	An array of functions returning pointers to **ints**
`int afa[]()[];`	An array of functions returning arrays of **ints** (ILLEGAL)
`int aff[]()();`	An array of functions returning functions returning **ints** (ILLEGAL)
`int **fpp();`	A function returning a pointer to a pointer to an **int**
`int (*fpa())[];`	A function returning a pointer to an array of **ints**
`int (*fpf())();`	A function returning a pointer to a function returning an **int**
`int *fap()[];`	A function returning an array of pointers to **ints** (ILLEGAL)
`int faa()[][];`	A function returning an array of arrays of **ints** (ILLEGAL)
`int faf()[]();`	A function returning an array of functions returning **ints** (ILLEGAL)
`int *ffp()();`	A function returning a function returning a pointer to an **int** (ILLEGAL)

3.12 Storage Classes

Every variable has several characteristics. One of those characteristics is its data type (which is detailed in Chapter 3). Another characteristic is its **storage class,** which we describe in this section. Storage classes describe two properties of variables—**duration** and **scope**. Duration represents the period over which memory is allocated for a variable. Scope refers to the region in the source code over which a variable's name has meaning.

You control a variable's storage class through *both* of the following:

- **position** Where in the file you declare the variable.
- **storage class specifier** An optional keyword in a declaration.

In the example below, variable x has fixed duration and global scope because it appears outside of a function; variable y has fixed duration because it is preceded by the **static** storage class specifier, and block scope because it is declared within a block:

```
int x = 1;

int main( void )
{
  static int y;
       .
       .
}
```

3.12.1 Declaration Position

Variable declarations fall into three categories, based on their position in a source file:

top-level declaration	A declaration that occurs outside of a function.
head-of-block declaration	A declaration that occurs at the beginning of a block. A block is any series of statements enclosed in braces. Note that the body of a function is itself a block.
function parameter declaration	A declaration of a function parameter.

The comments in the following program fragment illustrate these three types of declarations:

```
int    a[5];                  /* top-level declaration */
float  value;                 /* top-level declaration */

int  f( int arg1, char arg2) /* top-level declaration */
{
  unsigned char   count;      /* head-of-block declaration */
  int             q;          /* head-of-block declaration */
     .
     .
     .
     for(count = 0; count <= 200; count += 5)
        {
          int  so;            /* head-of-block declaration */
        .
        .
        .
        }
     .
     .
     .
}


main()                        /* top-level declaration */
{
  int  *p, m;                 /* head-of-block declaration */
     .
     .
     .
}
```

3.12.2 Scope of a Variable Declaration

The scope of a variable is the region in the source code over which a name's declaration is active. If a variable is active, it means that it is accessible.

There are four types of scope: **program**, **file**, **function**, and **block**.

- **Program scope** signifies that the variable is active among different source files that make up the entire executable program. Variables with program scope are usually referred to as **global variables**.
- **File scope** signifies that the variable is active from its declaration point to the end of the source file.
- **Function scope** signifies that the name is active from the beginning to the end of the function.
- **Block scope** signifies that the variable is active from its declaration point to the end of the block in which it is declared. A block is any series of statements enclosed in braces. This includes compound statements as well as function bodies.

In general, the scope of a variable is determined by the location of its declaration. Variables declared within a block have block scope; variables declared outside of a block have file scope if the **static** keyword is present, or program scope if **static** is not present; only **goto** labels have function scope.

The four scopes are arranged hierarchically as shown in Figure 3–22. A variable with program scope is also active within all files, functions, and blocks that make up the program. Likewise, a variable with file scope is also active within all functions and blocks in the file that follow its declaration, but is not active in other parts of the program. At the bottom of the hierarchy is block scope, the most limiting case.

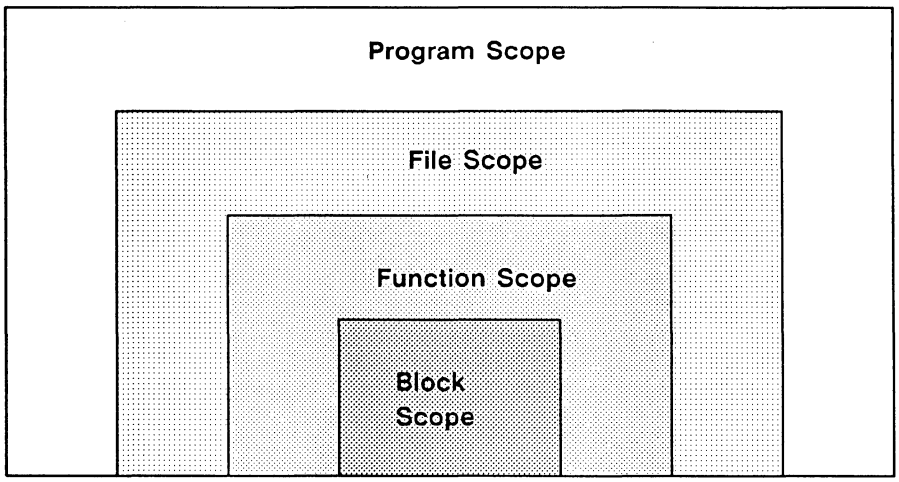


Figure 3–22. Hierarchy of Active Regions (Scopes)

The program fragment below shows variables with all four types of scope:

```
int i;          /* Program scope */
static int j;   /* File scope */

func( k )       /* Program scope */
int k;          /* Block scope */
{
  int m;        /* Block scope */

  start:        /* Function scope */
     .
     .
```

Note that function parameters have block scope. They are treated as if they are the first declarations in the top–level block of the function.

The C language allows you to give two variables the same name, provided they have different scopes. For example, the two functions below both use a variable called **j**, but because they are declared in different blocks, they do not conflict.

```
func1()
{
  int j;
     .
     .
}

func2()
{
  int j;
     .
     .
}
```

3.12.2.1 Visibility

The **visibility** of a variable determines whether or not the variable can be accessed in a specific region of the source file. A variable can become invisible throughout a region if another variable with the same name and name space is declared within the region in a new block. For instance:

```
/* Program name is "scope_example" */
#include <stdio.h>

int j = 10;      /* Program scope */

int main( void )
{
  int j;    /* Block scope -- hides global j */

  for (j=0; j < 5; ++j)
    printf( "j: %d\n", j );
}
```

There are two **j**'s, one with program scope and the other with block scope. Although they have the same name, they are distinct variables. The **j** with block scope temporarily hides the other **j**, so the result of running the program is:

```
j: 0
j: 1
j: 2
j: 3
j: 4
```

The **j** with program scope retains its value of 10.

3.12.2.2 Block Scope

A variable with block scope cannot be accessed outside of its block. Block scoping allows you to write sections of code without worrying about whether your variable names conflict with names used in other parts of the program.

It is also possible to declare a variable within a nested block. This temporarily hides any variables of the same name declared in outer blocks. This feature can be useful when you want to add some debugging code into a function. By creating a new block and declaring variables within it, you eliminate the possibility of naming conflicts. In addition, if you delete the debugging code at a later date, you need not look at the top of the function to find variable declarations that also need to be deleted.

In the following example, we add some debugging code that prints the values of the first ten elements of an array.

```
foo()
{
  int ar[20];
  int j;
        .
        .
/* Begin debug code */
  {
    /* This j does not conflict with other j's.*/
    int j;
    for (j=0; j <= 10; ++j)
      printf( "%d\t", ar[j] );
  }
/* End debug code */
       .
       .
}
```

3.12.2.3 Function Scope

The only names that have function scope are **goto** labels. Labels are active from the beginning to the end of a function. This means that labels must be unique within a function. Different functions, however, may use the same label names without creating conflicts.

3.12.2.4 File and Program Scope

Giving a variable **file scope** makes the variable active throughout the rest of the file. So if a file contains more than one function, all of the functions following the declaration are able to use the variable. To give a variable file scope, declare it outside of a function with the **static** keyword.

Variables with program scope, called **global variables**, are visible to routines in other files as well as their own file. To create a global variable, declare it outside of a function without the **static** keyword. In the following program segment, **j** has program scope and **k** has file scope. Both variables can be accessed by routines in the same file, but only **j** can be accessed by routines in other files.

```
int j;
static int k;

main()
{
      .
      .
```

Variables with file scope are particularly useful when you have a number of functions that operate on a shared data structure, but you don't want to make the data visible to other functions.

3.12.3 Duration of a Variable

The duration of a variable describes the lifetime of a variable's memory storage. There are two categories of duration: **automatic** and **fixed**. As the names imply, a fixed variable is one that is stationary, whereas an automatic variable is one whose memory storage is automatically allocated when its scope is entered during program execution. This means that a fixed variable has memory allocated for it at program start-up time, and the variable is associated with a single memory location until the end of the program. An automatic variable has memory allocated for it whenever its scope is entered. The automatic variable refers to that memory address only as long as code within the scope is being executed. Once the scope of the automatic variable is exited, the compiler is free to assign that memory location to the next automatic variable it sees. If the scope is re-entered, a new address is allocated for the variable. There is no way to ensure that an automatic variable will retain its value from one scope entry to another.

The difference between fixed and automatic variables is especially important for initialized variables. Fixed variables are initialized only *once* whereas automatic variables are initialized *each time* their block is re-entered. Consider the following program:

```
/* Program name is "example_of_static" */
#include <stdio.h>

void increment( void )
{
  int j = 1;
  static int k = 1;

  j++;
  k++;
  printf( "j: %d\tk: %d\n", j, k );
}

int main( void )
{
  increment();
  increment();
  increment();
}
```

The **increment()** function increments two variables, **j** and **k**, both initialized to 1. **j** has automatic duration by default, while **k** has fixed duration because of the **static** keyword. The result of running the program is:

```
j: 2      k: 2
j: 2      k: 3
j: 2      k: 4
```

When **increment()** is called the second time, memory for **j** is reallocated and **j** is reinitialized to 1. **k**, on the other hand, has still maintained its memory address and is *not* reini-

tialized, so its value of 2 from the first function call is still present. No matter how many times we call **increment()**, the value of **j** will always be 2, while **k** will increase by 1 every time we call it.

We can summarize this observation with the following rule: *an automatic variable, when declared with an initializer, is re-initialized every time its block is re-entered; a fixed variable is initialized only once, at program startup-time.*

Another important difference between automatic and fixed variables is that automatic variables are not initialized by default, whereas fixed variables get a default initial value of zero. If we rewrite the previous program without initializing the variables, we get:

```
/* Program name is "init_example" */
#include <stdio.h>

void increment( void )
{
  int j;
  static int k;

  j++;
  k++;
  printf( "j: %d\tk: %d\n", j, k );
}

int main( void )
{
  increment();
  increment();
  increment();
}
```

Executing the program on our machine results in:

```
j: 52517483 k: 1
j: 52517483 k: 2
j: 52517483 k: 3
```

The values of **j** are random because the variable is never initialized. With each invocation of **increment()**, **j** receives a new memory allocation and acquires whatever "garbage" value happens to be at the new location. Because Domain C uses a stack-frame implementation, the garbage values are, in this simple example, the same each time. The C language, however, does not guarantee this. If you use a more complicated calling sequence, the results will be different. The Domain C compiler issues the following warning if you attempt to use an uninitialized automatic variable before you have made an assignment to it:

```
******** Line 15: Warning:  Variable "auto2" was not
initialized before this use.
No errors, 1 warning, C Compiler, Rev 4.82
```

Another difference between initializing variables with fixed and automatic duration is the kinds of expressions that may be used as an initializer. For scalar variables with automatic duration, the initializer may be any expression, so long as all of the variables in the expression have been previously declared. For example, all of the following declarations are legal:

```
{
  extern double f();
  int x = 10, y = x*x;
  float z = x + f(x);
        .
        .
```

For variables with fixed duration, on the other hand, the initilization expressions must be constant expressions.

We can summarize the differences between fixed and automatic variables as follows:

- Fixed variables maintain their values from one block invocation to another, but automatic variables lose their value each time the block is deactivated.
- Fixed variables get a default initialization value of zero if you do not explicitly initialize them. If you do not explicitly initialize an automatic variable, the compiler will not initialize it for you.
- The run-time system initializes fixed variables only once, whereas automatic variables, if they are declared with an intializer, are re-initialized each time their block is entered.

Bug Alert: The Dual Meanings of "static"

One of the most confusing aspects about storage-class declarations in C is that the **static** keyword seems to have two effects depending on where it appears. In a declaration within a block, **static** gives a variable fixed duration instead of automatic duration. Outside of a function, on the other hand, **static** has nothing to do with duration. Rather, it controls the scope of a variable, giving it file scope instead of program scope.

One way of reconciling these dual meanings is to think of **static** as signifying both file scoping and fixed duration. Within a block, the stricter block scoping rules override **static**'s file scoping, so fixed duration is the only manifest result. Outside of a function, duration is already fixed, so file scoping is the only manifest result.

3.12.4 Storage Class Specifiers

As mentioned earlier, you can supply an optional **storage class specifier** when you declare a variable. There are four storage-class specifiers (**auto, static, extern,** and **register**). Any of the storage class keywords may appear before or after the type name in a declaration, but by convention they come before the type name. (The ANSI standard requires that storage class specifiers appear before type specifiers.) The semantics of each keyword depends to some extent on the location of the declaration. Omitting a storage class specifier also has a meaning, as described below. Table 3–3 summarizes the scope and duration semantics of each storage class specifier.

auto

The **auto** keyword, which makes a variable automatic, is legal only for variables with block scope. Since this is the default anyway, **auto** is somewhat superfluous and is rarely used.

static

The **static** keyword may be applied to declarations both within and outside of a function (except for function arguments), but the meaning differs in the two cases. In declarations within a function, **static** causes the variable to have fixed duration instead of the default automatic duration. For variables declared outside of a function, the **static** keyword gives the variable file scope instead of program scope.

extern

The **extern** specifier may be used for declarations both within and outside of a function (except for function arguments). In both cases, it signifies a global allusion, discussed in Section 3.13.

register

The **register** keyword may be used only for variables declared within a function. It makes the variable automatic, but also passes a hint to the compiler to store the variable in a register whenever possible. You should use the **register** keyword for automatic variables that are accessed frequently. Compilers support this feature at various levels. Some don't support it at all, while others support as many as 20 concurrent register assignments. Note that it is illegal to apply the address-of operator (**&**) to any variable declared with **register**.

omitted

For variables with block scope, omitting a storage class specifier is the same as specifying **auto**. For variables declared outside of a function, omitting the storage class specifier is the same as specifying **extern**. It causes the compiler to produce a global definition.

Here are some sample declarations that contain storage class specifiers:

```
auto     int    i;
register short  quart;
static   char   dog[] = "Fenster";
extern   float  f;
```

Table 3-3. Storage Class Summary

Storage Class Specifier \ Place Where Declared	Outside of a Function (top-level)	Within a Function (head-of-block)	Function Arguments
auto or **register**	NOT ALLOWED	scope: block duration: automatic	scope: block duration: automatic
static	scope: file duration: fixed	scope: block duration: fixed	NOT ALLOWED
extern	scope: program duration: fixed	scope: block duration: fixed	NOT ALLOWED
No storage class specifier present	scope: program duration: fixed	scope: block duration: automatic	scope: block duration: automatic

3.12.5 The register Specifier

The **register** keyword enables you to help the compiler by giving it suggestions about which variables should be kept in registers. However, **register** is only a hint, not a directive—the compiler is free to ignore it. In fact, the Domain compiler is so efficient in allocating variables to registers that using the **register** keyword has little or no effect on most programs.

Since a variable declared with **register** might never be assigned a memory address, it is illegal to take the address of a **register** variable (registers are not addressable). This is true regardless of whether the variable is actually assigned to a register. You will get a compile-time error if you ever try to take the address of a variable declared with **register**.

3.13 Global Variables

A **global variable** (also called an **external variable**) is one that can be accessed by modules in different source files; that is, a global variable has program scope. There are two types of declarations for global variables: **allusions** and **definitions**, as described in the next section.

3.13.1 Definitions and Allusions

The difference between an **allusion** and a **definition** in C is subtle but important. An allusion associates a data type with an identifier, but does not actually allocate any storage for it. A definition, on the other hand, actually allocates memory. For example, consider the following allusions and definitions:

```
        int  x;             /* This is a definition  */
static  int  y;             /* This is a definition  */
extern  int  z;             /* This is an allusion   */
```

If you use the storage class specifier **extern**, you generate an allusion. If you use a storage class specifier other than **extern**, or if you omit a storage class specifier, then you generate a variable definition.

The distinction between allusions and definitions is particularly important when creating global variables.

> **NOTE:** At some points during this manual, the distinction between an allusion and a definition is unimportant For these instances, we use the more general word "declaration."

Typically, you put all allusions in a header file, which can be included in other source files. This ensures that all source files use consistent allusions. Any change to a declaration in a header file is automatically propagated to all source files that include that header file.

3.13.2 Defining Global Variables

In Domain C, every global variable can be alluded to zero or more times (in different files), but must be defined at least once. It may be defined more than once in different files. You cannot, however, define a global variable more than once in the *same* file. If you explicitly initialize a global variable in more than one file, the last initializer read by the linker is the variable's initial value at run time. Therefore, the order in which you list files in the **bind** or **ld** command determines the initial values of external variables. If you do not initialize a global definition, its initial value defaults to 0. To demonstrate these rules, consider Figures 3–23, 3–24, and 3–25.

t1.c

```
extern int x;/*all*/

main()
{
  printf("%5d", x);
  f();
  g();
}
```

t2.c

```
extern int x;/*all*/

f()
{
  printf("%5d", x);
}
```

t3.c

```
int x;/*def*/

g()
{
  printf("%5d\n", x);
}
```

```
$ cc t1; cc t2; cc t3
$ bind t1.bin t2.bin t3.bin -b t
$ t
    0    0    0
```

```
$ cc t1.c t2.c t3.c

$ a.out
    0    0    0
```

Figure 3-23. Two Declarations and One Definition with No Initialization

t1.c

```
extern int x;/*all*/

main()
{
  printf("%5d", x);
  f();
  g();
}
```

t2.c

```
extern int x;/*all*/

f()
{
  printf("%5d", x);
}
```

t3.c

```
int x = 5;/*def*/

g()
{
  printf("%5d\n", x);
}
```

```
$ cc t1; cc t2; cc t3
$ bind t1.bin t2.bin t3.bin -b t
$ t
    5    5    5
```

```
$ cc t1.c t2.c t3.c

$ a.out
    5    5    5
```

Figure 3-24. The Effect of Initializing a Global Variable

t1.c

```
extern int x;/*all*/

main()
{
  printf("%5d", x);
  f();
  g();
}
```

t2.c

```
int x = 7;/*def*/

f()
{
  printf("%5d", x);
}
```

t3.c

```
int x = 5;/*def*/

g()
{
  printf("%5d\n", x);
}
```

```
$ cc t1; cc t2; cc t3
$ bind t1.bin t2.bin t3.bin -b t
$ t
    5    5    5

$ bind t1.bin t2.bin t3.bin -b t
$ t
    7    7    7
```

```
$ cc t1.c t2.c t3.c

$ a.out
    5    5    5

$ cc t1.c t2.c t3.c
$ a.out
    7    7    7
```

Figure 3–25. The Effect of Linking Order on Variable Initialization

For further clarification on global variables, we provide the following program fragments:

Here is FILE 1:

```
int d1;   /* This is a definition of a global variable. */
int d2=1; /* This is a definition of a global variable with an
           * initializer.
           */
extern int d3; /* This is an allusion to a global variable defined
                * in FILE2.
                */
/* extern int d4=5;   THIS IS ILLEGAL! You cannot initialize an
 *                    allusion.
 */

int main( void )
{
  int local;  /* This is a definition of a local variable. It is
               * not exported by the binder.
               */
  extern int d5;  /* This is an allusion to a global variable
                   * defined in FILE 2.
                   */
}
```

Here is FILE 2:

```
int  d3 = 0;  /* This is a definition of a global variable. */
char d5;      /* This is a definition of a global variable. */

void some_function( void )
{
  extern int d1;   /* This is an allusion to the variable defined on
                    * line #1 of FILE 1.
                    */
        .
        .
}
```

3.13.3 Portability Considerations Regarding Global Variables

If you are planning to port your Domain C programs to a different machine, take into account that not all compilers use the same strategy for external definitions and declarations. For maximum portability, follow these guidelines:

- Do not *define* the same global variable more than once in the same program. Domain C permits you to define a global variable multiple times, but other C compilers may be stricter.
- For each routine that refers to a global variable, declare the variable with the keyword **extern**, and without an initializer.

3.13.4 Sections

The Domain C compiler creates a named **section** for each globally defined variable. Sections are detailed in the *Domain/OS Programming Environment Reference* manual. When the object files are bound together, the linker makes sure that all global variables with the same name refer to the same named section.

3.14 Storage Class of Functions

Just like variables, functions also have a scope, although the rules are somewhat different. When discussing storage class of functions it is important to distinguish between function definitions and function allusions.

3.14.1 Function Definitions

A **function definition** is a complete function—that is, a data type that the function returns, the name of the function, optional parameters, parameter declarations, and the function body. For example, here is the function definition of a function named **fun**:

```
#include <stdio.h>

int fun( int x, int y)
{
   printf( "%d %d", x, y );
   return (x + y);
}
```

By default, function definitions have global scope. In other words, you can call these routines from any place in the program (including some other file). If you want the function definition to have file scope instead, then use the storage class specifier **static**. For example:

```
#include <stdio.h>

static int fun( int x, int y)
{
   printf( "%d %d", x, y );
   return (x + y);
}
```

By using **static**, you limit the scope function **fun** to the file in which it is defined. Note that **static** is the only legal storage class specifier for a function definition.

3.14.2 Function Allusions

A function allusion identifies a function that is defined elsewhere, either in the same source file or in another source file. A function allusion can begin with the **extern** storage class specifier. It optionally contains the data type that the function can return, and concludes with the name of the function followed by an empty pair of parentheses. (This is the old-style type of function allusion; the new style uses prototypes, as described below.) For example:

```
extern int fun();
extern fun();
int fun();
```

Note that you can omit *either* the type specifier or **extern**, but not *both*. If you omit both, the declaration will be interpreted as a function invocation.

Domain C supports a new syntax for function allusions called **prototypes**. A prototype enables you to specify the types and number of arguments that the function accepts. For example:

```
extern int fun( int, char *, float );
```

Prototypes are described in detail in Chapter 5.

You can specify a function allusion either within a block or outside of a block. When declared within a block, it means that you can invoke that function within the block. When declared outside of a block, you can invoke the function anywhere from the declaration point to the end of the source file. Technically, you do not need to declare functions that return an **int** since this is the default. However, it is good programming practice to declare all functions since it makes your programs easier to understand.

For more information on function allusions and definitions, see Chapter 5.

3.15 Reference Variables — Domain Extension

The Domain C compiler supports **reference variables** as implemented in the C++ language. This discussion describes the most common usages of reference variables. For a more complete discussion, we recommend that you read *The C++ Programming Language* by Bjarne Stroustrup. (Reference variable features will not be activated if you compile with the **-ntype** option.)

A reference variable is a variable that refers to another object (an **lvalue** or an **rvalue**). Whenever a reference variable appears in an expression, the object it denotes is accessed. Reference variables have three main applications:

- Reference variables allow you to create **aliases** for a variable so that two or more names refer to the same object.
- Reference variables allow you to give names to constants, and, more importantly, to use the constants as lvalues. In effect, reference variables turn constants into variables.
- Reference variables provide a clean syntax for passing function arguments by reference.

These applications of reference variables are discussed in Chapter 5. The following section describes how to declare reference variables.

3.15.1 Declaring Reference Variables

To declare a reference variable, precede the variable name with the address-of operator (&) and include an initializer:

```
int j;
int &rj = j;          /* rj refers to j */
float &rf = 3.141;    /* rf refers to the constant 3.141 */
```

The initializer is required because it specifies the object that the reference variable denotes. Having made these declarations, you can write:

```
rj = 1;  /* assigns 1 to j */
rj++;    /* increments j */
rf *= rf /* squares 3.141 */
```

The last example is the most interesting because it uses a reference variable denoting a constant as an lvalue. This is legal because the compiler generates a temporary variable for all reference variables initialized with a constant value. For example, the declaration,

```
int &r = 0;
```

causes the compiler to generate a hidden temporary variable initialized to zero. Whenever **r** appears in an expression, this hidden variable is accessed.

3.16 The #attribute Modifier — Domain Extension

The Domain C compiler supports a declaration modifier called **#attribute** that enables you to access special features of the Domain C compiler. One of the purposes of **#attribute** is to turn off certain kinds of compiler optimizations. This feature is particularly useful for writing device drivers or other programs that access fixed memory locations.

Although it begins with the # character, **#attribute** is a reserved word, not a preprocessor statement. You use it when you declare or define a variable, tag name, or typedef. The **#attribute** modifier always takes one of the following arguments (called **attribute specifiers**) enclosed in brackets:

address	Binds a variable to a specific virtual address.
device	Informs the compiler that the variable is a device register. The **device** specifier is similar to **volatile**, but restricts optimizations even further.
section	Specifies a named section in which to overlay the variable.
volatile	Informs the compiler that the variable may change in ways that it cannot predict. Consequently, the compiler refrains from executing certain optimizations.

Each of these specifiers is described in detail in later sections. First, however, we provide some general information about the **#attribute** modifier.

3.16.1 Inheritance of Declaration Modifiers

The **device**, **volatile**, and **section** modifiers are inheritable in the type declaration hierarchy. That is, if you define a type in terms of some more primitive type that was declared with one or more of these modifiers, then the new type inherits those modifiers. For example, the following declaration defines a type called **SEMAPHORE** and an array called **resource**:

```
typedef int SEMAPHORE #attribute[volatile];
SEMAPHORE resource[10];
```

The **resource** array inherits the **volatile** storage class from the definition of the **SEMAPHORE** typedef. Note that this rule does not apply to the **address** specifier because this specifier is valid only in variable definitions, not in tag name or typedef declarations.

3.16.2 #attribute and Pointer Types

It is usually incorrect to associate the **device** and **volatile** specifiers with a pointer type. For example, declaring a pointer to a device register by means of the following declaration is almost certainly incorrect:

```
int *iodata #attribute[device];
```

The correct specification is:

```
typedef int DEVINT #attribute[device];
DEVINT *iodata;
```

which declares a pointer to an **int** with the **#attribute** modifier, rather than assigning the modifier to the pointer itself.

3.16.3 The volatile Specifier

The syntax of the **volatile** specifier is:

[*specifier*] *data_type variable_name* **#attribute[volatile]** [*initializer*]

where *specifier* can be **extern**, **auto**, **static**, **register**, or **typedef**.

The **volatile** specifier informs the compiler that memory contents may change in a way that the compiler cannot predict. There are two situations, in particular, where this might occur:

- The variable is in a shared memory location accessed by two or more processes.
- The variable can be accessed by two different access paths. (That is, multiple pointers with different base types refer to the same memory locations.)

In both of these situations, it is crucial that you tell the compiler *not* to perform certain optimizations as it normally would. For example, the following code causes optimizations leading to erroneous code.

```
/* Program name is "volatile_example" */
#include <stdio.h>
#ifdef ATTR
#  define VOL #attribute[volatile]
#else
#  define VOL
#endif
typedef int VINT VOL;

void killer( int a, VINT b )
{
  int j;
  int *p = &a + 1;
  j = b*(b+1);
  *p = 0;
  j = j + b*(b+1);
  printf( "b = %d\n", b*(b+1) );
}

int main( void )
{
  killer( 1, 2 );
}
```

In the preceding program, the compiler sees that the calculation

```
b * (b+1)
```

is done three times without any change to **b**. Since it appears to the compiler that it is wasteful to do the same calculation needlessly, it will make the calculation only once, then store the result in a register. Then, instead of calculating it a second or third time, the value will simply be fetched from the register. The problem with this optimization is that **b**'s value is indirectly changed between the first and second calculations. Therefore, you must use **#attribute[volatile]** to tell the compiler to avoid the optimization. Notice that **#attribute** is defined in a conditional compilation directive. Therefore, if we compile with the following compilation option:

```
-def ATTR
```

and run the resulting program, we get the following results:

```
b = 0
```

However, if we compile without the **-def ATTR** option, and we run the program, we get the following results:

```
b = 6
```

3.16.4 The device Specifier

The syntax for the device specifier is:

$$\left[\textit{specifier}\right]\ \textit{data\_type}\ \textit{variable\_name}\ \#\textbf{attribute}[\textbf{device}\left[(\left[\textbf{read,}\right]\ \left[\textbf{write}\right])\right]]\ \left[\textit{initializer}\right]$$

where *specifier* can be **extern**, **auto**, **static**, **register**, or **typedef**.

The **device** specifier informs the compiler that a device register (control or data) is mapped as a specific virtual address. The **device** specifier prevents the same optimizations that **volatile** prevents, and prevents two other optimizations as well.

The first optimization that **device** prevents concerns adjacent references. By default, the compiler optimizes certain adjacent references by merging them into large reference. The **device** specifier prevents this optimization. For example, consider the following fragment:

```
short int  a,b;

    a=0;
    b=0;
```

By default, the compiler optimizes the two 16-bit assignments by merging them into one 32-bit assignment. (That is, at run time, the system assigns a 32-bit zero instead of assigning two 16-bit zeros.) By specifying the **device** specifier, you suppress this optimization.

The **device** specifier also prevents the compiler from generating gratuitous read-modify-write references for device registers. That is, specifying a variable as **#device** causes the compiler to avoid using instructions that do unnecessary reads.

Now let's demonstrate **device** through some examples. Suppose **kb** in the following fragment is a device register that accepts characters from the keyboard:

```
char   c, c1, *kb;

    .
    .
    .
    c  = *kb;
    c1 = *kb;
```

The purpose of the program is to read a character from the keyboard and store it in **c**, then read the next character and store it in **c1**. However, the C compiler, unaware that the value of **kb** can be changed outside of the block, optimizes the code as follows: It stores the value of **kb** in a register, and thus assigns both **c** and **c1** identical values. Obviously, this is not what the programmer intended, since Domain C assigns the same character to both **c** and **c1**. To ensure that Domain C reads **kb** twice, declare it as:

```
char  *kb #attribute[device];
```

Another situation where normal optimization techniques can change the meaning of a program is in loop–invariant expressions. For instance, using **kb** again, suppose we have the program segment:

```
int x;
char c, *kb;

{
  while (x < 10)
  {
    c = *kb;
    foo(c);
    ++x;
  }
```

The purpose of the block is to read 10 successive characters from the keyboard and pass each to a function called **foo**. However, to the compiler, it looks like an inefficient program since **c** will be assigned the same value 10 times. To optimize the program, the compiler may translate it as if it had been written:

```
int x;
char c, *kb;

{
  c = *kb;
  while (x < 10)
  {
    foo(c);
    ++x;
  }
```

To ensure that the compiler does not optimize your program in that manner, declare **kb** as follows:

```
char c #attribute[device];
```

In addition to suppressing optimizations, you can also use **device** to specify that a device is either exclusively read from or exclusively written to. You achieve this by using the read and write options:

device(read) — This attribute specifies read-only access for this variable or type. That is, if you attempt to write to this variable, the compiler flags the attempt as invalid and issues an error message. Although the syntax is available, the read and write options currently have no effect.

device(write) — This attribute specifies write-only access for this variable or type. That is, if you attempt to read from this variable, the compiler flags the attempt as invalid and issues an error message. Although the syntax is available, the read and write options currently have no effect. It will be implemented in a future release of Domain C.

device(read,write) — This attribute specifies both read and write access for this variable. Using it is identical to using **device** by itself (without any options).

device(write,read) — Same as **device(read,write)**.

For example, here are some sample declarations using **device**:

```
typedef int a[10] #attribute[device(read)];   /* read access */
        char c    #attribute[device(write);   /* write access */
        char c2   #attribute[device(read,write)]; /* read and
                                                   * write
                                                   * access */
```

3.16.5 The address Specifier

The syntax for the **address** specifier is:

[*specifier*] *data_type variable_name* **#attribute[address]** [*initializer*]

where *specifier* can be **auto**, **static**, or **register**.

The **address** specifier binds a variable to the specified virtual address, specified by a constant. You can use **address** for a variable definition only; therefore, you cannot use it with **typedef** or **extern**. The **address** specifier is useful for referencing objects at fixed lo-

cations in the address space (such as device registers, the PEB page, or certain system data structures). Typically, the compiler generates absolute addressing modes when accessing such an operand.

Using **address** by itself (without **device** or **volatile**) does not suppress any compiler optimizations. You should use it in conjunction with **device** or **volatile**. The example below associates the variable **peb_page** with the hexadecimal virtual address FF7000.

```
char peb_page #attribute[device, address(0xFF7000)];
```

3.16.6 The section Specifier

The syntax for the **section** specifier is:

[**extern**] *data_type variable_name* **#attribute[section(***name***)]** [*initializer*]

where *name* is the named section in which to place the variable. Note that the **#attribute[section]** modifier is legal only for global declarations. You will receive an error if you attempt to use it with local declarations.

When you compile with **/bin/cc**, the compiler places all uninitialized global declarations in a section of the object file called **.bss**. All initialized global variables are placed in a section called **.data**. This is the standard format for UNIX object files. The **/com/cc** compiler, on the other hand, creates a special named section for each global variable, whether it is initialized or not. By default, the name of the section is the same as the global variable. (You can obtain the **/bin/cc** object file format by compiling with the **-bss** option.)

The **section** specifier enables you to mimic /com/cc behavior when you compile with **/bin/cc**. This is particularly useful for interacting with FORTRAN programs that use common blocks. For example, suppose a FORTRAN program contains the following common block definition:

```
integer*4 first
real*8    second
char*20   third

common    /com_block/ first, second, third
```

These declarations produce a named section called **com_block** in the object file that contains the three variables named **first**, **second**, and **third**. If you want to access these variables from a C program compiled with /bin/cc, you need to use the **section** specifier:

```
typedef struct {
                int    first;
                double second;
                char   third[20];
              } COM_BLOCK;
COM_BLOCK com_block #attribute[section(com_block)];
```

If you are compiling with **/com/cc**, the section **#attribute[section]** modifier is unnecessary because **/com/cc** automatically creates a named section for each global variable. The binder then overlays sections that have the same name. See Chapter 7 for more information about sharing global data with Pascal and FORTRAN programs.

Chapter 4

Code

This chapter describes the statements and operators that make up the action part of a Domain C function.

We provide an overview at the beginning of this chapter. The remainder of the chapter is a Domain C encyclopedia. If you are just beginning to learn C, we suggest you read a good C tutorial textbook before trying to use this chapter.

This overview of Domain C code is divided into the following categories:

- Statements
- Operators
- Type Conversions
- Preprocessor Directives

4.1 Statements

There are many type of C statements—null statements, simple statements, compound statements, branching statements, and looping statements. The following sections briefly describe each of these types.

4.1.1 Null Statement

A **null statement** is simply a semicolon by itself. The null statement is sometimes used as the block of a **for** or **while** loop, when the action is specified in the loop. The following loop, for instance, reads characters from **stdin** until an **EOF** character is encountered:

```
while((c = getchar()) != EOF)
  ;   /* null statement */
```

4.1.2 Simple Statement

A **simple statement** consists of an expression followed by a semicolon. Here are a few examples of simple statements:

```
x = 5;    /* a variable assignment */
x++;      /* a variable increment */
f(x);     /* a function call (see Chapter 5 for details) */
```

4.1.3 Compound Statement or Block

A **compound statement** or **block** has the following format:

```
{
  declaration1
  .
  .
  .
  declarationN

  statement1
  .
  .
  .
  statementN
}
```

That is, a compound statement consists of one or more optional **declarations** followed by one or more optional **statements**. A **declaration** can be any variable or typedef declaration. (Note that such a declaration has block scope.) A **statement** can be any null statement, simple statement, or compound statement. The body of a function is itself a block.

C programmers commonly use compound statements as the body of a loop. In the following example, two statements (an assignment statement and a function call) make up the compound statement:

```
for (x = 1; x < 11; x++)
{
  running_total = running_total + x; /* assignment statement */
  printf("running_total is %d\n", running_total); /* function
                                                     call */
}
```

A right brace } marks the end of a compound statement; do not put a semicolon after this right brace.

4.1.4 Branching Statements

C supports two conditional branching statements—**if**(and **if/else**) and **switch.** The **if** and **if/else** statements test expressions and execute statements depending on the results of the test. The **switch** statement selects among several statements based on constant values. The **case, default,** and **break** keywords are optional elements of a **switch** statement.

C supports two unconditional branching statements—**goto** and **return.** The **goto** statement causes a jump to a **label** (or more specifically, a jump to the first statement following that label). All statements may be preceded by a label. The **return** statement causes an unconditional return to the calling routine. You can optionally use **return** to pass data back to the caller.

4.1.5 Looping Statements

Domain C supports three looping statements—**for, while,** and **do/while.** These statements enable you to iterate through a block of code. Within a loop, you can use the **continue** and **break** statements. The **continue** statement causes a jump to the next iteration of the loop, while **break** transfers control to the first statement following the end of the loop.

4.2 Overview: Operators

Operators are the verbs of the C language that let you calculate values. C's rich set of operators is one of its distinguishing characteristics. The operator symbols are composed of one or more special characters. If an operator consists of more than one character, you should enter the characters without any intervening spaces:

```
x <= y    /* legal expression */
x < = y   /* illegal expression */
```

Each operator takes one or more **operands.** If you think of operators as verbs, then the operands are the subject and object of those verbs.

Domain C supports the following kinds of operators:

- Pointer operators
- Increment and decrement operators
- Cast operators
- **sizeof** operator
- Arithmetic operators
- Comparison (relational) operators
- Bit operators
- Logical operators
- Conditional expression operators
- Comma operator
- Assignment operator

We summarize these operators in this section. For many of the operators, one or more of the operands must be an **lvalue**. An lvalue is an expression that refers to a region of storage that can be manipulated. In other words, an lvalue is any expression that you can use on the left side of an assignment operation. For example, all simple variables, like **int**s and **float**s, are lvalues. An element of an array is also an lvalue; however, an entire array is not. A member of a structure or union is an lvalue; an entire structure or union is not.

4.2.1 Pointer Operators

We begin this overview with a look at the pointer operators:

******ptr_exp*	Dereferences a pointer. That is, it finds the contents stored at the virtual address that *ptr_exp* holds.
ptr->member	Dereferences a *ptr* to a structure or union where *member* is a member of that structure or union.
&*lvalue*	Finds the virtual address where the *lvalue* is stored.

See the "pointer operations" listing later in this chapter for details.

4.2.2 Increment and Decrement Operators

C supports the increment and decrement unary operators listed below.

++lvalue	Increments the current value of *lvalue* before *lvalue* is referenced.
lvalue++	Increments the current value of *lvalue* after *lvalue* has been referenced.
--lvalue	Decrements the current value of *lvalue* before *lvalue* is referenced.
lvalue--	Decrements the current value of *lvalue* after *lvalue* has been referenced.

For details, see the "increment and decrement operators" listing later in this chapter.

4.2.3 Cast Operator

C supports the cast operator which takes the following form:

(data_type)exp	Casts the value of *exp* to a new *data type*.

For details, see the "cast operator" listing later in this chapter.

4.2.4 sizeof Operator

The following list provides an overview of the **sizeof** operator:

sizeof *exp*	Calculates the size (in bytes) of *exp*.
sizeof(*data_type*)	Calculates the size (in bytes) that a variable of this *data_type* takes up in memory.

For details, see the "**sizeof**" listing later in this chapter.

4.2.5 Arithmetic Operators

The following list summarizes all the binary arithmetic operators:

exp1 + *exp2*	Adds *exp1* and *exp2*. An *exp* can be any integer expression or floating–point expression.
exp1 – *exp2*	Subtracts *exp2* from *exp1*. An *exp* can be any integer expression or floating–point expression.
exp1 * *exp2*	Multiplies *exp1* by *exp2*. An *exp* can be any integer expression or floating–point expression.
exp1 / *exp2*	Divides *exp1* by *exp2*. (Can perform integer or real division. If integer division, / operator performs division and truncates result to an integer.)
exp1 % *exp2*	Finds modulo of *exp1* divided by *exp2*. (That is, finds the remainder of an integer division.) An *exp* can be any integer expression.
–*exp*	Negates the value of *exp*. (That is, it multiplies *exp* by –1.) *exp* can be any integer expression or floating–point expression.

For full details on these operators, see the "arithmetic operators" listing later in this chapter.

4.2.6 Comparison (Relational) Operators

Use the following operators to compare two expressions:

exp1 < *exp2*	Evaluates to 1 (true) if *exp1* is less than *exp2*; otherwise, evaluates to 0 (false).
exp1 > *exp2*	Evaluates to 1 if *exp1* is greater than *exp2*; otherwise, evaluates to 0.
exp1 <= *exp2*	Evaluates to 1 if *exp1* is less than or equal to *exp2*; otherwise, evaluates to 0.
exp1 >= *exp2*	Evaluates to 1 if *exp1* is greater than or equal to *exp2*; otherwise, evaluates to 0.
exp1 == *exp2*	Evaluates to 1 if *exp1* is equal to *exp2*; otherwise, evaluates to 0.

exp1 != *exp2*	Evaluates to 1 if *exp1* is not equal to *exp2*; otherwise, evaluates to 0.

For details, see the "relational operators" listing later in this chapter.

4.2.7 Bit Operators

Use operators from the following list to perform bit operations. Note that all operands in this list must be integers.

exp1 << *exp2*	Left shifts the bits in *exp1* by *exp2* positions.
exp1 >> *exp2*	Right shifts the bits in *exp1* by *exp2* positions.
exp1 & *exp2*	Performs a bitwise AND operation.
exp1 ^ *exp2*	Performs a bitwise exclusive OR operation.
exp1 | *exp2*	Performs a bitwise inclusive OR operation.
~*exp*	Calculates the one's-complement of *exp*.

For details, see the "bit operators" listing later in this chapter.

4.2.8 Logical Operators

The following list summarizes the three logical operators:

exp1 && *exp2*	Performs a logical AND on the values of *exp1* and *exp2*. In C, the value 0 is equivalent to false, and any nonzero value is equivalent to true.
exp1 || *exp2*	Performs a logical OR on the values of *exp1* and *exp2*.
!*exp*	Calculates the logical negation of *exp*.

For details, see the "logical operators" listing later in this chapter.

4.2.9 Conditional Expression Operator

C supports the following conditional expression operator:

exp1 ? *exp2* : *exp3*	C shorthand for an **if/else statement**. If *exp1* is true (non-zero), then the result is *exp2*. If *exp1* is false (zero), then the result is *exp3*. Note that the conditional operator has the advantage that it can be used in some places that an **if/else** statement cannot.

For details, see the "conditional expression operator" listing later in this chapter.

4.2.10 Comma Operator

C supports the comma operator as follows:

exp1, *exp2*	Separates two expressions. Note that all expressions return values. The value of a comma operation is equal to the value of *exp2*.

For details, see the "comma operator" listing later in this chapter.

4.2.11 Assignment Operators

Finally, C supports all of the following assignment operators:

lvalue = *exp*	Sets *lvalue* (a variable name) to the value of *exp*.
lvalue += *exp*	Sets *lvalue* equal to *lvalue* + *exp*.
lvalue -= *exp*	Sets *lvalue* equal to *lvalue* - *exp*.
lvalue *= *exp*	Sets *lvalue* equal to *lvalue* * *exp*.
lvalue /= *exp*	Sets *lvalue* equal to *lvalue* / *exp*.
lvalue %= *exp*	Sets *lvalue* equal to *lvalue* % *exp*.
lvalue >>= *exp*	Sets *lvalue* equal to *lvalue* >> *exp*.
lvalue <<= *exp*	Sets *lvalue* equal to *lvalue* << *exp*.
lvalue &= *exp*	Sets *lvalue* equal to *lvalue* & *exp*.
lvalue ^= *exp*	Sets *lvalue* equal to *lvalue* ^ *exp*.

lvalue |= *exp* Sets *lvalue* equal to *lvalue* | *exp*.

See the "assignment operators" listing later in this chapter.

4.2.12 Precedence and Associativity of Operators

All operators have two important properties associated with them called **precedence** and **associativity**. Both properties affect how operands are attached to operators. Operators with higher precedence have their operands **bound**, or **grouped**, to them before operators of lower precedence, regardless of the order in which they appear. For example, the multiplication operator has higher precedence than the addition operator, so the two expressions,

```
2 + 3 * 4
3 * 4 + 2
```

both evaluate to 14—the operands 3 and 4 are grouped with the multiplication operator rather than the addition operator because the multiplication operator has higher precedence. If there were no precedence rules, and the compiler grouped operands to operators in left-to-right order, the first expression,

```
2 + 3 * 4
```

would evaluate to 20. Table 4-1 lists every C operator in order of precedence.

In cases where operators have the same precedence, associativity (sometimes called **binding**) is used to determine the order in which operands are grouped with operators. Grouping occurs in either **right-to-left** or **left-to-right** order, depending on the operator. Right-to-left associativity means that the compiler starts on the right of the expression and works left. Left-to-right associativity means that the compiler starts on the left of the expression and works right. For example, the plus and minus operators have the same precedence and are both left-to-right associative:

```
a + b - c;  /* add a to b, then subtract c */
```

The assignment operator, on the other hand, is right-associative:

```
a = b = c;  /* assign c to b, then assign b to a */
```

4.2.13 Parentheses

The compiler groups operands and operators that appear within parentheses first, so you can use parentheses to specify a particular grouping order. For example:

```
/*  subtract 3 from 2, then multiply that by 4 --
 *  result is -4
 */
(2 - 3) * 4

/*  multiply 3 and 4, then subtract from 2 --
 *  result is -10
 */
2 - (3 * 4)
```

In the second case, the parentheses are unnecessary since multiplication has a higher precedence than addition. Nevertheless, parentheses serve a valuable stylistic function by making an expression more readable, even though they may be redundant from a semantic viewpoint.

In the event of nested parentheses, the compiler groups the expression enclosed by the innermost parentheses first.

4.2.14 Order of Evaluation

An important point to understand is that precedence and associativity have little to do with **order of evaluation**, another important property of expressions. The order of evaluation refers to the actual order in which the compiler evaluates operators. This is independent of the order in which the compiler groups operands to operators. For most operators, the compiler is free to evaluate subexpressions in any order it pleases. It may even reorganize the expression, so long as the reorganization does not affect the final result. For example, given the expression,

```
(2 + 3) * 4
```

the compiler might first add 2 and 3, and then multiply by 4. On the other hand, a compiler is free to reorganize the expression into:

```
(2 * 4) + (3 * 4)
```

since this gives the same result.

The order of evaluation can have a critical impact on expressions that contain side effects. Moreover, reorganization of expressions can sometimes cause overflow conditions.

Table 4–1. Binding and Precedence of Operators

class of operator	operators in that class	binding	precedence
primary	() [] -> .	Left–to–Right	HIGHEST
unary	cast operator sizeof & (address of) * (dereference) – (reverse sign) ~ ! ++ --	Right–to–Left	
multiplicative	* / %	Left–to–Right	
additive	+ –	Left–to–Right	
shift	<< >>	Left–to–Right	
relational	< <= > >=	Left–to–Right	
equality	== !=	Left–to–Right	
bitwise AND	&	Left–to–Right	
bitwise exclusive OR	^	Left–to–Right	
bitwise inclusive OR	|	Left–to–Right	
logical AND	&&	Left–to–Right	
logical OR	||	Left–to–Right	
conditional	? :	Right–to–Left	
assignment	= += -= *= /= %= >>= <<= &= ^= !=	Right–to–Left	
comma	,	Left–to–Right	LOWEST

4.3 Type Conversions

The C language allows you to mix arithmetic types in expressions with few restrictions. For example, you can write:

```
num = 3 * 2.1;
```

even though the expression on the right-hand side of the assignment is a mixture of two types, an **int** and a **double**. Also, the data type of **num** could be any scalar data type except a pointer.

To make sense out of an expression with mixed types, C performs conversions automatically. These **implicit conversions** make the programmer's job easier, but it puts a greater burden on the compiler since it is responsible for reconciling mixed types. This can be dangerous since the compiler may make conversions that you don't expect. For example, the expression,

```
3.0 + 1/2
```

does not evaluate to 3.5 as you might expect. Instead, it evaluates to 3.0 because the value .5 (result of 1/2) is converted to an integer (the fractional part is truncated, leaving a value of zero).

Implicit conversions, sometimes called **quiet conversions** or **automatic conversions**, occur under four circumstances:

1. In assignment statements, the value on the right side of the assignment is converted to the data type of the variable on the left side. These are called **assignment conversions** and are described in the "assignment operators" section of this chapter.

2. Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. An **unsigned char** or **unsigned short** is converted to an **unsigned int**. These are called **integral widening conversions**.

3. In an arithmetic expression, objects are converted to conform to the conversion rules of the operator. These **arithmetic conversions** are described later in this section.

4. In certain situations, arguments to functions are converted. This type of conversion is described in Chapter 5.

As an example of the first type of conversion, suppose **j** is an **int** in the following statement:

```
j = 2.6;
```

Before assigning the **double** constant to **j**, the compiler converts it to an **int**, giving it an integral value of 2. Note that the compiler *truncates* the fractional part rather than rounding to the closest integer.

The second type of implicit conversion, called **integral widening** or **integral promotion**, is almost always invisible.

To understand the third type of implicit conversion, we first need to briefly describe how the compiler processes expressions. When the compiler encounters an expression, it divides it into **subexpressions**, where each subexpression consists of one operator and one or more objects, called **operands**, that are bound to the operator. For example, the expression,

```
-3 / 4 + 2.5
```

contains three operators: –, /, and +. The operand to – is 3; there are two operands to /, –3 and 4; and there are two operands to +, –3/4 and 2.5.

The minus operator is said to be a **unary operator** because it takes just one operand, whereas the division and addition operators are **binary operators**. Each operator has its own rules for operand type agreement, but most binary operators require both operands to have the same type. If the types differ, the compiler converts one of the operands to agree with the other one. To decide which operand to convert, the compiler resorts to the hierarchy of data types shown in Figure 4–1, and converts the "lower" type to the "higher" type. For example:

```
1 + 2.5
```

involves two types, an **int** and a **double**. Before evaluating it, the compiler converts the **int** into a **double** because **double** is higher than **int** in the type hierarchy. The conversion from an **int** to a **double** does not usually affect the result in any way. It is as if the expression were written:

```
1.0 + 2.5
```

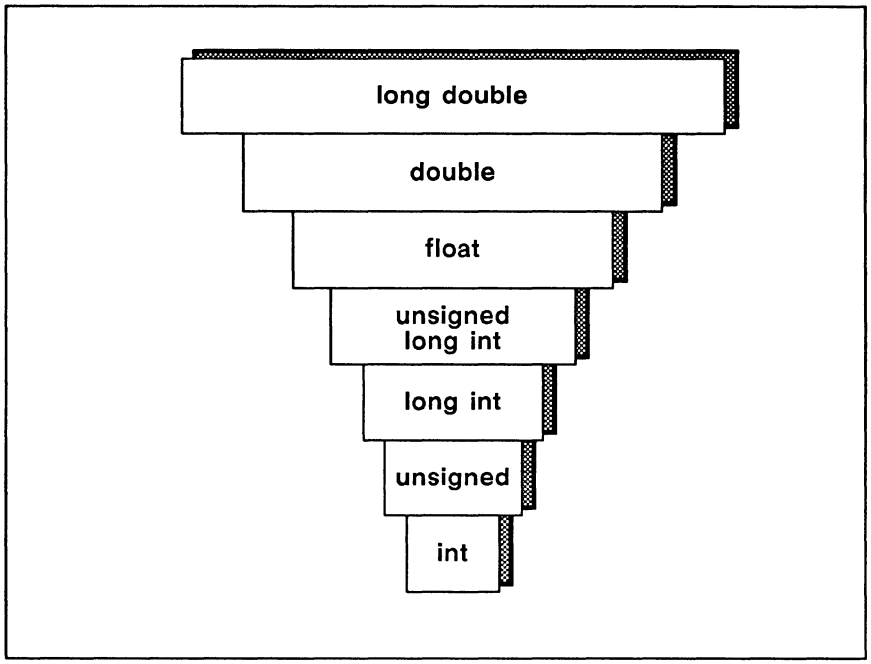


Figure 4–1. Hierarchy of C Scalar Data Types

The rules for implicit conversions in expressions can be summarized as follows. Note that these conversions occur *after* all integral widening conversions have taken place.

- If a pair of operands contains a **long double**, the other value is converted to **long double**.
- Otherwise, if one of the operands is a **double**, the other is converted to **double**.
- Otherwise, if one of the operands is a **float**, the other is converted to **float**.
- Otherwise, if one of the operands is an **unsigned long int**, the other is converted to **unsigned long int**.
- Otherwise, if one of the operands is a **long int**, then the other is converted to **long int**.
- Otherwise, if one of the operands is an **unsigned int**, then the other is converted to **unsigned int**.

In general, most implicit conversions are invisible. They occur without any obvious effect.

4.4 Overview: Preprocessor Directives

The compiler analyzes **preprocessor directives** before analyzing any statements or declarations. The preprocessor directives provide information to the compiler on how the code should be compiled. There is no limit to the number of preprocessor directives that a program can contain. Preprocessor directives (with the exception of **#module**, **#section**, and **#systype**) can appear on any line in a program.

Domain C supports the preprocessor directives shown in Table 4–3. Preprocessor directives always begin with the # character.

In addition to these directives, Domain C supports the predefined macros and names shown in Table 4–2.

Table 4–2. Predefined Macros and Names

Name or Macro	What It Does
defined	A macro that returns 1 if the argument is defined; 0 if the argument is not defined.
systype	A macro that sets the systype environment variable.
__DATE__	A name that expands to the date at compilation time.
__FILE__	A name that expands to the current source filename.
__LINE__	A name that expands to the current line number in the source file.
__TIME__	A name that expands to the time of compilation.
__STDC__	A name that expands to 1 if prototyping is turned on; otherwise it expands to zero.

Table 4-3. Preprocessor Directives

Preprocessor Directive		What It Does
#debug	*	Marks source code for conditional compilation.
#define,#undef		Defines and undefines constants and macros.
#eject	*	Inserts a page break into the listing file.
#elif	*	Same as an **#else** directive followed by an **#if** directive. (The **#elif** directive is support by the UNIX preprocessor (**cpp**) but not by the preprocessor in the Domain C compiler. Therefore, use **#elif** only if you are compiling in a UNIX environment or explicitly specify the **/bin/cc** command.
#if, #ifdef, #ifndef, #else, #endif		Controls conditional compilation.
#include		Loads an include file.
#line		Resets the compiler's knowledge of the current source line number and filename.
#list, #nolist		Enables and disables the listing of source code in the listing file.
#module	*	Changes the internally stored name of the object module.
#section	*	Directs the binder to place instructions and data into named sections rather than the default sections.
#systype	*	Defines the target system on which the program will run.
* Preprocessor directives marked with an asterisk can begin on any column; however, the other preprocessor directives must begin in the very first column of a line.		

4.5 Encyclopedia of Domain C Code

The remainder of this chapter contains an alphabetical listing of all the elements that can make up the action part of a function. Figure 4–2 shows all the listings of C keywords in this encyclopedia, Figure 4–3 provides all the preprocessor directive listings, and Figure 4–4 gives all the other listings.

break	**if**
continue	**return**
do/while	**sizeof**
for	**switch**
goto	**while**

Figure 4–2. Keyword Listings in Encyclopedia

__DATE__ and __TIME__	**__LINE__ and __FILE__**
#debug	**#line**
#define, #undef	**#list**
#eject	**#module**
#if, #ifdef, #ifndef, #else, #endif	**#section**
#include	**__STDC__ and _BFMT__COFF**
	#systype

Figure 4–3. Preprocessor Directive Listings in Encyclopedia

arithmetic operators	expressions
array operations	increment and decrement operators
assignment operators	logical operators
bit operators	pointer operations
cast operations	predefined macros
comma operator	relational operators
conditional expression operator	structure and union operations
enum operations	

Figure 4–4. Other Listings in Encyclopedia

arithmetic operators Operators used to perform arithmetic calculations.

FORMAT

exp1 + *exp2*	Addition
exp1 - *exp2*	Subtraction
exp1 * *exp2*	Multiplication
exp1 / *exp2*	Division
exp1 % *exp2*	Modulo division
-*exp*	Sign reversal

ARGUMENTS

exp	Any constant or variable expression.

DESCRIPTION

The addition, subtraction, and multiplication (+, -, and *) operators perform the usual arithmetic operations in C programs. All of the arithmetic operators (except the unary sign reversal operator) bind from left to right. The operands may be any integral or floating-point value (except for the modulo operator, which accepts only integer operands). The addition and subtraction operators also accept pointer types as operands. Pointer arithmetic is described in the "pointer operations" section of this chapter.

C's modulo operator (%) produces the remainder of integer division and so equals zero if the two numbers divide each other exactly. This can be useful for something like determining whether or not it's a U.S. presidential election year. For example:

```
if (year % 4 == 0)
  printf("This is a U.S. presidential election year.\n");
else
  printf("There will not be a U.S. presidential election this\
year.\n");
```

As required by the ANSI standard, Domain C supports the following relationship between the remainder and division operators:

a equals **a%b** + (**a**/**b**) * **b** for any integer values of **a** and **b**

As with division expressions, the result of a remainder expression is undefined if the right operand is zero.

The additive inverse operator (-) multiplies its sole operand by -1. For example, if **x** is an integer with the value -8, then -**x** evaluates to 8.

Refer to the precedence rules at the beginning of this chapter for information about how these and other operators evaluate with respect to each other.

Bug Alert: Integer Division and Remainder

When both operands of the division operator (/) are integers, the result is an integer. If both operands are positive, and the division is inexact, the fractional part is truncated:

5/2	evaluates to	2
7/2	evaluates to	3
1/3	evaluates to	0

If either operand is negative, however, the compiler is free to round the result either up or down. In accord with the PCC implementation of C, the Domain C compiler always rounds up:

-5/2	evaluates to	-2 (on Apollo machines) but -3 (on some machines)
7/-2	evaluates to	-3 (on Apollo machines) but -4 (on some machines)
-1/-3	evaluates to	0 (on Apollo machines) but -1 (on some machines)

By the same token, the sign of the result of a remainder operation is undefined by the K&R and ANSI standards:

-5 % 2	evaluates to	1 or -1
7 % -4	evaluates to	3 or -3

Domain C makes the sign of the result agree with the sign of the left-hand operand:

-5 % 2	evaluates to	-1 (on Apollo machines)
7 % -4	evaluates to	3 (on Apollo machines)

This is consistent with the PCC implementation.

For portability reasons, you should avoid division and remainder operations with negative numbers since the results can vary from one compiler to another. One way to avoid the sign problem during division is to always cast the operands to **float** or **double**. Even if the result is assigned to an integer, you are guaranteed that the compiler will convert to an integer by truncating the fractional part. For example:

```
/*  If j is an integer, it will be assigned the value -2. */
j = (float) 5 / -2;
```

Although this is a portable solution, it is expensive, since it requires the CPU to perform floating-point arithmetic.

The sign of the remainder is a more difficult problem to circumvent because the operands *must* be integer—you cannot cast them to **float** or **double**. If you always want the sign to be positive, you can use the run-time library **abs()** function, which returns the absolute value of its argument:

```
/*  Ensures that the value assigned to j is positive. */
j = abs(k%m);
```

If the sign of the remainder is important to your program's operations, you should use the run-time library **div()** function, which computes the quotient and the remainder of its two arguments. The sign of both results is determined in a guaranteed and portable manner. (See the description of **div()** in the *SysV Programmer's Reference* manual or the *BSD Programmer's Reference* manual.)

array operations Operations that may be performed with arrays.

DESCRIPTION

Chapter 3 explains how to declare array variables. Here we explain how to use array variables in statements.

You assign a value to an element of an array by specifying an assignment statement of the following form:

array_name[*component_number*] = *value*;

For example, given the following array declaration

```
float    r_array[1000];
```

you can assign the value 5.29 to element 3 with the following statement:

```
r_array[3] = 5.29;
```

Note that the *component_number* must always be an integral value. Consider the following legal and illegal declarations:

```
r_array[3]       = 5.29;   /* legal */
r_array['B']     = 5.29;   /* legal */
r_array[143.5]   = 5.29;   /* illegal */
```

The following program fragment assigns values to an integer array and shows the use of a simple index expression:

```
int i, num[5];
.
.
.
for (i = 0; i < 5; i++)
         num[i] = i;
```

The array **num** can hold five integers, and those five are assigned with a simple **for** loop. Notice that the loop begins its assignments with the zeroth element of the array. All C array subscripts, or indexes, begin at zero (array[0]). Some programming languages always begin at 1 (array[1]), while others allow the programmer to determine the initial subscript value, but C always starts counting at zero. This is important because it means if you create an array of size *n*, no *n*th element is defined. In the example above, **num** has these five elements:

```
num[0]              /*  first element  */
num[1]              /*  second  */
num[2]              /*  third  */
num[3]              /*  fourth  */
num[4]              /*  fifth  */
```

Even though there is no **num[5]** element, the compiler does not complain if you assign something to it (or **num[6]**, or **num[12]**, or whatever), and that fact can create hard-to-find errors. When storing an array value, C looks at the array name and then uses the subscript value to determine the memory offset. No bounds checking occurs, as explained in the "Bug Alert: Walking Off the End of an Array."

Subscripting with enums

Domain C allows you to use an enumerated value as an array index. In the following code fragment, the value 3.14159 is assigned to **array[2]**:

```
{
  enum subscripts { zero, one, two, three, four};
  float array[10];

  array[two] = 3.14159;
}
```

Bug Alert: Walking Off the End of an Array

Unlike some programming languages, C does not require compilers to *check array bounds*. This means that you can attempt to access elements for which no memory has been allocated. The results are unpredictable. Sometimes you will access memory that has been allocated for other variables. Sometimes you will attempt to access special protected areas of memory and your program will abort. Usually this type of error occurs because you are off by one in testing for the end of the array. For example, consider the following program which attempts to initialize every element of an array to zero:

```
main()
{
  int ar[10], j;

  for (j=0; j <= 10; j++)
    ar[j] = 0;
}
```

Since we have declared **ar[]** to hold ten elements, we can validly refer to elements 0 through 9. Our **for** loop, however, has an **off-by-one bug** in it. The loop runs from 0 through 10, so element 10 also gets assigned zero. Since there is no element 10, the compiler overwrites a portion of memory, very likely the portion of memory reserved for **j**. This will produce an infinite loop because **j** will be reset to zero.

Accessing Array Elements Through Pointers

One way to access array elements is to enter the array name followed by a subscript. Another way is through pointers. The declarations,

```
short ar[4];
short *p;
```

create an array of four variables of type **short**, called **ar[0]**, **ar[1]**, **ar[2]**, and **ar[3]**, and a variable named **p** that is a pointer to a **short**. Using the address-of operator (**&**), you can now make the assignment,

```
p = &ar[0];
```

which assigns the address of array element 0 to **p**. If we dereference **p**,

```
*p
```

we get the value of element **ar[0]**.

Until the value of **p** is changed, the expressions **ar[0]** and ***p** refer to the same memory location. Due to the scaled nature of pointer arithmetic, the expression,

```
*(p+3)
```

refers to the same memory contents as:

```
ar[3]
```

In fact, for any integer expression *e*,

```
*(p+e)
```

is the same as:

```
ar[e]
```

This brings us to the first important relationship between arrays and pointers: *Adding an integer to a pointer that points to the beginning of an array, and then dereferencing that expression, is the same as using the integer as a subscript value to the array.*

The second important relationship is that an array name that is not followed by a subscript is interpreted as a pointer to the initial element of the array (except when an array name appears as the operand of the **sizeof** operator). That is, the expressions,

```
ar
```

and

```
&ar[0]
```

are exactly the same. Combining these two relationships, we arrive at the following important equivalence:

ar[n] is the same as *(ar + n)

This relationship is unique to the C language and is one of C's most important features. When the C compiler sees an array name, it translates it into a pointer to the initial element of the array. Then the compiler interprets the subscript as an offset from the **base address** position. For example, the compiler interprets the expression **ar[2]** as a pointer to the first element of **ar**, plus an offset of 2 elements. Due to scaling, the offset determines how many elements to skip, so an offset of 2 means skip two elements. The two expressions

```
ar[2]
*(ar+2)
```

are equivalent. In both cases, **ar** is a pointer to the initial element of the array, and 2 is an offset that tells the compiler to add 2 to the pointer value.

Because of this interrelationship, pointer variables and array names can be used interchangeably to reference array elements. It is important to remember, however, that the values of pointer variables can be changed whereas array names cannot be changed. This is because an array name by itself is not a variable—it refers to the address of the array variable. You cannot change the address of variables. This means that a naked array name (one without a subscript or indirection operator) cannot appear on the left-hand side of an assignment statement. For instance:

```
float ar[5], *p;

p = ar;           /* legal -- same as p= &ar[0]     */
ar = p;           /* illegal -- you may not assign  */
                  /*            to an array address */
&p = ar;          /* illegal -- you may not assign  */
                  /*          to a pointer address  */
ar++;             /* illegal -- you may not         */
                  /*    increment an array address  */
ar[1] = *(p+3);   /* legal -- ar[1] is a variable   */
p++;              /* legal -- you may increment a   */
                  /*          pointer variable      */
++ar[2]           /* legal -- increment element 2 or array */
```

In the above examples, note that scaling allows you to use the increment and decrement operators to point to the next or previous element of an array.

Passing Arrays as Function Arguments

In C, an array name that appears as a function argument is interpreted as the address of the first element of the array. For instance:

```
int main( void )
{
  extern float func( float [] );
  float x, farray[5];
        .
        .
  x = func( farray ); /* Same as func(&farray[0]) */
        .
        .
```

On the receiving side, you need to declare the argument as a pointer to the initial element of an array. There are two ways to do this:

```
func( float *ar )
{
     .
     .
}
```

or

```
func( float ar[] )
{
     .
     .
}
```

The second example declares **ar** to be an array of indeterminate size. You may omit the size specification because no storage is being allocated for the array. (You may include a size for documentation purposes.) The array has already been created in the calling routine, and what is being passed is really a pointer to the first element of the array. Since the compiler knows that array expressions result in pointers to the first element of the array, it converts **ar** into a pointer to a **float**, just like the first declaration. Functionally, therefore, the two versions are equivalent.

The choice of declaring a function argument as an array or a pointer has no effect on the compiler's operation—it is purely for human readability. To the compiler, **ar** simply points to a **float**—it is *not* an array. Because of the pointer–array equivalence, however, you can still access **ar** as if it were an array. But you cannot find out the size of the array in the calling function by using the **sizeof** operator on the argument. For example:

```
/* Program name is "print_size" */
#include <stdio.h>

void print_size( float arg[] )
{
  printf( "The size of arg is: %d\n", sizeof(arg ) );
}

int main( void )
{
  float f_array[10];
  printf( "The size of f_array is: %d\n", sizeof(f_array) );
  print_size( f_array );
}
```

The results of running this program are:

```
The size of f_array is: 40
The size of arg is: 4
```

The variable **f_array** is an array of ten 4-byte **floats**, so the value 40 is its correct size in bytes. The variable **arg**, on the other hand, is converted to a pointer to a **float**. Pointers are four bytes long, so the size of **arg** is 4. Because it is impossible for the called function to deduce the size of the passed array, it is often a good idea to pass the size of the array along with the base address. This enables the receiving function to check array boundaries:

```
#define MAX_SIZE 1000

void foo( f_array, f_array_size );
float f_array[];
int f_array_size;
{
     .
     .
  if (f_array_size > MAX_SIZE)
  {
    printf( "Array too large.\n" );
    exit( 1 );
  }
     .
     .
```

You can obtain the number of elements in an array by dividing the size of the array by the size of each element. On the calling side, you would write:

```
foo( f_array, sizeof(f_array)/sizeof(f_array[0]) );
```

Note that this expression works regardless of the type of element in **f_array[]**.

Returning Arrays from Functions

The **return** statement can pass only one value back to the caller. It may therefore seem impossible to pass an array back to the caller, but it can be done. The trick is to define the called function so that it returns a pointer to the base type of the array. The following example demonstrates this method. In it, we pass in an array of lowercase letters to the function **f()**, and it returns an array of uppercase letters.

```
/* Program name is "returning_arrays".  It demonstrates how a
 * function can return an array to the caller.
 */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* Define a function that returns a pointer to a character */
char *toupper_string( char *arg )
{
  static char result[100];
  int i=0;
  while (*arg)
    result[i++] = toupper( *arg++ );
  return result;  /* pass back the address of the first element
                   * of array 'result'.
                   */
}

int main( void )
{
  char x[100], *px;
  strcpy( x, "hi there" );
  px = toupper_string( x );  /* upon return from the function,
                              * px points to the first element
                              * of array result
                              */
  printf( "%s => %s\n", x, px );
}
```

NOTE: In the preceding example, we declare array **result** as a **static** so that it will not disappear after function invocation. Note though that any dereference of pointer **px** may inadvertently alter the contents of the array, so be careful.

Multidimensional Arrays

An array of arrays is a **multidimensional array** and is declared with consecutive pairs of brackets. To access an element in a multidimensional array, you specify as many subscripts as are necessary.

Consider the following array of arrays:

```
int ar[2][3] = { { 0, 1, 2 },
                 { 3, 4, 5 }
               };
```

The array reference,

```
ar[1][2]
```

is interpreted as

```
*(ar[1] + 2)
```

which is further expanded to:

```
*(*(ar+1)+2)
```

Recall that **ar** is an array of arrays. When *(ar+1) is evaluated, therefore, the 1 is scaled to the size of the object, which in this case is a 3-element array of **int**s (which we assume are four bytes long), and the 2 is scaled to the size of an **int**:

```
*((int *) ((char *)ar + (1*3*4)) + (2*4))
```

We put in the **(char *)** cast to turn off scaling because we have already made the scaling explicit. The **(int *)** cast ensures that we get all four bytes of the integer when we dereference the address. After doing the arithmetic, the expression becomes:

```
*(int *) ((char *)ar + 20 )
```

The value 20 has already been scaled so it represents the number of bytes to skip. If **ar** starts at address 1000 **ar[1][2]** refers to the **int** that begins at address 1014 (in hex), which is 5.

If you specify fewer subscripts than there are dimensions, the result is a pointer to the base type of the array. For example, given the 2-dimensional array declared above, you could make the reference,

```
ar[1]
```

which is the same as:

```
&ar[1][0]
```

The result is a pointer to an **int**.

Passing Multidimensional Arrays as Arguments

To pass a multidimensional array as an argument, you pass the array name as you would a single-dimension array. The value passed is a pointer to the initial element of the array,

but in this case the initial element is itself an array. On the receiving side, you must declare the argument appropriately, as shown in the following example.

```
f1()
{
  int ar[5][6][7];
        .
        .
  f2( ar );
        .
        .
}

f2( received_arg )
int received_arg[][6][7];
{
        .
        .
}
```

Again, you may omit the size of the array being passed, but you must specify the size of each element in the array. Most compilers don't check bounds, so it doesn't really matter whether you specify the first size. For example, the compiler would interpret the declaration of **received_arg** as if it had been written:

```
int (*received_arg)[6][7];
```

Another way to pass multidimensional arrays is to explicitly pass a pointer to the first element, and pass the dimensions of the array as additional arguments. In our example, what gets passed is actually a pointer to a pointer to a pointer to an **int**.

```
f1()
{
  int ar[5][6][7];
        .
        .
  f2( ar, 5, 6, 7 );
        .
        .

f2( received_arg, dim1, dim2, dim3 )
int ***received_arg;
int dim1, dim2, dim3;
{
        .
        .
}
```

The advantage of this approach is that you need not know ahead of time the shape of the multidimensional array. The disadvantage is that you need to manually perform the indexing arithmetic to access an element. For example, to access **ar[x][y][z]** in **f2()**, you would need to write:

```
*((int *)received_arg + x*dim3*dim2 + y*dim2 + z)
```

Note that we need to cast **received_arg** to a pointer to an **int** because we are performing our own scaling. Although this method requires considerably more work on the programmer's part, it gives more flexibility to **f2()** since it can accept 3-dimensional arrays of any size and shape. Moreover, it is possible to define a macro that simplifies the indexing expression.

Bug Alert: Referencing Elements in a Multidimensional Array

One of the most common mistakes made by beginning C programmers—especially those familiar with another programming language—is to use a comma to separate subscripts,

```
ar[1,2] = 0;  /* Legal, but probably wrong */
```

instead of:

```
ar[1][2] = 0; /* Correct */
```

The comma notation is used in some other languages, such as FORTRAN and Pascal. In C, however, this notation has a very different meaning because the comma is a C operator in its own right. The first statement above causes the compiler to evaluate the expression **1** and discard the result; then evaluate the expression **2**. The result of a comma expression is the value of the rightmost operand, so the value 2 becomes the subscript to **ar**. As a result, the array reference accesses element 2 of **ar**.

If **ar** is a 2-dimensional array of **int**s, the type of **ar[2]** is a pointer to an **int**, so this mistake will produce a type incompatibility error. This can be misleading since the real mistake is using a comma instead of brackets.

EXAMPLE

```
/*  Program name is "bubble_sort".  It sorts an array of
 *  ints in ascending order using the bubble sort algorithm.
 */
#define FALSE 0
#define TRUE 1
#include <stdio.h>

void bubble_sort( int list[], int list_size )
{
  int j, k, temp, sorted = FALSE;
  while ( sorted )
  {
    sorted = TRUE;   /* assume list is sorted */
/* Print loop -- not part of bubble sort algorithm */
      for (k = 0; k < list_size; k++)
        printf( "%d\t", list[k] );
      printf( "\n" );
/* End of print loop */
    for (j = 0; j < list_size -1; j++)
    {
      if (list[j] > list[j+1])
      {
        temp = list[j];
        list[j] = list[j+1];
        list[j+1] = temp;
/*  At least 1 element is out of order
 */
        sorted = FALSE;
      }
    } /* end of for loop */
  } /* end of while loop */
}

int main( void )
{
  int i;
  static int list[] = { 13, 56, 23, 1, 89, 58,
                        20, 125, 86, 3};
  bubble_sort( list, sizeof(list)/sizeof(list[0]));
  exit( 0 );
}
```

The function accepts two parameters, a pointer to the first element of an array of **int**s and an **int** representing the size of the array.

The following program calls **bubble_sort**() with a 10-element array.

```
int main( void )
{
  int i;
  static int list[] = { 13, 56, 23, 1, 89, 58,
                        20, 125, 86, 3};

  bubble_sort( list, sizeof(list)/sizeof(list[0]));
  exit( 0 );
}
```

USING THIS EXAMPLE

Program execution results in the following output:

```
13    56    23    1     89    58    20    125   86    3
13    23    1     56    58    20    89    86    3     125
13    1     23    56    20    58    86    3     89    125
1     13    23    20    56    58    3     86    89    125
1     13    20    23    56    3     58    86    89    125
1     13    20    23    3     56    58    86    89    125
1     13    20    3     23    56    58    86    89    125
1     13    3     20    23    56    58    86    89    125
1     3     13    20    23    56    58    86    89    125
```

The bubble sort is not very efficient, but it's a simple algorithm that illustrates array manipulation. The standard run–time library contains a much more efficient sorting function called **qsort()**, which is described in the *SysV Programmer's Reference* manual and the *BSD Programmer's Reference* manual.

assignment operators Assign new values to variables.

FORMAT

lvalue = *exp*	Simple assignment
lvalue += *exp*	Addition and assignment
lvalue -= *exp*	Subtraction and assignment
lvalue *= *exp*	Multiplication and assignment
lvalue /= *exp*	Division and assignment
lvalue %= *exp*	Modulo division and assignment
lvalue <<= *exp*	Left shift and assignment
lvalue >>= *exp*	Right shift and assignment
lvalue &= *exp*	Bitwise AND and assignment
lvalue ^= *exp*	Bitwise XOR and assignment
lvalue |= *exp*	Bitwise OR and assignment

ARGUMENTS

lvalue	Any lvalue.
exp	Any legal expression.

DESCRIPTION

The = is the fundamental assignment operator in C. The other assignment operators provide shorthand ways to represent common variable assignments. We begin with a discussion of =.

The Assignment (=) Operator

When C sees an equal sign, it processes the statement on the right side of the sign and assigns the result to the variable on the left side. For example:

```
x = 3;    /*  assigns the value 3 to variable x */

x = y;    /*  assigns the value of y to x  */

x = (y*z); /*  performs the multiplication and assigns
            *  the result to x
            */
```

An assignment expression itself has a value, which is the same value that is assigned to the left-hand operand.

The assign operator has right-to-left associativity, so the expression,

```
a = b = c = d = 1;
```

is interpreted as:

```
(a = (b = (c = (d = 1))));
```

First 1 is assigned to **d**, then **d** is assigned to **c**, then **c** is assigned to **b**, and finally, **b** is assigned to **a**. The value of the entire expression is 1. This is a convenient syntax for assigning the same value to more than one variable. Note, however, that each assignment may cause quiet conversions, so,

```
int j;
double f;
f = j = 3.5;
```

assigns the truncated value 3 to both **f** and **j**. On the other hand,

```
j = f = 3.5;
```

assigns 3.5 to **f** and 3 to **j**.

The Other Assignment Operators

C's assignment operators provide a handy way to avoid some keystrokes. Any statement in which the left-hand side of the equation is repeated on the right is a candidate for an assignment operator. If you have a statement like this:

```
i = i + 10;
```

you can use the assignment operator format to shorten the statement to:

```
i += 10;
```

In other words, any statement of the form

```
var = var op exp;    /*  traditional form */
```

can be represented in the following shorthand form:

```
var op= exp;         /*  shorthand form */
```

The only internal difference between the two forms is that **var** is evaluated only once in the shorthand form. Most of the time this is not important; however, it is important when the left-hand operand contains side effects, as in the following example:

```
int  *ip;

*ip++  += 1;           /* These two statements produce */
*ip++  =  *ip++  + 1;  /* different results. */
```

The second statement is ambiguous because C does not specify which assignment operand is evaluated first. See Section 4.2.14 for more information concerning order of evaluation.

Assignment Operators in Older C Compilers

Some older C compilers accept assignment operators written with the equal sign first (for example, =+ instead of +=). When the Domain C compiler encounters such an old-style operator, it processes it as if the two signs were reversed, and issues a warning message.

Also, some compilers accept a space between the two signs. In those compilers, something like

```
+ =
```

is interpreted as

```
+=
```

Since this can lead to ambiguous expressions, the Domain C compiler forbids the space between the operator and the equal sign.

Assignment Type Conversions

Whenever you assign a value to a variable, the value is converted to the variable's data type if possible. In the example below, for instance, the floating-point constant **3.5** is converted to an **int** so that **i** gets the integer value 3.

```
main()
{
   int i;
   i = 3.5;
}
```

Unlike arithmetic conversions, which always expand the datum, assignment conversions can shorten the datum and therefore affect its value. For example, suppose **c** is a **char**, and you make the assignment:

```
c = 882;
```

The binary representation of 882 is:

```
00000011 01110010
```

It requires two bytes of storage, but the variable **c** has only one byte allocated for it, so the two upper bits don't get assigned to **c**. This is known as **overflow** and the result is not defined by the ANSI and K&R standards for signed types. Domain C simply ignores the extra byte, so **c** would be assigned the right-most byte:

```
01110010
```

This would erroneously give **c** the value of 114. The principle illustrated for **char**s also applies to **short**s, **int**s, and **long int**s. For **unsigned** types, however, C has well-defined rules for dealing with overflow conditions. When an integer value **x** is converted to a smaller unsigned integer type, the result is the non-negative remainder of

```
x / (U_MAX+1)
```

where **U_MAX** is the largest number that can be represented in the shorter unsigned type. For example, if **j** is an **unsigned short**, which is two bytes, then the assignment

```
j = 71124;
```

assigns to **j** the remainder of:

```
71124 / (65535+1)
```

The remainder is 5588. Note that for non-negative numbers, and for negative numbers represented in two's complement notation, this is the same result that you would obtain by ignoring the extra bytes.

It is perfectly legal to assign an integer value to a floating-point variable. In this case, the integer value is implicitly converted to a floating-point type. If the floating-point type is capable of representing the integer, there is no change in value. If **f** is a **double**, the assignment

```
f = 10;
```

is executed as if it had been written:

```
f = 10.0;
```

This conversion is invisible. There are cases, however, where a floating-point type is not capable of exactly representing all integer values. Even though the range of floating-point values is generally greater than the range of integer values, the precision may not be as good for large numbers. In these instances, conversion of an integer to a floating-point value may result in a loss of precision. Consider the following example:

```
#include <stdio.h>

main()
{
  long int j = 2147483600;
  float x;

  x = j;
  printf( "j is %d\nx is %10f\n", j, x );
  exit( 0 );
}
```

If you compile this program with the **-nopt** switch to ensure that **x** is not stored in a register, and then execute it, you get:

```
j is 2147483600
x is 2147483648.000000
```

The most risky mixture of integer and floating-point values is the case where a floating-point value is assigned to an integer variable. First, the fractional part is discarded. Then, if the resulting integer can fit in the integer variable, the assignment is made. In the following statement, assuming **j** is an **int**, the **double** value 2.5 is converted to the **int** value 2 before it is assigned.

```
j = 2.5;
```

This causes a loss of precision which could have a dramatic impact on your program. The same truncation process occurs for negative values. After the assignment,

```
j = -5.8;
```

the value of **j** is -5.

An equally serious situation occurs when the floating-point value cannot fit in an integer. For example:

```
j = 999999999999.0
```

This causes an overflow condition which will produce unpredictable results if it is not caught by the compiler. As a general rule, it is a good idea to keep floating-point and integer values separate unless you have a good reason for mixing them.

As is the case with assigning floating-point values to integer variables, there are also potential problems when assigning **double** values to **float** variables There are two potential problems: loss of precision and an overflow condition. In Domain C a **double** can represent approximately 16 decimal places, and a **float** can only represent 7 decimal places. If **f** is a **float** variable, and you make the assignment,

```
f = 1.0123456789
```

the computer rounds the **double** constant value before assigning it to **f**. The value actually assigned to **f**, therefore, will be 1.012346 (Domain C always rounds toward zero). The following example shows rounding due to conversions.

```
/* Program name is "float_rounding".  It show how double values
 * can be rounded when assigned to a float.
 */
#include <stdio.h>

int main( void )
{
  float f32;
  double f64;
  int i;
  for (i=1, f64=0; i < 1000; ++i)
    f64 += 1.0/i;

  f32 = f64;
  printf( "Value of f64: %1.7f\n", f64 );
  printf( "Value of f32: %1.7f\n", f32 );
}
```

The output is:

```
Value of f64: 7.4844709
Value of f32: 7.4844708
```

A more serious problem occurs when the value being assigned is too large to be represented in the variable. For example, the largest positive number that can be represented by a **float** is approximately 2e38. What happens if you try to execute the following assignment?

```
f = 2e40;
```

The behavior is not defined by the K&R or ANSI standards. In this simple case, the compiler will recognize the problem and report a compile-time error. In other instances, however, a run-time error could result.

EXAMPLE

```
/*  Following are examples of each assignment operator. In each
 *  case, x = 5 and y = 2 before the statement is executed.
 */

x = y;       ⇒       x = 2
x += y + 1;  ⇒       x = 8
x -= y * 3;  ⇒       x = -1
x *= y + 1;  ⇒       x = 15
x /= y;      ⇒       x = 2
x %= y;      ⇒       x = 1
x ≪= y;      ⇒       x = 20
x ≫= y;      ⇒       x = 1
x &= y;      ⇒       x = 0
x ^= y;      ⇒       x = 7
x |= y;      ⇒       x = 7
x = y = 1    ⇒       x = 1, y = 1
```

_BFMT__COFF	Refer to the **__STDC__** listing later in this chapter.

bit operators Access specific bits in an object.

FORMAT

exp1 << *exp2*	Left shifts (logical shift) the bits in *exp1* by *exp2* positions
exp1 >> *exp2*	Right shifts (logical or arithmetic shift) the bits in *exp1* by *exp2* positions
exp1 & *exp2*	Performs a bitwise AND operation
exp1 ^ *exp2*	Performs a bitwise OR operation
exp1 | *exp2*	Performs a bitwise inclusive OR operation
~exp1	Performs a bitwise negation (one's complement) operation

ARGUMENTS

exp1 Any integer expression.

exp2 Any integer expression.

DESCRIPTION

Domain C supports the usual six bit operators, which we group for descriptive purposes into shift operators and logical operators.

Bit Shift Operators

The << and >> operators shift an integer left or right respectively. The operands must have integer type, and all automatic promotions are performed for each operand. For example, the following program fragment

```
short int   to_the_left = 53, to_the_right = 53;
short int   left_shifted_result, right_shifted_result;

left_shifted_result = to_the_left << 2;
right_shifted_result  = to_the_right >> 2;
```

sets **left_shifted_result** to 212 and **right_shifted_result** to 13. The results are clearer in binary:

```
     base 2            base 10
0000000000110101         53
0000000011010100        212  /* 53 shifted left 2 bits  */
0000000000001101         13  /* 53 shifted right 2 bits */
```

Shifting to the left is equivalent to multiplying by powers of two.

$$x << y \quad \text{is equivalent to} \quad x * 2^{y}$$

Shifting non–negative integers to the right is equivalent to dividing by powers of two:

$$x >> y \quad \text{is equivalent to} \quad x / 2^{y}$$

The << operator always fills the vacated rightmost bits with zeros. If *exp1* is unsigned, the >> operator fills the vacated leftmost bits with zeros. If *exp1* is signed, then >> fills the leftmost bits with ones (if the sign bit is 1) and zeros (if the sign bit is 0). In other words, if *exp1* is signed, the two bit shift operators preserve its sign.

> **NOTE:** Not all compilers preserve the sign bit when doing bit shift operations on signed integers. The K&R and ANSI standards make this behavior implementation–defined. Domain C is consistent with the PCC implementation of C.

Make sure that the right operand is not larger than the size of the object being shifted. For example, the following produces unpredictable and nonportable results because **int**s have fewer than 50 bits:

```
10 >> 50
```

You will also get nonportable results if the shift count (the second operand) is a negative value.

Bit Logical Operators

The logical bitwise operators are similar to the Boolean operators, except that they operate on every bit in the operand(s). For instance, the bitwise AND operator (**&**) compares each bit of the left operand to the corresponding bit in the right operand. If both bits are one, a one is placed at that bit position in the result. Otherwise, a zero is placed at that bit position.

The four logical operators perform logical operations on a bit–by–bit level using the following truth tables:

& AND

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	0
1	0	0
1	1	1

| Inlusive OR

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	1
1	0	1
1	1	1

^ Exclusive OR

bit x of op1	bit x of op2	bit x of result
0	0	0
0	1	1
1	0	1
1	1	0

~ Bitwise Complement

bit x of op2	bit x of result
0	0
0	1

Figure 4–5. Bitwise Operators

Table 4–4 shows some examples of the bitwise AND operator.

Table 4–4. The Bitwise AND Operator

Expression	Hexadecimal Value	Binary Representation	
9430	0x24D6	00100100	11010110
5722	0x165A	00010110	01011010
9430 & 5722	0x0452	00000100	01010010

The bitwise inclusive OR operator (|) places a 1 in the resulting value's bit position if either operand has a bit set at the position (see Table 4–5).

Table 4–5. Examples Using the Bitwise Inclusive OR Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 | 5722	0x36DE	00110110 11011110

The bitwise exclusive OR (XOR) operator (^) sets a bit in the resulting value's bit position if either operand (but not both) has a bit set at the position (see Table 4–6).

Table 4–6. Example Using the XOR Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 ^ 5722	0x328C	00110010 10001100

The bitwise complement operator (~) reverses each bit in the operand (see Table 4–7).

Table 4–7. Example Using the Bitwise Complement Operator

Expression	Hexadecimal Value	Binary Representation
9430	0x24d6	00100100 11010110
~9430	0xdb29	11011011 00101001

---

break Provides an early exit from **for, while,** and **do/while** loops and from **switch** statements.

FORMAT

break;

DESCRIPTION

There are times when it is convenient to be able to exit from a loop without testing a condition at the top or bottom. The **break** statement allows you to exit immediately from the **for, while,** or **do/while** loop that encloses it. Execution resumes at the first statement after the end of the loop.

The **break** statement is also used to exit from **switch** statements. For more information on that use of **break,** see **switch** later in this encyclopedia.

EXAMPLE

```
/*  Program name is "break_example". This program finds what
 * number day (out of 365) a user-supplied date is in a year.
 * Leap years are ignored.
 */
#include <stdio.h>

int main( void )
{
  int i, month_num, day, tot_days;
  static int m[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30,
                      31, 30, 31};
  char answer = 'y';
  printf ("\n");

/*  The program asks for a month and day and then checks to see
 *  if they are valid. If not, the break statement terminates
 *  the do/while loop. Otherwise, the number day is computed
 *  and printed.
 */
  while ((answer != 'n') && (answer != 'N'))
  {
    printf( "Enter the month and day separated by a space: " );
    scanf ( "%d %d", &month_num, &day );
    fflush( stdin );
    if (month_num > 12 || day > m[month_num])
    {
      printf ( "You entered an invalid date\n" );
      break;
    }       /*  end if  */
    tot_days = 0;
    for (i = 1; i < month_num; i++)
      tot_days += m[i];
    tot_days += day;
    printf( "The date you entered is number %d of the year.\n",
             tot_days );
    printf ( "Again? " );
    scanf ("%c", &answer);
    fflush( stdin );
  }           /*  end while  */
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter the month and day separated by a space: 7 13
The date you entered is number 194 of the year.
Again? y
Enter the month and day separated by a space: 19 24
You entered an invalid date
```

cast operations Convert a value to another data type.

FORMAT

(*data_type*) *exp*

ARGUMENTS

data_type Any scalar data type including a scalar data type created through a *typedef* statement. *data_type* cannot be an aggregate type, but it can be a pointer to an aggregate type.

exp Any scalar expression.

DESCRIPTION

To **cast** a value means to explicitly convert it to another data type. For example, given the following two definitions:

```
int    y = 5;
float  x;
```

the following cast operation casts the value of y to **float**:

```
x = (float) y;       /* x now equals 5.0 */
```

Here are four more casts (assume that **j** is a scalar variable):

```
i = (float) j;   /* Cast j's value to float */
i = (char *)j;   /* Cast j's value to a pointer to a char */
i = ((int *)())j;/* Cast j's value to a pointer to a function
                  * returning an int
                  */
i = (float) (double) j; /* Cast j's value first to a double
                         * and then to a float
                         */
```

It is important to note that if *exp* is a variable, a cast does not change this variable's data type; it only changes the type of the variable's value for that one expression. For instance, in the preceding casting examples, the cast does not produce any permanent effect on variable **j**.

There are no restrictions on casting from one scalar type to another, except that you may not cast a **void** object to any other type. You should be careful when casting integers to

pointers. If the integer value does not represent a valid address, the results are unpredictable.

The type specifier that makes up the cast expression is called an **abstract declarator**. The rules for composing abstract declarators are described in Chapter 3.

Casting Integers to Other Integers

It is possible to cast one integer into an integer of a different size and to convert a floating-point value, enumeration value or pointer to an integer. Conversions from one type of integer to another fall into five cases (A–E) as shown in Table 4–8. Each of these conversions is described in the following sections.

Table 4–8. Integer Conversions

	Converted Type					
Original Type	char	short	int	unsigned char	unsigned short	unsigned int
char	A	B	B	D	E	E
short	C	A	B	C	D	E
int (long)	C	C	A	C	C	D
unsigned char	D	B	B	A	B	B
unsigned short	C	D	B	C	A	B
unsigned int	C	C	D	C	C	A

CASE A: Trivial Conversions

It is legal to "convert" a value to its current type by casting it, but this conversion has no effect.

CASE B: Integer Widening

Casting an integer to a larger size is fairly straightforward. The value remains the same but the storage area is widened. The compiler preserves the sign of the original value by filling the new leftmost bits with ones if the value is negative or with zeros if the value is positive. When converting to an unsigned integer, the value is always positive so the new bits are always filled with zeros. The following table illustrates this principle.

```
                      hex      dec
char i =               37       55
(short) i  =>        0037       55
(int) i    =>    00000037       55

char j =               c3      -61
(short) j  =>        ffc3      -61
(int) j    =>    ffffffc3      -61

unsigned char k =      37       55
(short) k  =>        0037       55
(int) k    =>    00000037       55
```

CASE C: Casting Integers to a Smaller Type

When an **int** value is cast to a narrower type (**short** or **char**), the excess bits on the left are discarded. The same is true when a **short** is cast to a **char**. For instance, if an **int** is cast to a **short**, the 16 leftmost bits are truncated. The following table of values illustrates these conversions.

```
                                    hex          dec
signed long int     i = cf34bf1    217271281

(signed short int)i    =>  4bf1        19441
(signed char)i         =>    f1          -15
(unsigned char)i       =>    f1          241
```

Note that if, after casting to a signed type, the leftmost bit is 1, then the number is negative. However, if you cast to an unsigned type and after the shortening the leftmost bit is 1, then that 1 is part of the value (not the sign bit).

CASE D: Casting from Signed to Unsigned, and Vice Versa

When the orginal type and the converted type are the same size, a representation change is necessary. That is, the internal representation of the value remains the same, but the sign bit is interpreted differently by the compiler. For instance:

```
                             hex             dec         hex      dec
signed int  i     =     ffffca9            -855     0000f2a1    62113

(unsigned int)i  =>     ffffca9      4294966441     0000f2a1    62113
```

The hexadecimal notation shows that the numbers are the same internally, but the decimal notation shows that the compiler interprets them differently.

CASE E: Casting Signed to Unsigned and Widening

This case is equivalent to performing two conversions in succession. First, the value is converted to the **signed** widened type as described in case B, and then it is converted to

signed as described in case D. In the table below, note that the new leftmost bits are filled with ones to preserve negativeness even though the final value is unsigned.

```
                                  hex          dec
signed short int    i =          ff55         -171

(unsigned long int)i =>        fffff55   4294967125
```

Casting Floating-Point Values to Integers

Casting floating-point values to integers may produce useless values if an overflow condition occurs. The conversion is made simply by truncating the fractional part of the number. For example, the floating-point value 3.712 is converted to the integer 3 and the floating-point value -504.2 is converted to -504.

Here are some more examples:

```
float f = 3.700, f2 = -502.2, f3 = 7.35e9;

(int)f           => 3
(unsigned int)f  => 3
(char)f          => 3

(int)f2          => -502       in decimal    fffffe0a in hex
(unsigned int)f2 => 4294966794 in decimal or fffffe0a in hex
(char)f2         => 10         in decimal          0a in hex

(int)f3          => run-time error
(unsigned int)f3 => run-time error
(char)f3         => run-time error
```

Note that converting a large **float** to a **char** produces unpredictable results if the rounded value cannot fit in one byte. If the value cannot fit in four bytes, the run-time system issues an overflow error.

Casting Pointers to Integers

Pointers are treated like **unsigned ints** and obey the same conversion rules.

Casting Enumerated Values to Integers

When you cast an enumerated expression, the conversion goes through two steps. First, the enumerated value is converted to an **int** and then the **int** is converted to the final target data type. Note that the sign is preserved during these conversions.

Casting Double to Float and Vice Versa

When you cast a **float** up to a **double**, the system extends the number's precision without changing its true value. However, when you cast a **double** down to a **float**, the system shrinks the number's precision and this shrinking may change the number's value due to rounding. The rounding generally occurs on the sixth or seventh decimal digit. Also, when you cast down from **double** to **float**, you run the risk of causing a run-time overflow error caused by a **double** that is too big or too small to fit within the confines of a **float**.

Casting Pointers to Pointers

You may cast a pointer of one type to a pointer to any other type. For example:

```
int *int_p;
float *float_p;
struct S *str_p;
extern foo( struct T * );
     .
     .
int_p = (int *) float_p;
float_p = (float *) str_p;
foo( (struct T *) str_p );
```

The cast is required whenever you assign a pointer value to a pointer variable that has a different base type, and when you pass a pointer value as a parameter to a function that has been prototyped with a different pointer type. The only exception to this rule concerns generic pointers (pointers to **void**). You may assign any pointer value to a generic pointer without casting. See Chapter 3 for more information about generic pointers.

comma operator Separates two expressions and returns the value of the latter.

FORMAT

exp1, exp2

ARGUMENTS

exp1 Any expression.

exp2 Any expression.

DESCRIPTION

Use the comma operator to separate two expressions that are to be evaluated one right after the other. The comma operator is popular within **for** loops, as demonstrated by the following example:

```
for (i = 10, j = 4; (i * j) < n; i++, j++);
```

In the preceding example, the comma operator allows you to initialize both **i** and **j** at the beginning of the loop. The comma operator also allows you to increment **i** and **j** together.

Note that all expressions return values. (See the "expressions" listing in this chapter for details.) When using a comma operator, the expression returns the value of the rightmost expression. For example, the following statement sets variable **j** to **2**:

```
j = (x = 1, y = 2);
```

Note, however, that assignments such as these are considered poor programming style. You should confine use of the comma operator to **for** loops.

conditional expression operator Alternative to **if...else** statement constructions.

FORMAT

exp1 ? *exp2* : *exp3*

ARGUMENTS

exp1	Any expression.
exp2	Any expression.
exp3	Any expression.

DESCRIPTION

The conditional expression construction provides a shorthand way of coding an **if...else** condition. The syntax described above is equivalent to:

```
if (exp1)
  exp2;
else
  exp3;
```

When a conditional expression is executed, *exp1* is evaluated first. If it is true (that is, nonzero) *exp2* is evaluated and its result is the value of the conditional expression. If *exp1* is false, *exp3* is evaluated and its result is the value of the conditional expression.

There is no requirement that you put parentheses around the *exp1* portion of the conditional expression, but doing so will improve your code's readability.

Both *exp2* and *exp3* must be assignment-compatible. If *exp2* and *exp3* are pointers to different types, then the compiler issues a warning. The value of a conditional expression is either *expr2* or *expr3*, whichever is selected. Note that the other expression is not evaluated. The type of the result is the type that would be produced if *exp2* and *exp3* were mixed in an expression. For instance, if *exp2* is a **char** and *exp3* is a **double**, the result type will be **double** regardless of whether *exp2* or *exp3* is selected.

EXAMPLE

```
/*  Program name is "conditional_exp_op_example"
 *  This program reads four user-input numbers, adds
 *  them together and prints the total. It then uses
 *  the conditional expression to determine whether
 *  the user wants to continue. If the string answer
 *  is 'y' or 'Y', a value of 1 (true) is assigned to
 *  again. If the answer is anything else, a value of
 *  0 (false) is assigned.
 */
#include <stdio.h>

int main( void )
{
  int a, b, c, d, again, total;
  char answer;

  printf ("\n");
  again = 1;
  while (again)
  {
    total = 0;
    printf ("Enter four numbers -- separated by spaces -- that\
you want added together: ");
    scanf ("%d %d %d %d", &a, &b, &c, &d);
    fflush( stdin );
    total = a + b + c + d;
    printf ("\nThe total is: %d\n", total);
    printf ("Do you want to continue ? ");
    scanf ("%c", &answer);
    again = (answer == 'y' || answer == 'Y') ? 1 : 0;
  }     /*  end while  */
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter four numbers -- separated by spaces -- that you
want added together: 20 30 40 50

The total is: 140
Do you want to continue ? y
Enter four numbers -- separated by spaces -- that you
want added together: 1 2 3 4

The total is:
Do you want to continue ? n
```

continue Causes the next iteration of the enclosing **for**, **while**, or **do/while** loop to begin immediately.

FORMAT

continue;

DESCRIPTION

Continue halts execution of its enclosing **for**, **while**, or **do/while** loop and skips to the next iteration of the loop. In the **while** and **do/while**, this means the expression is tested immediately, and in the **for** loop, the third expression (if present) is evaluated.

EXAMPLE

```
/* Program name is "continue_example".  This program
 * reads a file of student names and test scores and
 * computes each student's average grade. However,
 * the instructor has decided to drop the score from
 * the third test because she discovered someone had
 * found and distributed the answer sheet. So the for
 * loop includes a continue statement that tells it to
 *  read over this test's score, excluding it from the
 * averaging calculations.
 */
#include <stdio.h>

int main( void )
{
  int test_score, tot_score, i;
  float average;
  FILE *fp;
  char fname[10], lname[15];

  fp = fopen( "grades_data", "r" );
  printf ( "\n\n" );
  while (!feof( fp ))          /* while not end of file */
  {
    tot_score = 0;
    fscanf( fp, "%s %s", fname, lname );
    printf( "\nStudent's name: %s %s\nGrades: ", fname,
                                                  lname );
    for (i = 0; i < 5; i++)
    {
      fscanf( fp, "%d", &test_score );
      printf( "%d ", test_score );
      if (i == 2)        /*  leave out this test score  */
        continue;
      tot_score += test_score;
    }                  /*  end for i  */
    fscanf( fp, "\n" );  /*  read end-of-line at end of  */
                         /*  each student's data         */
    average = tot_score/4.0;
    printf( "\nAverage test score: %4.1f\n", average );
  }                      /*  end while  */
  fclose( fp );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Student's name: Barry Quigley
Grades: 85 91 88 100 75
Average test score: 87.8

Student's name: Pepper Rosenberg
Grades: 91 76 88 92 88
Average test score: 86.8

Student's name: Sue Connell
Grades: 95 93 91 92 89
Average test score: 92.3
```

__DATE__ and __TIME__ (predefined symbols)	Expands to the date and time of compilation.

FORMAT

__DATE__ **__TIME__**	Note that there are two underscores before and two underscores after each of these preprocessor symbols

DESCRIPTION

The preprocessor recognizes these special predefined symbols and replaces their occurrences with the following:

__DATE__	Expands to a string representing the date of program compilation.
__TIME__	Expands to a string representing the time of program compilation.

EXAMPLE

The **__DATE__** and **__TIME__** macros are useful for recording the date and time a file was last compiled. For instance:

```
/* Program name is "date_and_time_example". */

void print_version( void )
{
  printf( "This utility last compiled on %s at %s\n",
           __DATE__, __TIME__ );
}

int main( void )
{
  print_version();
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
$ date_and_time_example.bin
This utility last compiled on Nov 16 1987 at 17:34:12
```

#debug (preprocessor directive) Marks source code for conditional compilation. (Domain Extension)

FORMAT

#debug *a_line_of_source_code*

ARGUMENT

a_line_of_source_code	Any line of source code.

DESCRIPTION

Domain C provides the **#debug** preprocessor control line, which marks source code for conditional compilation. If you compile with the **-cond** compiler option (explained in Chapter 6), lines prefixed with **#debug** are compiled. If you compile with the **-ncond** switch (which is the default), then lines prefixed by **#debug** are ignored. (Note that **-cond** and **-ncond** are **/com/cc** options; they are not available with **/bin/cc**.)

In general, you should use the conditional compilation preprocessor directives rather than **#debug**, since the former are portable and the latter is not. (See the **"#if"** listing of this encyclopedia for information on the conditional compilation directives.)

EXAMPLE

```
/* Program name is "debug_preprocessor_cmd". Use this
 * program to experiment with the -cond and -ncond
 * compiler options
 */
#include <stdio.h>

int main( void )
{
  char a_letter;
  printf( "Enter a letter -- " );
  scanf( "%c", &a_letter );
#debug printf("Echo the input -- %c\t%d\n",a_letter, a_letter);
}
```

USING THIS EXAMPLE

If we compile with the **-ncond** switch (or without the **-cond** switch), we get the following results:

```
Enter a letter -- r
```

If we compile with the **-cond** switch, we get these results instead:

```
Enter a letter -- r
Echo the input -- r          114
```

default	Refer to **switch** later in this encyclopedia.

#define and #undef (preprocessor directives) Defines and undefines program constants and macros.

FORMAT

#define *macro_name macro_body*	Define constants
#define *macro_name*(*arg* [{,*arg*}]) *macro_body*	Define macros
#undef *macro_name*	Undefine constants and macros

ARGUMENTS

macro_name	An identifier.
arg	An identifier.
macro_body	Any group of tokens. If the *macro_body* is to span more than one line, you must place a backslash \ at the end of the line (just as you would for a long string).

DESCRIPTION

A **macro** is a name that has an associated text string, called the **macro body**. By convention, macro names that represent constants consist of uppercase letters only. This makes it easy to distinguish macro names from variable names, which are generally composed of lowercase characters. In the following example, **BIG_BUFF** is the macro name and 512 is the macro body.

```
#define BIG_BUFF 512
```

When a macro name appears outside of its definition (referred to as an **invocation**), it is replaced with its macro body. The act of text replacement is referred to as **macro expansion**. For example, having defined **BIG_BUFF**, you might write:

```
char buf[BIG_BUFF];
```

During the preprocessing stage, this line of code would be translated into:

```
char buf[512];
```

The simplest and most common use of macros is to represent numeric constant values, as in the case of **BIG_BUFF**. There is another form of macros that is similar to a C function in that it takes arguments that can be used in the macro body. The syntax for this type of macro is shown in Figure 4–6.

For example, you could write:

```
#define MUL_BY_TWO(a) ((a) + (a))
```

Then you can use **MUL_BY_TWO** in your program just as you would use a function. For example, the macro invocation,

```
j = MUL_BY_TWO(5);
```

is translated by the preprocessor into:

```
j = ((5) + (5));
```

The actual argument **5** is substituted for the formal argument **a** wherever it appears in the macro body. The parentheses around **a** and around the macro body are necessary to ensure correct binding when the macro is expanded.

Note that macro arguments are not variables—they have no type, and no storage is allocated for them. Consequently, macro arguments do not conflict with variables that have the same name. The following, for example, is perfectly legal:

```
j = MUL_BY_TWO(a-1);
```

which, after expansion, becomes:

```
j = ((a-1) + (a-1));
```

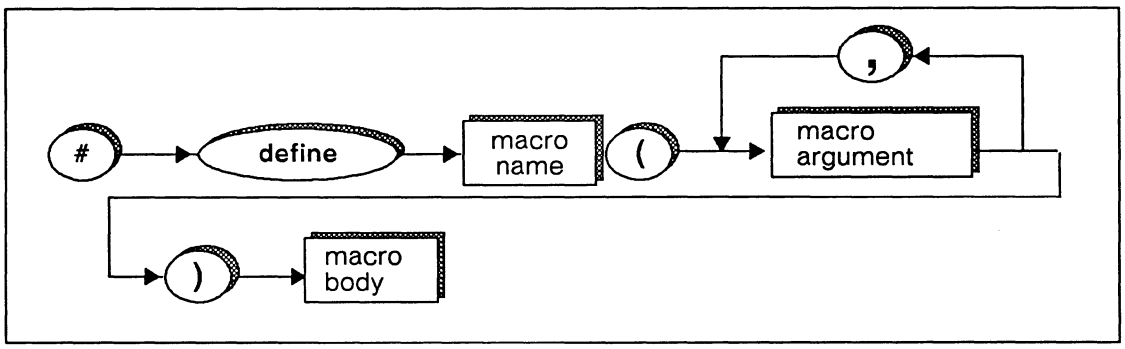


Figure 4-6. Syntax of a Function-Like Macro

Bug Alert: Ending a Macro Definition with a Semicolon

One of the most common bugs is to place a semicolon at the end of a macro definition, as in:

```
#define SIZE 10;
```

The semicolon becomes part of the replacement string, so that a statement like,

```
x = SIZE;
```

expands to:

```
x = 10;;
```

This programming error will actually go unnoticed by the compiler, which will interpret the second semicolon as a null statement. The following, however, will cause a compile-time parsing error:

```
int array[SIZE];
```

What makes this bug so difficult to find is that the line on which the error is reported looks perfectly legal. The most pernicious example of this type of bug occurs when the resulting syntax, after replacement, is legal but is semantically different from what was intended. For example:

```
#define GOOD_CONDITION (var == 1);
            .
            .
while GOOD_CONDITION
  foo();
```

This expands to:

```
while (var == 1);
  foo();
```

The semicolon after (**var == 1**) is interpreted as a null statement, and more importantly, as the body of the **while** loop. As a result, the call to **foo()** is *not* part of the **while** body. If **var** equals one, you will get an infinite loop.

Domain C supports the **-es** option (with **/com/cc**) and the **-E** option (with **/bin/cc**) that let you execute just the preprocessor. This makes it much easier to find this type of bug because you can inspect the source code after all of the macros have been expanded.

Bug Alert: Binding of Macro Arguments

A potential problem with macros is that argument expressions that are not carefully parenthesized can produce erroneous results due to operator precedence and binding. Consider the following macro:

```
#define square( a ) a * a
```

square has the advantage that it will work regardless of the argument data types. However, watch what happens when we pass it an arithmetic expression:

```
j = 2 * square( 3 + 4 );
```

expands to:

```
j = 2 * 3 + 4 * 3 + 4;
```

Because of operator precedence, the compiler interprets this expression as:

```
j = (2 * 3) + (4 * 3) + 4;
```

which assigns the value of 22 to **j**, instead of 98. To avoid this problem, you should always enclose the macro body and macro arguments in parentheses:

```
#define square( a ) ((a) * (a))
```

Now, the macro invocation expands to:

```
j = 2 * ((3 + 4) * (3 + 4));
```

which produces the correct result.

No Type Checking for Macro Arguments

From an operational point of view, the macro **MUL_BY_TWO** may seem identical to the following function:

```
int mul_by_two( a )
int a;
{
  return a+a;
}
```

However, there is one significant difference—there is no type checking for macros. In the function version of **mul_by_two**, you must pass an integral value, and the function must return an **int**. In the macro version, you can substitute any type of value for **a**.

Suppose, for example, that **f** is a **float** variable. If you write,

```
f = MUL_BY_TWO( 2.5 );
```

the preprocessor translates it into:

```
f = ((2.5) + (2.5));
```

which assigns the value 5.0 to **f**. In contrast, if you write,

```
f = mul_by_two( 2.5 );
```

the compiler takes one of two actions, depending on whether function prototypes are being used. In the presence of prototyping, the compiler converts 2.5 into an **int**, giving it a value of 2; adds two and two together, and returns 4 instead of 5.0. Without function prototypes, the compiler passes a double-precision 2.5 to the function, which interprets it as an **int**. This produces unpredictable results.

The lack of type checking for macro arguments can be a powerful feature if used with care. Consider the following macro, which returns the lesser of two arguments:

```
#define MIN( a, b ) ((a) < (b) ? (a) : (b))
```

Note that this works regardless of whether **a** and **b** are integers or floating-point values. It is extremely difficult to write an equivalent function that works for all data types.

Another difference between macros and functions is that the preprocessor checks to make sure that the number of arguments in the definition is the same as the number of arguments in the invocation. The C compiler only does this type of checking for functions if you use the ANSI prototyping syntax in the function declaration. For example, the statement,

```
MUL_BY_TWO(x, y);
```

would produce a compile-time error. The analogous statement

```
mul_by_two(x, y);
```

would produce a compile-time error only if the function is declared with the ANSI prototyping syntax. Otherwise, this statement would compile without errors, but would produce unpredictable results when executed.

Bug Alert: Using = to Define a Macro

A common mistake made in defining macros is to use the assignment operator as if you were initializing a variable. Instead of writing,

```
#define MAX 100
```

you write:

```
#define MAX = 100
```

This type of mistake can lead to obscure bugs. For example, the expression,

```
for (j=MAX; j > 0; j--)
```

would expand to:

```
for (j== 100; j > 0; j--)
```

Suddenly, the assignment is turned into a relational expression. The expression is legal, so the compiler will not complain, making the error difficult to track down.

Bug Alert: Space Between Left Parenthesis and Macro Name

Note in Figure 4–6 that the left parenthesis must come immediately after the macro name, without any intervening spaces. Insertion of a space usually results in a compile-time error, but occasionally obscure bugs can result. Consider the following macro:

```
#define neg_a_plus_f(a) -(a) + f
```

The expression,

```
j = neg_a_plus_f(x);
```

expands to:

```
j = -(x) + f;
```

But watch what happens if we accidentally insert a space between the left parenthesis and the macro name in the definition:

```
#define neg_a_plus_f (a) (-(a) + f)
```

Now, the expression expands to:

```
j = (a) -(a) + f(x);
```

If **a** is a variable name and **f** is a function name, this will look like a perfectly legal expression to the compiler.

Macros vs. Functions

Macros and functions are similar in that they both enable a set of operations to be represented by a single name. Sometimes it is difficult to decide whether to implement an operation as a macro or as a function.

In general, macros execute more quickly than functions because there is none of the function overhead involved in copying arguments and maintaining stack frames. When trying to speed up slow programs, therefore, you should be on the lookout for small, heavily used functions that can be implemented as macros. Converting functions to macros, however, will have a noticeable impact on execution speed only if the function is called frequently. Using macros can also have a significant impact on code size. The resulting executable object will probably be larger if you use macros, unless the equivalent function requires a lot of overhead.

The following lists summarize the advantages and disadvantages of macros compared to functions.

Advantages of Macros

- Macros are usually faster than functions since they avoid the function call overhead.
- The number of macro arguments is checked to match the definition. (Domain C compiler also does this for functions if you use the new ANSI prototyping syntax.)
- No type restriction is placed on arguments so that one macro may serve for several data types.

Disadvantages of Macros

- Macro arguments are re-evaluated at each mention in the macro body, which can lead to unexpected behavior if an argument contains side effects.
- Function bodies are compiled once so that multiple calls to the same function can share the same code without repeating it each time. Macros, on the other hand, are expanded each time they appear in a program. As a result, a program with many large macros may be longer than a program that uses functions in place of the macros.
- Though macros check the number of arguments, they don't check the argument types. ANSI function prototypes check both the number of arguments and the argument types.
- It is more difficult to debug programs that contain macros because the source code goes through an additional layer of translation, making the object code even further removed from the source code.

Bug Alert: Side Effects in Macro Arguments

A potential hazard of macros involves side effect operators in argument expressions. Suppose, for instance that we invoke the **MIN** macro as follows:

```
a = MIN( b++, c );
```

The preprocessor translates this into:

```
a = ((b++) < (c) ? (b++) : c);
```

If **b** is less than **c**, it gets incremented twice, obviously not what is intended. To be on the safe side, you should never use a side effect operator in a macro invocation. Side effect operators include the increment and decrement operators, the assignment operators, and function invocations.

Removing a Macro Definition

Once defined, a macro name retains its meaning until the end of the source file, or until it is explicitly removed with an **#undef** directive. The most typical use of **#undef** is to remove a definition so you can redefine it.

According to the ANSI standard and most existing C compilers, it is illegal to redefine a macro without an intervening **#undef** statement, unless the two definitions are the same. This is a useful rule because it enables you to define the same macro in different header files. If you include multiple header files (and hence, multiple definitions of the same macro), your compiler will complain only if the definitions conflict.

do/while Executes the statements within a loop until a specified condition is satisfied.

FORMAT

```
do
    statement;
while (exp);
```

ARGUMENTS

statement	A null statement, simple statement, or compound statement.
exp	Any expression.

DESCRIPTION

This is one of the three looping constructions available in C. Unlike the **for** and **while** loops, however, the **do/while** performs *statement* first and then tests *exp*. If *exp* evaluates to nonzero (true), *statement* is executed again, but when *exp* evaluates to zero (false), execution of the loop stops. This type of loop is always executed at least once.

Two ways to jump out of a **do/while** loop prematurely (that is, before *exp* becomes false) are the following:

- Use **break** to transfer control to the first statement following the **do/while** loop.
- Use **goto** to transfer control to some labeled statement outside the loop.

EXAMPLE

```
/*  Program name is "do.while_example". This program finds the
 *  summation of an integer that a user supplies, and the
 *  summation of the squares of that integer. The use of the
 *  do/while means that the code inside the loop is always
 *  executed at least once.
 */
#include <stdio.h>

int main( void )
{
  int num, sum, square_sum;
  char answer;

  printf( "\n" );
  do
  {
    printf( "Enter an integer: " );
    scanf( "%d", &num );
    sum = (num*(num+1))/2;
    square_sum = (num*(num+1)*(2*num+1))/6;
    printf("The summation of %d is: %d\n", num, sum );
    printf( "The summation of its squares is: %d\n",
             square_sum );
    printf( "\nAgain? " );
    fflush( stdin );
    scanf( "%c", &answer );
  } while ((answer != 'n') && (answer != 'N'));
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter an integer: 10
The summation of 10 is: 55
The summation of its squares is: 385

Again? y
Enter an integer: 25
The summation of 25 is: 325
The summation of its squares is: 5525

Again? n
```

#eject (preprocessor directive) Inserts a page break into the listing file. (Domain Extension)

FORMAT

#eject

DESCRIPTION

Domain C supports the **#eject** directive, which inserts a page break (formfeed) into the listing file. The statement that follows the **#eject** command is output at the top of a new page. The **#eject** directive does not affect the object file in any way.

else	Refer to **if** later in this encyclopedia.

#else	Refer to **#if** later in this encyclopedia.

#endif	Refer to **#if** later in this encyclopedia.

enum operations Operations that can be performed on enums.

DESCRIPTION

Chapter 3 explains how to define enumerated variables. Here, we explain how to use enumerated variables in the action part of your program. In conformance with the ANSI standard, Domain C allows you to use enums where integers may be used. However, we recommend that you use enums only in the following situations:

- Assign an enumerated value to an enumerated variable.
- Compare an enumerated value or variable to another enumerated value or variable.
- Use an enumerated variable as an array subscript.
- Use an enumerated variable in switch control expressions, and use enumerated values in switch case labels.
- Pass an enumerated variable to a function or return an enumerated value from a function.

For example, here is a program fragment that shows some of the possible uses of enumerated variables:

```
enum fruits {mango, apple, lemon, orange} tasty_fruits;

tasty_fruits = mango;  /* assign enum value to an enum var. */

if (tasty_fruits > apple) /* compare enum var to enum value */
   printf( "A tart fruit.\n" );

switch (tasty_fruits)  /* use enum var in a switch statement */
{
   case apple  : printf( "Grown in temperate climates.\n" );
                 break;
   case mango  :
   case lemon  :
   case orange : printf("Grown in tropical or semi-tropical
\regions.\n");
                 break;
}
```

expressions Combinations of operators and operands that evaluate to a single value.

DESCRIPTION

An **expression** consists of one or more operands and zero or more operators linked together to compute a value. For instance,

```
a + 2
```

is a legal expression that results in the sum of **a** and **2**. The variable **a** all by itself is also an expression, as is the constant **2**, since they both represent a value. There are four important types of expressions:

- **Constant Expressions** contain only constant values. For example, the following are all constant expressions:

```
5
5 + 6 * 13 / 3.0
'a'
```

- **Integral Expressions** are expressions that, after all automatic and explicit type conversions, produce a result that has one of the integer types. If **j** and **k** are integers, the following are all integral expressions:

```
j
j * k
j / k + 3
k - 'a'
3 + (int) 5.0
```

- **Float expressions** are expressions that, after all automatic and explicit type conversions, produce a result that has one of the floating-point types. If **x** is a **float** or **double**, the following are floating-point expressions:

```
x
x + 3
x / y * 5
3.0
3.0 - 2
3 + (float) 4
```

- **Pointer expressions** are expressions that evaluate to an address value. These include expressions containing pointer variables, the address-of operator (**&**), string literals, and array names. If **p** is a pointer and **j** is an **int**, the following are pointer expressions:

```
p
&j
p + 1
"abc"
(char *) 0x000fffff
```

All Expressions Have Values

One of the interesting features of C is that all expressions produce a value, called a **byproduct value**, as they are evaluated at run time. For many expressions, you won't know or care what this byproduct is. In some expressions, though, you can exploit this feature to write more compact code. Let us now look at a few examples.

Example 1

First, consider the following simple expression statement:

```
x = 6;
```

The byproduct value of all assignment expressions is the value that gets assigned, which in this case is 6. However, we do not use this byproduct value in any way.

The following example does use this byproduct value:

```
y = x = 6;
```

The equals operator binds from right to left; therefore, C first evaluates the expression **x = 6**. The byproduct of this operation is 6, so C sees the second operation as

```
y = 6
```

Example 2

Now, let us consider the following relational operator expression:

```
(10 < j < 20)
```

It is certainly tempting to use an expression like the preceding to find out whether **j** is between 10 and 20. However, it won't work. Since the relational operators bind from left to right, C first evaluates

```
10 < j
```

Note that the byproduct of a relational operation is 0 if the comparison is false and 1 if the comparison is true. Pretend that **j** equals 5. Therefore, the expression **10 < j** is false, and the byproduct is 0. Thus, the next expression that C evaluates is

```
0 < 20
```

which evaluates to true (or 1), which is the wrong answer.

Example 3

Finally, consider the following fragment:

```
static char  a_char, c[20] = {"Valerie"}, *pc = c;

   while (a_char = *pc++)
      {
            .
            .
            .
```

This **while** statement uses C's ability to both assign and test a value. Every iteration of **while** assigns a new value to variable **a_char**. The byproduct of an assignment is equal to the value that gets assigned. The byproduct value will remain nonzero until the end of the string is reached. When that happens, the byproduct value will become zero (false), and the **while** loop will end.

__FILE__	Refer to **__LINE__** listing later in this encyclopedia.

for Executes the statement(s) within a loop as long as *exp2* is true.

FORMAT

for ([*exp1*]; [*exp2*]; [*exp3*])
statement;

ARGUMENTS

exp1 — An optional element of the command. It can be any expression, although it usually is some sort of assignment statement. *exp1* is evaluated only once—at the beginning of the loop iteration.

exp2 — An optional element of the command. It can be any expression, but is usually a relational expression. If omitted, *exp2* is taken as being permanently true.

exp3 — An optional element of the command. It can be any expression, but it usually serves as the iteration instructions for the loop. It is evaluated each time after *statement* has been executed.

statement — Can be a null statement, simple statement, or compound statement.

DESCRIPTION

This is one of the three looping constructions available in C. The other two are **while** and **do/while**. The **for** statement operates as follows:

1. First, *exp1* is evaluated. This is usually an assignment expression that initializes one or more variables.
2. Then *exp2* is evaluated. This is the conditional part of the statement.
3. If *exp2* is false, program control exits the **for** statement and flows to the next statement in the program. If *exp2* is true, *statement* is executed.
4. After *statement* is executed, *exp3* is evaluated. Then the statement loops back to test *exp2* again.

Note that *exp1* is evaluated only once, whereas *exp2* and *exp3* are evaluated on each iteration. The operation of a **for** loop is shown pictorially in Figure 4–7.

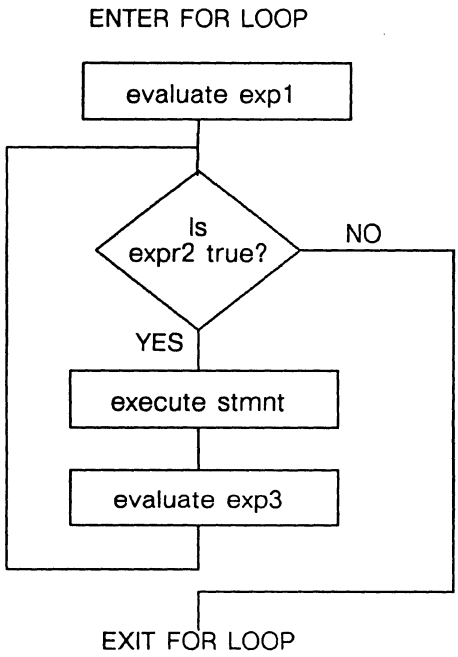


Figure 4–7. How a **for** *Loop Is Executed*

Note that **for** loops can be written as **while** loops, and vice versa. For example, the **for** loop

```
for (j = 0; j <  10; j++)
{
  do_something();
}
```

is the same as the following **while** loop:

```
j = 0;
while (j < 10)
{
  do_something();
  j++;
}
```

The **for** loop is used most commonly in situations when a variable has to be initialized and reinitialized. Most loops have this kind of construction:

for (*initialize_loop_variable* ; *finished?* ; *change_loop_variable*)
    instructions;

where *change_loop_variable* can increment or decrement the loop variable, depending on what you want. And unlike some programming languages, which restrict you to changing the loop variable by +1 or −1 only, C lets you change the loop variable by any amount. If, for example, you want to make some change to just the even-numbered members of an array, you can write:

```
for (i = 0; i < ARRAY_SIZE; i += 2)
        /*  instructions  */;
```

Any of the three expressions, or even the *statement*, can be omitted from a **for** loop, but the semicolons must appear. It is permissible, for example, to do all the work in the *exp* part of the loop and just have a semicolon appear in the *statement* section. This is convenient if you are scanning a fixed-length array to determine the length of the string stored in it. The following **for** loop scans backward from the array's maximum size, reading over any blanks, end-of-line characters, or nulls, until it finds an alphanumeric character:

```
for (i = ARRAY_SIZE-1; a[i] == ' ' || a[i] == '\n' ||
                       a[i] == '\0'; i--)
        ; /* null statement */
```

C also provides a way to combine several **for** loops into one. You can use the comma operator (,) to string together expressions. If you want to process two indexes in parallel operations, separate them with commas. For example:

```
for (i = 0, j = 10; i < j; i++, j--)
        /*  statement  */;
```

The above loop initializes **i** to zero and **j** to 10 and loops through, incrementing **i** and decrementing **j**, until **i** equals **j**.

The following describes two ways to jump out of a **for** loop prematurely (that is, before *exp2* becomes false):

- Use **break** to transfer control to the first statement following the **for** loop.
- Use **goto** to transfer control to some labeled statement outside the loop.

EXAMPLE

```
/*  Program name is "for_example". The following computes a
 *  permutation -- that is, P(n,m) = n!/(n-m)! -- using for
 *  loops to compute n! and (n-m)!)
 */
#include <stdio.h>
#define SIZE 10

int main( void )
{
  int n, m, n_total, m_total, perm, i, j, mid, count;

  printf( "Enter the numbers for the permutation (n things" );
  printf( "taken m at a time)\nseparated by a space: " );
  scanf( "%d %d", &n, &m );
  n_total = m_total = 1;
  for (i = n; i > 0; i--)          /*  compute n!  */
    n_total *= i;
```

```
  for (i = n - m; i > 0; i--)      /*  compute (n-m)!  */
    m_total *= i;
  perm = n_total/m_total;
  printf( "P(%d,%d) = %d\n\n", n, m, perm );

/* This series of for loops prints a pattern of "Z's" and shows
 * how loops can be nested and how you can either increment or
 * decrement your loop variable. The loops also show the proper
 * placement of curly braces to indicate that the outer loops
 * have multiple statements.
 */
  printf( "Now, print the pattern three times:\n\n" );
  mid = SIZE/2;

/*  controls how many times pattern is printed */
  for (count = 0; count < 3; count++)
  {
    for (j = 0; j < mid; j++)
    {

/*  loop for printing an individual line       */
      for (i = 0; i < SIZE; i++)
        if (i < mid - j || i > mid + j)
          printf( " " );
        else
          printf( "Z" );
      printf( "\n" );
    }
    for (j = mid; j >= 0; j--)
    {
      for (i = 0; i <= SIZE; i++)
        if (i < mid - j || i > mid + j)
          printf( " " );
        else
          printf( "Z" );
      printf( "\n" );
    }
  }
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter the numbers for the permutation (n things taken m at a
time) separated by a space: 4 3
P(4,3) = 24

Now, print the pattern three times:

     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
     Z
    ZZZ
   ZZZZZ
  ZZZZZZZ
 ZZZZZZZZZ
ZZZZZZZZZZZ
 ZZZZZZZZZ
  ZZZZZZZ
   ZZZZZ
    ZZZ
     Z
```

goto Unconditionally jumps to a specified label.

FORMAT

goto *label;*

ARGUMENTS

label	This is the label to which you want the **goto** to jump.

DESCRIPTION

Few programming statements have produced as much debate as the **goto** statement. The **goto** statement is necessary in more rudimentary languages, but its use in high-level languages is generally frowned upon. Nevertheless, most high-level programming languages, including C, contain a **goto** statement for those rare situations where it can't be avoided.

The purpose of the **goto** statement is to enable program control to jump to some other spot. The destination spot is identified by a **statement label**, which is just a name followed by a colon. The label must be in the same function as the **goto** statement that references it.

With deeply nested logic there are times when it is cleaner and simpler to bail out with one **goto** rather than backing out of the nested statements. The most common and accepted use for a **goto** is to handle an extraordinary error condition. The following sample program shows a **goto** that easily could be avoided through the use of a **while** loop, and also shows what an illegal **goto** looks like.

EXAMPLE

```
/*  Program name is "goto_example". This program finds the
 *  circumference and area of a circle when the user gives
 *  the circle's radius.
 */
#include <stdio.h>
#define PI 3.14159

int main( void )
{
  float cir, radius, area;
  char answer;
  extern void something_different( void );

circles:
  printf( "Enter the circle's radius: " );
  scanf( "%f", &radius );
  cir = 2 * PI * radius;
  area = PI * (radius * radius);
  printf( "The circle's circumference is: %6.3f\n", cir );
  printf( "Its area is: %6.3f\n", area );
  printf( "\nAgain? y or n: " );
  fflush( stdin );
  scanf( "%c", &answer );
  if (answer == 'y' || answer == 'Y')
    goto circles;
  else
  {
    printf( "Do you want to try something different? " );
    fflush( stdin );
    scanf( "%c", &answer );
    if (answer == 'y' || answer == 'Y')
/*    goto different;          WRONG! This label is in  */
/*                             another block.           */
      something_different();
  } /*  end else  */
}

void something_different( void )
{
different:
  printf( "Hello. This is something different.\n" );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter the circle's radius: 3.5
The circle's circumference is: 21.991
Its area is: 38.484

Again? y or n: y
Enter the circle's radius: 6.1
The circle's circumference is: 38.327
Its area is: 116.899

Again? y or n: n
Do you want to try something different? y
Hello. This is something different.
```

if Tests one or more conditions and executes one or more statements according to the outcome of the tests.

FORMAT

```
if (exp)        /* format 1 */
   statement


if(exp)         /* format 2 */
   statement1
else
   statement2
```

ARGUMENTS

exp	Any expression.
statement	Any null, simple, or compound statement. Note that a statement can itself be another **if** statement. Remember, a statement ends with a semicolon.

DESCRIPTION

The **if** and **switch** statements are the two conditional branching statements in C. The **if** statement can take either of the two forms shown in the Format section.

In the first form, if *exp* evaluates to true (any nonzero value), C executes *statement*, while if *exp* is false (evaluates to zero), C simply falls through to the next line in the program.

In the second form, if *exp* evaluates to true, C executes *statement1*, but if *exp* is false, *statement2* is performed.

Note that a statement can itself be an **if** or **if/else** statement. Therefore, you can test multiple conditions with a command that looks like this:

```
if (exp1)     /*  multiple conditions */
   statement1
else if (exp2)
   statement2
else if (exp3)
   statement3
.
.
.
else
   statementN
```

The important thing to remember is that C executes at most only one statement in the **if...else** and **if...else/if...else** constructions. Several expressions may indeed be true, but only the statement associated with the first true expression is executed. The system does not even look at subsequent expressions. For example:

```
/* determine reason the South lost the Civil War */
if (less_money)
  printf( "It had less money than the North.\n" );
else if (fewer_supplies)
  printf( "It had fewer supplies than the North.\n" );
else if (fewer_soldiers)
  printf( "It had fewer soldiers.\n" );
else
{
  printf( "Its agrarian society couldn't compete with the" );
  printf( "North's industrial one.\n" );
}
```

All the expressions in the above code fragment could be evaluated to true, but the run-time system would only get as far as the first line and never even test the remaining expressions.

If you use a compound statement in one of the **if** constructions, remember to use the curly braces to indicate where the statement begins and ends. For example:

```
if (x > y)
{
  temp = x;
  x = y;
  y = temp;
}
else
  /* make next comparison */
```

Braces also are important when you nest **if** statements. Since the **else** portion of the statement is optional, you may not have one for an inner **if**. However, C associates an **else** with the closest previous **if** unless you use braces to show that isn't what you want. For example:

```
if (month == 12)
{
  if (day == 25)     /*  month = November  */
    printf( "Today is Christmas.\n" );
}
else
  printf( "It's not even December.\n" );
```

Without the braces, the **else** would be associated with the inner **if** statement, and so the no-December message would be printed for any day in December except December 24. Nothing would be printed if **month** did not equal 12.

Bug Alert: The Dangling else

Nested **if** statements create the problem of matching each **else** phrase to the right **if** statement. This is often called the **dangling else** problem. The general rule is:

An **else** *is always associated with the nearest previous* **if**.

Each **if** statement, however, can have only one **else** phrase. It is important to format nested **if**s correctly to avoid confusion. An **else** phrase should always be at the same indentation level as its associated **if**. However, don't be misled by indentations that look right even though the syntax is incorrect.

EXAMPLE

```
/*  Program name is "if.else_example". */
#include <stdio.h>

int main( void )
{
  int age, of_age;
  char answer;

/*  This if statement is an example of the second form (see
 *  "Description" section).
 */
  printf( "\nEnter an age: " );
  scanf( "%d", &age );
  if (age > 17)
    printf( "You're an adult.\n" );
  else
  {
    of_age = 18 - age;
    printf( "You have %d years before you're an adult.\n",
             of_age);
  } /*  end else  */
  printf( "\n" );
  printf( "This part will help you decide whether to jog \
today.\n" );
  printf( "What is the weather like?\n" );
  printf( "      raining = r\n" );
  printf( "      cold = c\n" );
  printf( "      muggy = m\n" );
  printf( "      hot = h\n" );
  printf( "      nice = n\n" );
  printf( "Enter one of the choices: " );
  fflush( stdin );
  scanf( "\n%c", &answer );

/*  This illustrates the common "else if" idiom */
  if (answer == 'r')
    printf( "It's too wet to jog today. Don't bother.\n" );
  else if (answer == 'c')
    printf("You'll freeze if you jog today. Stay indoors.\n" );
  else if (answer == 'm')
    printf("It's no fun to run in high humidity. Skip it.\n" );
  else if (answer == 'h')
    printf( "You'll sweat to death if you try to jog today. So\
don't.\n" );
  else if (answer == 'n')
    printf("You don't have any excuses. You'd better go run.\n");
  else
    printf( "You didn't give a valid answer.\n" );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter an age: 15
You have 3 years before you're an adult.

This part will help you decide whether to jog today.
What is the weather like?
      raining = r
      cold = c
      muggy = m
      hot = h
      nice = n
Enter one of the choices: r
It's too wet to jog today. Don't bother.
```

#if, #ifdef, #ifndef, #elif, #else, #endif (preprocessor directives) and defined (predefined macros)
Control conditional compilation.

FORMAT

#if *const_exp*	
#else	
#elif	(Supported only by the UNIX preprocessor)
#endif	
#ifdef *identifier*	
#ifndef *identifier*	
defined(*identifier*)	Predefined macro
defined *identifer*	Predefined macro

ARGUMENTS

const_exp	Any constant expression.
identifier	Any identifier.

DESCRIPTION

These preprocessor directives and predefined macros work together, so we explain them together in this one listing.

The #if, #else, and #endif Preprocessor Directives

Use these preprocessor directives to conditionally compile sections of your source code. For example, suppose you are writing a program that is to run on either a color or monochromatic node. Further suppose that although most of the program is independent of the target, a fraction of the program does depend on the target. In other words, the code for the color target is different from the code for the monochromatic target. To solve this problem you could just write two different programs. However, this makes program debugging and maintenance much more expensive since a change in one program would have to be duplicated in the other. A better solution is to use the conditional compilation preprocessor directives as follows:

```
.
. /* code applying to both color and monochromatic nodes */
.
#if color
   .
   .  /* code for color nodes only */
   .
#else
   .
   .  /* code for monochromatic nodes only */
   .
#endif
.
. /* code applying to both color and monochromatic nodes */
.
```

The **#if** directive takes a constant expression as its sole argument. If this constant expression evaluates to nonzero, then all the code up until an **#else** or **#endif** is compiled. If the constant expression evalutes to zero, then no code is compiled until the next **#else** or **#endif** directive.

There are a number of differences between the preprocessor conditional statements and the C language conditional statements:

- The conditional expression in an **#if** directive need not be enclosed in parentheses. (Parentheses may optionally be included.)
- Blocks of statements under the control of a conditional preprocessor directive are not enclosed in braces. Instead, they are bounded by an **#else**, or **#endif** statement.
- Every **#if** block may contain only one **#else** block.
- Every **#if** block must end with an **#endif** directive.
- Any macros in the conditional expression are expanded before the expression is evaluated.
- If a conditional expression contains a name that has not been defined, it is replaced by the constant zero. For example, the sequence,

  ```
  #undef x
  #if x
  ```

  expands to:

  ```
  #if 0
  ```

Conditional compilation is particularly useful during the debugging stage of program development since you can turn sections of code on or off by changing the value of a macro. Consider the following snippet:

```
#if DEBUG
      if (exp_debug)
      {
        printf( "lhs = " );
        print_value( result );
        printf( " rhs = " );
        print_value( &rvalue );
        printf( "\n" );
      }
#endif
```

If the macro **DEBUG** is a nonzero value, the **if** statement and **printf()** calls will be compiled. If **DEBUG** is zero, these statements will be ignored as if they were a comment. If **DEBUG** is not defined, it is the same as if it were defined to expand to zero.

Domain C has a command line option that lets you define macros before compilation begins. If you compile under the UNIX system, use the **-D** option to define macros. Under Aegis, use the **-def** option. To receive debug information, you would define the macro **DEBUG** to be some nonzero value:

```
cc -DDEBUG=1 test     (under the UNIX system)
cc -def DEBUG=1 test (under the Aegis system)
```

Note that the **#if** and **#endif** directives control whether the enclosed C statements are compiled, not necessarily whether they are executed. In the above example, the **printf()** calls are only executed if the **exp_debug** variable has a nonzero value. This double-layer approach enables you to include the diagnostic statements in the executable program, but still decide each time you run the program whether you want them executed. If, for the final version, you need to reduce the size of the executable program, you can compile it with **DEBUG** set to zero.

Another common use of the conditional compilation mechanism is to choose between the old function declaration syntax and the new ANSI prototyping syntax:

```
#if (__STDC__ == 1)
  extern int foo( char a, float b );
  extern *char goo( char *string );
#else
  extern int foo();
  extern *char goo();
#endif
```

By default, the compiler sets **__STDC__** to 1 and uses the prototyping syntax to declare the types of each argument. If you compile with **-ntype**, the compiler uses the old function declaration syntax.

The #elif Directive

The **#elif** directive is supported by the UNIX preprocessor (**cpp**), but not by the Aegis preprocessor. Therefore, use **#elif** only if you are compiling in a UNIX environment or if you explicitly use the **/bin/cc** command. The **#elif** directive is a shorthand for the combination of an **#else** directive followed by an **#if** directive. For example, the following sequence is written without **#elif**s.

```
#if (TEST == 0)
  printf( "No test\n" );
#else
#if (test == 1)
  printf( "Test #1\n" );
#else
#if (test == 2)
  printf( "Test #3\n" );
#endif
```

Using **#elif**s, you could rewrite this:

```
#if (TEST == 0)
  printf( "No test\n" );
#elif (test == 1)
  printf( "Test #1\n" );
#elif (test == 2)
  printf( "Test #3\n" );
#endif
```

The #ifdef and #ifndef Preprocessor Directives

Use the **#ifdef** command to determine if an identifier is currently defined. In this context "defined" means that the identifier was used in a **#define** preprocessor directive or used in the **-D** (**/bin/cc**) or **-def** (**/com/cc**) compiler option. **#ifndef** checks whether an identifier is *not* currently defined.

For example:

```
#ifdef TEST
  printf( "This is a test.\n" );
#else
  printf( "This is not a test.\n" );
#endif
```

If the macro **TEST** is defined, the first **printf()** call will be compiled. If **TEST** is not a defined macro, the second **printf()** call is compiled. Note that it doesn't matter what **TEST** expands to, only whether it exists or not. As with **#if**, an **#ifdef** and **#ifndef** block must be terminated by an **#endif** statement.

Another way to write the previous example is to use the preprocessor **defined** operator (an ANSI feature):

```
#if defined TEST
```

or

```
#if defined( TEST )
```

The parentheses around the macro name are optional. By definition,

```
#if defined macro_name
```

is equivalent to:

```
#ifdef macro_name
```

and the directive,

```
#if !defined macro_name
```

is equivalent to:

```
#ifndef macro_name
```

The **defined** macro is particularly useful for performing logical operations. For example:

```
#if defined(Domain) && !defined(Aegis) && DEBUG
```

In most instances, you can use **#if** instead of **#ifdef** and **#ifndef**, since the macro name expands to zero if it is not defined. The one exception where you need to use **#ifdef** or **#ifndef** is when the macro is defined to zero. For example, you may want to define the macro **FALSE** to expand to zero. If you use an **#if** directive to test whether **FALSE** is defined, **FALSE** will be redefined even if it is already defined to expand to zero. More important, it won't be redefined if it is defined to something other than zero.

```
#if !FALSE
#  define FALSE 0
#endif
```

You can avoid both of these problems by using **#ifndef.**

```
#ifndef FALSE
#  define FALSE 0
#elif FALSE
#  undef FALSE
#  define FALSE 0
#endif
```

#ifdef Refer to the **#if** listing earlier in this chapter.

#ifndef Refer to the **#if** listing earlier in this chapter.

#include (preprocessor directive) Inserts an include file into the source code.

FORMAT

#include <*pathname*>
#include "*pathname*"

ARGUMENTS

pathname The pathname of the file that is to be included into the source code.

DESCRIPTION

The **#include** preprocessor directive inserts the contents of the specified file into the source file prior to compilation. For example, if you put the following **#include** into your source code

```
f(x);
#include "//lucas/eleven/rings.ins.c"
g(x);
```

then the C precompiler inserts the entire contents of the file into your source code between **f(x)** and **g(x)**. After this insertion, the C compiler compiles the inserted lines just as it would compile any other lines of source code.

The **#include** command enables you to create common definition files, called **header files**, to be shared by several source files. Header files traditionally have a **.h** suffix and contain data structure definitions, macro definitions, and any global data necessary for modules to communicate with each other.

The Domain preprocessors support up to 12 levels of nested header files.

The Domain/OS operating system supplies many header files (sometimes called "include files") that describe structures internal to the operating system. The C run-time library also includes a number of header files that must be included in order to invoke associated functions. See the *SysV Programmer's Reference* manual and the *BSD Programmer's Reference* manual for more information about run-time library header files.

By default, the C compiler automatically tries to include the following file at the beginning of each source file you compile:

```
/usr/include/apollo_$std.h
```

This file sets up predefined, system-wide definitions. Because it is automatically included, you do not need to explicitly include this file in your code.

If the compiler cannot locate **/usr/include/apollo_$std.h**, no action is taken and no error is reported. If the compiler does locate the file, it processes the file like any other include file.

In Domain/OS pathname strings, the backslash character (\) represents the parent directory. Consequently, when the compiler detects a backslash character in an include file string, it does not interpret it as a normal escape character. This special interpretation of backslash applies *only* to include files.

How the C Preprocessor Searches for Include Files

The **#include** command has two forms:

#include <*filename*>

or

#include "*filename*"

If the filename is surrounded by angle brackets, the preprocessor looks in a list of implementation-defined places for the file. On Domain/OS systems, the compiler looks in the directory **/usr/include** unless alternative directories are specified with the **-idir** option (**/com/cc**) or the **-I** option (**/bin/cc**). (See the description of **-idir** and **-I** in Chapter 6 for more information about specifying search directories.)

If the filename is surrounded by double quotes, the preprocessor looks for the file according to the file specification rules of the operating system (described below). If the preprocessor can't find the file there, it searches for the file as if it had been enclosed in angle brackets.

For header files enclosed in double quotes, the Domain/OS operating system distinguishes between two kinds of pathnames: **relative pathnames** and **absolute pathnames**. An absolute pathname begins with a slash (/), double slash (//), backslash (\), tilde (~), or period (.); for example, the following include files are all absolute pathnames:

```
#include "//rastelli/six/plates.ins.c"
#include "/ignatov/seven/clubs"
#include "~/brunn/spinning.ins.c"
#include "./noakes/passing/tricks.ins.c"
```

When *pathname* is an absolute file, the C preprocessor searches this pathname only. If the preprocessor does not find this pathname, it issues an error.

Relative pathnames begin with an identifier; for example, here are two relative pathnames:

```
#include "jensby/jensen.ins.h"
#include "my_include_file.h"
```

The search method for relative pathnames depends on whether you use the UNIX **/bin/cc** interface or the Aegis /com/cc interface. This difference is due to the fact that **/bin/cc** invokes the UNIX preprocessor (**cpp**) whereas /com/cc uses the Aegis preprocessor. With **/bin/cc**, relative pathnames are relative to the directory of the including source file. With **/com/cc**, relative pathnames are always relative to the working directory. These differences are described in more detail below.

Compiling with /com/cc

For relative pathnames delimited by double quotation marks ("*pathname*"), the compiler first searches for *pathname* in the working directory. If it is not there, the compiler searches any directories you specified with **-idir**. If it is not in any of them, the compiler searches directory **/usr/include**. If it is not there, the compiler issues an error.

Compiling with /bin/cc

For relative pathnames delimited by double quotation marks ("*pathname*"), the compiler searches for *pathname* in the following order:

1. The preprocessor searches in the directory of the including source file.
2. If it is not there, the preprocessor searches in the working directory.
3. If it is not there, the preprocessor searches in any directories you specified with **-I**.
4. If it is not in any of them, the preprocessor searches in directory **/usr/include**.
5. If it is not there, the preprocessor issues an error.

increment and decrement operators	Operators that you can use to increment or decrement variables.

FORMAT

lvalue++	Increment, postfix form
++*lvalue*	Increment, prefix form
lvalue--	Decrement, postfix form
--*lvalue*	Decrement, prefix form

ARGUMENTS

lvalue — Any previously declared integer or pointer lvalue. (See Section 4.2 for a definition of lvalue.) Note that although *lvalue* can be a pointer variable, it cannot be a pointer to a function.

DESCRIPTION

C's increment (++) and decrement (--) operators are good examples of the language's tendency toward compactness. The increment operator adds 1 to its operand, and the decrement operator subtracts 1 from its operand. So while in many languages statements must look something like these

```
i = i + 1;
j = j - 1;
```

to increment the variable **i** and decrement **j**, in C you can just type

```
i++;
j--;
```

The increment and decrement operators are unary. The operand must be a scalar lvalue—it is illegal to increment or decrement a constant, structure, or union. It is legal to increment or decrement pointer variables, but the meaning of adding 1 to a pointer is different from adding 1 to an arithmetic value. This is described in the "pointer arithmetic" section of this chapter.

There are two forms for each of the operators: **postfix** and **prefix**. Both forms increment or decrement the appropriate variable, but they do so at different times. The statement **++i** (prefix form) increments **i** *before* using its value, while **i++** (postfix form) increments it *after* its value has been used. This difference can be important to your program.

The postfix increment and decrement operators fetch the current value of the variable and store a copy of it in a temporary location. The compiler then increments or decrements the variable. The temporary copy, which has the variable's value *before* it was modified, is used in the expression. For example:

```
/* Program name is "inc.dec_example1" */

#include <stdio.h>

int main( void )
{
  int j = 5, k = 5;
  printf( "j: %d\t k: %d\n", j++, k-- );
  printf( "J: %d\t k: %d\n", j, k );
}
```

The result is:

```
j: 5      k: 5
j: 6      k: 4
```

In the first **printf()** call, the initial values of **j** and **k** are used, but once they have been used they are incremented and decremented, respectively.

In contrast, the *prefix* increment and decrement operators modify their operands *before* they fetch the values:

```
/* Program name is "inc.dec_example2" */

#include <stdio.h>

int main( void )
{
  int j = 5, k = 5;
  printf( "j: %d\t k: %d\n", ++j, --k );
  printf( "J: %d\t k: %d\n", j, k );
}
```

The result of this version is:

```
j: 6      k: 4
j: 6      k: 4
```

In many cases, you are interested only in the side effect, not in the result of the expression. In these instances, it doesn't matter which operator you use. For example, as a stand-alone assignment, or as the third expression in a for loop, the side effect is the same whether you use the prefix or postfix versions:

```
x++;
```

is equivalent to:

```
++x;
```

and the statement

```
for (j = 0; j <= 10; j++)
```

is equivalent to:

```
for (j = 0; j <= 10; ++j)
```

You need to be careful, however, when you use the increment and decrement operators within an expression. Consider the following function that inserts newlines into a text string at regular intervals.

```
#include <stdio.h>

void break_line( int interval )
{
  int c, j=0;

  while ((c = getchar()) != '\n')
  {
    if ((j++ % interval) == 0)
      printf( "\n" );
    putchar( c );
  }
}
```

This works because we use the postfix increment operator. If we were to use the prefix increment operator, the function would break the first line one character early.

Precedence of Increment and Decrement Operators

Note in Table 4-1 that the increment and decrement operators have the same precedence, but bind from right to left. So the expression,

```
--j++
```

is evaluated as:

```
--(j++)
```

This expression is illegal because j++ is not an lvalue as required by the -- operator. In general, you should avoid using multiple increment or decrement operators together.

Bug Alert: Side Effects

The increment and decrement operators and the assignment operators cause **side effects**. That is, they not only result in a value, but they change the value of a variable as well. A problem with side effect operators is that it is not always possible to predict the order in which the side effects occur. Consider the following statement:

```
x = j * j++;
```

The C language does not specify which multiplication operand is to be evaluated first. One compiler may evaluate the left-hand operand first, while another evaluates the right-hand operand first. The results are different in the two cases. If **j** equals 5, and the left-hand operand is evaluated first, the expression will be interpreted as:

```
x = 5 * 5;    /* x is assigned 25 */
```

If the right-hand operand is evaluated first, the expression becomes:

```
x = 6 * 5;    /* x is assigned 30 */
```

Statements such as this one are not portable and should be avoided. The side effect problem also crops up in function calls because the C language does not guarantee the order in which arguments are evaluated. For example, the function call,

```
f( a, a++ )
```

is not portable because compilers are free to evaluate the arguments in any order they choose.

To prevent side effect bugs, follow this rule: *If you use a side effect operator in an expression, do not use the affected variable anywhere else in the expression.* The ambiguous expression above, for instance, can be made unambiguous by breaking it into two assignments:

```
x = j * j;
++j;
```

EXAMPLE

```
/*  Program name is "inc.dec_example3".  */

#include <stdio.h>

int main( void )
{
  int n, m, n_total, m_total, perm, i, num;

/*  The following computes a permutation -- that is,
 *  P(n,m) = n!/(n-m)! -- using decrement operators
 *  to compute n! and (n-m)!)
 */
  printf( "Enter the numbers for the permutation (n" );
  printf( " things taken m at a time)\nseparated by a" );
  printf( " space: " );
  scanf( "%d %d", &n, &m) ;
  n_total = m_total = 1;

  for (i = n; i > 0; i--)          /*  compute n!  */
    n_total *= i;
  for (i = n - m; i > 0; i--)      /*  compute (n-m)!  */
    m_total *= i;
  perm = n_total/m_total;
  printf( "P(%d,%d) = %d\n\n", n, m, perm );

/*  This part shows the increment operator  */
  printf ("\nAnd now, the squares of 1 to 5:\n");
  for (n = 1; n <= 5; n++)
  {
    num = n*n;
    printf( "%d\n", num );
  }
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter the numbers for the permutation (n things taken m at a
time) separated by a space: 4 3
P(4,3) = 24


And now, the squares of 1 to 5:
1
4
9
16
25
```

__LINE__ and __FILE__ (predefined symbols) Predefined symbols that expand to the current line number and source filename.

FORMAT

__LINE__
__FILE__

Note that there are two underscores before and two underscores after each of these preprocessor symbols

DESCRIPTION

The preprocessor recognizes these special predefined symbols and replaces their occurrences with the following:

__LINE__ Expands to the source file line number on which it is invoked.

__FILE__ Expands to the name of the file in which it is invoked.

The **__LINE__** and **__FILE__** macros are valuable diagnostic tools. Suppose, for example, that you want a check facility that compares two expressions for equality and, if they are unequal, calls an error reporting function with the source filename and the line number of the check failure.

```
#include <stdio.h>
#define CHECK( a, b ) \
   if ((a) != (b)) \
    fail( a, b, __FILE__, __LINE__)

void fail( int a, int b, char *p, int line )
{
  printf( "Check failed in file %s at line %d:\
received %d, expected %d\n", p, line, a, b );
}
```

At various points in a program, you can check to make sure that a variable **x** equals zero by including the following diagnostic:

```
CHECK(x, 0);
```

Note that blank lines are included in the line count. Comment lines also are included. The symbol substitutions are performed before any other preprocessor commands. Consequently, **__FILE__** and **__LINE__** get defined before any **#include** statements that might change their values. Note that **__LINE__** and **__FILE__** are affected by the **#line** directive.

#line (preprocessor directive) Lets you set the current source line number.

FORMAT

#line *integer* [*"filename"*] /* first form */

#integer [*"filename"*] /* second form */

ARGUMENTS

integer An integer constant.

"filename" A filename enclosed in double quotes.

DESCRIPTION

The **#line** preprocessor directive allows you to set the compiler's knowledge of the current source line number and (optionally) current source file. The compiler reports errors in terms of the line numbers set by this option. In addition, the debugger line number table is built with these line numbers. The debugger source file option is given the last *"filename"* in the source, as long as that file truly exists. (The compiler verifies the existence of the source file before it creates the debug entry.)

The word **line** may be omitted, as shown in the second form, but this feature is not portable. The optional *filename* must be enclosed in double quotes. The filename may be any legal pathname.

EXAMPLE

The following example illustrates the behavior of #line.

```
/* Program name is "line_example". Example of #line
 * preprocessor directive.
 */
#include <stdio.h>

int main( void )
{
  printf( "Current line %d\nFilename: %s\n\n", __LINE__, __FILE__ );
#line 100
  printf( "Current line %d\nFilename: %s\n\n", __LINE__, __FILE__ );
#line 200 "new_name"
  printf( "Current line %d\nFilename: %s\n\n", __LINE__, __FILE__ );
}
```

USING THIS EXAMPLE

Assuming that the source file for this program is called **line_example.c**, execution produces:

```
Current line: 7
Filename: line_example.c

Current line: 101
Filename: line_example.c

Current line: 201
Filename: new_name
```

The **#line** feature is particularly useful for programs that produce C source text. For instance, **yacc** (which stands for Yet Another Compiler Compiler) is a UNIX utility that facilitates building compilers. The **yacc** utility reads files written in the **yacc** language and produces a file written in the C language, which can then be compiled by a C compiler. A problem arises, however, if the C compiler encounters an error in the **yacc**–produced C file. You want to know which line in the original **yacc** file is causing the error, but the C compiler will report the error–producing line in the C text file. To solve this problem, **yacc** writes **#line** directives in the C source file so that the compiler is fooled into reporting errors based on the **yacc** line numbers rather than the C line numbers.

#list and #nolist (preprocessor directives) Enables and disables the listing of source code in the listing file. (Domain Extension)

FORMAT

#list
#nolist

DESCRIPTION

The **#list** preprocessor directive enables the listing of source code in the listing file, while **#nolist** inhibits the listing of source code. For example, this sequence of preprocessor directives

```
.
.
.
#nolist
#include "/my_insert_files/beth.ins.c"
#list
.
.
.
```

excludes the contents of **/my_insert_files/beth.ins.c** from the source listing. Note that **#list** and **#nolist** have no effect on the compilation; they only affect the source listing file.

The default is **#list**.

logical operators Logical AND, OR, and NOT operators.

FORMAT

exp1 && *exp2*	Logical AND
exp1 || *exp2*	Logical OR
!*exp1*	Logical NOT

ARGUMENTS

exp1 Any expression.

exp2 Any expression.

DESCRIPTION

The logical AND operator (&&) and the logical OR operator (||) evaluate the truth or falseness of pairs of expressions. The AND operator evaluates to 1 if and only if both expressions are true. The OR operator evaluates to 1 if *either* expression is true. To test whether y is greater than x and less than z, you would write:

```
(x < y) && (y < z)
```

The logical negation operator (!) takes only one operand. If the operand is true, the result is false; if the operand is false, the result is true.

Recall that in C, true is equivalent to any nonzero value, and false is equivalent to zero. Table 4–9 shows the logical tables for each operator, along with the numerical equivalent. Note that all of the operators return 1 for true and 0 for false.

Table 4–9. Truth Table for C's Logical Operators

Operand	Operator	Operand	Result
zero	&&	zero	0
nonzero	&&	zero	0
zero	&&	nonzero	0
nonzero	&&	nonzero	1
zero	||	zero	0
nonzero	||	zero	1
zero	||	nonzero	1
nonzero	||	nonzero	1
not applicable	!	zero	1
	!	nonzero	0

The operands to the logical operators may be integers or floating-point objects. The expression

```
1 && -5
```

results in 1 because both operands are nonzero. The same is true of the expression

```
0.5 && -5
```

Logical operators (and the comma and conditional operators) are the only operators for which the order of evaluation of the operands is defined. The compiler must evaluate operands from left to right. Moreover, the compiler is guaranteed *not* to evaluate an operand if it's unnecessary. For example, in the expression

```
if ((a != 0) && (b/a == 6.0))
```

if **a** equals zero, the expression **(b/a == 6)** will *not* be evaluated. This rule can have unexpected consequences when one of the expressions contains side effects (See the Bug Alert in this section.)

Table 4-10 shows a number of examples that use relational and logical operators. Note that the logical NOT operator has a higher precedence than the others. The AND operator has higher precedence than the OR operator. Both the logical AND and OR operators have lower precedence than the relational and arithmetic operators.

Table 4-10. Examples of Expressions Using the Logical Operators

Given the following declarations:

```
int j = 0, m = 1, n = -1;
float x = 2.5, y = 0.0;
```

Expression	Equivalent Expression	Result
j && m	(j) && (m)	0
j < m && n < m	(j < m) && (n < m)	1
m + n || ! j	(m + n) || (!j)	1
x * 5 && 5 || m / n	((x * 5) && 5) || (m / n)	1
j <= 10 && x >= 1 && m	((j <= 10) && (x >= 1)) && m	1
!x || !n || m + n	((!x) || (!n)) || (m + n)	0
x * y < j + m || n	((x * y) < (j + m)) || n	1
(x > y) + !j || n++	((x > y) + (!j)) || (n++)	1
(j || m) + (x || ++n)	(j || m) + (x || (++n))	2

Bug Alert: Side Effects in Relational Expressions

Relational operators (and the conditional and comma operators) are the only operators for which the order of evaluation of the operands is defined. For these operators, operands must be evaluated from left to right. However, the system evaluates only as much of a relational expression as it needs to determine the result. In many cases, this means that the system does not need to evaluate the entire expression. For instance, consider the following expression:

```
if ((a < b) && (c == d))
```

The system begins by evaluating **(a < b)**. If **a** is not less than **b**, the system knows that the entire expression is false, so it will not evaluate **(c == d)**. This can cause problems if some of the expressions contain side effects:

```
if ((a < b) && (c == d++))
```

In this case, **d** is only incremented when **a** is less than **b**. This may or may not be what the programmer intended. In general, you should avoid using side effect operators in relational expressions.

EXAMPLE

```
/*  Program name is "logical_ops_example". This program
 *  shows how logical operators are used. Notice that
 *  several logical expressions can be strung together
 *  to create multiple conditions. Also notice how the
 *  NOT operator (!) is used. In the program itself,
 *  the integer variables are initialized to zero,
 *  which C evaluates as being false. Then, if a
 *  question is answered "yes", the appropriate
 *  variable is reset to 1. C considers a nonzero
 *  value to be true.
 */
#include <stdio.h>

int main( void )
{
  int won_lottery, enough_vacation, money_saved;
  char answer;

  won_lottery = enough_vacation = money_saved = 0;

  printf("\nThis program determines whether you can " );
  printf( "take your next vacation in Europe.\n" );
  printf( "Have you won the lottery? y or n: " );
  fflush( stdin );
  scanf( "%c", &answer );
  if (answer == 'y')
```

```
    won_lottery = 1;

  printf( "Do you have enough vacation days saved? \
y or n: " );
  fflush( stdin );
  scanf ("%c", &answer);
  if (answer == 'y')
    enough_vacation = 1;

  printf( "Have you saved enough money for the trip? \
y or n: " );
  fflush( stdin );
  scanf( "%c", &answer );
  if (answer == 'y')
    money_saved = 1;

  printf( "\n" );
  if (won_lottery)
  {
    printf("Why do you need a program to decide if you");
    printf( " can afford a trip to Europe?\n" );
  }  /*  end if  */
  if (won_lottery || (enough_vacation && money_saved))
    printf( "Look out Paris!\n" );
  else if (enough_vacation && (!money_saved))
    printf( "You've got the time, but you haven't got \
the dollars.\n" );
  else if (!enough_vacation || (!money_saved))
  {
    printf( "Tough luck. Try saving your money and " );
    printf( "vacation days next year.\n" );
  }  /*  end else/if  */
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
This program determines whether you can take your next vacation
in Europe.
Have you won the lottery? y or n: y
Do you have enough vacation days saved? y or n: n
Have you saved enough money for the trip? y or n: n

Why do you need a program to decide if you can afford a trip to
Europe?
Look out Paris!
```

#module (preprocessor directive)	Changes the name of the object module for debugging purposes, and, optionally, lets you define procedure and data section names. (Domain Extension)

FORMAT

#module *module_name* [, *psect_name* [, *dsect_name*]]

ARGUMENTS

module_name An identifier that serves as the new name of the module.

psect_name An optional identifier. This is the name of the procedure section that the code will go into.

dsect_name An optional identifier. This is the name of the data section that the data will go into.

DESCRIPTION

The **#module** directive serves two purposes. First, it enables you to change the name of the object module for debugging purposes. Second, it allows you to define a procedure and data section for the code in the file. By defining sections, you can have some control over how the linker groups data and instructions in memory. This is described in more detail in the description of **#section** later in this chapter.

There may be at most one **#module** statement per source file and it must appear before any other tokens, with the exception of the #systype directive. The *module_name* is required, but both *psect_name* and *dsect_name* are optional. If you include a *psect_name* or *dsect_name*, the specified section names are active until the end of the file or until a **#section** statement redefines one or both of the names. The following examples illustrate the legal syntaxes of **#module**.

```
#module example          /* defines "example" as the module name */

#module example, proc_a   /* defines "proc_a" as the procedure
                           * section
                           */

#module example, proc_a, data_a  /* defines "proc_a" as the
                                  *  procedure section and "data_a"
                                  *  as the data section
                                  */
#module example,,data_a   /*  defines "data_a" as the data section,
                           *  but uses the default name for the
                           *  procedure section
                           */
```

The **dde** and **dbx** utilities—the Domain/OS language-level debuggers—use the module name as the starting point when they search for functions and static variable names. If you do not use **#module**, the compiler uses the source filename in uppercase, with underscores replacing any periods. For example, the default module name for **test.1.c** is **test_1_c**.

See the *Domain Distributed Debugging Environment (Domain/DDE) Reference* manual for more information on accessing identifiers while debugging.

#nolist Refer to the **#list** entry earlier in this chapter. (Domain Extension)

pointer operations Operations performed with pointers.

DESCRIPTION

A pointer variable is a variable that can hold the address of an object. Chapter 3 describes how to declare pointer variables. Here, we describe how to use pointer variables in the code portion of your program. We discuss pointers to functions in Chapter 5.

We start with a discussion of the two principal pointer operations—finding an address and dereferencing a pointer.

Assigning an Address Value to a Pointer

To declare a **pointer** variable, you precede the variable name with an asterisk. The following declaration, for example, makes **ptr** a variable that can hold addresses of **long int** variables.

```
long *ptr;
```

The data type, **long** in this case, refers to the type of variable that **ptr** can point to. To assign a pointer variable with the virtual address of a variable, you can use the address-of operator **&**. For instance, the following is legal:

```
long *ptr;
long long_var;
ptr = &long_var;  /*  Assign the address of long_var to ptr.
                  */
```

But this is illegal:

```
long *ptr;
float float_var;
ptr = &float_var; /* ILLEGAL - because ptr can only store the
                   * address of a long int.
                   */
```

The following program illustrates the difference between a pointer variable and an integer variable:

```
/* Program name is "ptr_example1". */

#include <stdio.h>

int main( void )
{
  int j = 1;
  int *pj;

  pj = &j;  /* Assign the address of j to pj */
  printf( "The value of j is: %d\n", j );
  printf( "The address of j is: %d\n", pj );
}
```

The result is:

```
The value of j is: 1
The address of j is: 5219405
```

Dereferencing a Pointer

To **dereference** a pointer (get the value stored at the pointer address), use the * operator. The following program shows how dereferencing works.

```
/* Program name is "ptr_example2". */

#include <stdio.h>

int main( void )
{
  char *p_ch;
  char ch1 = 'A', ch2;

  printf( "The address of p_ch is %d\n", &p_ch );

  p_ch = &ch1;
  printf( "The value of p_ch is %d\n", p_ch );
  printf( "The dereferenced value of p_ch is %c\n",
          *p_ch );
}
```

The output from running this program is:

```
The address of p_ch is 52194052
The value of p_ch is 52194050
The dereferenced value of p_ch is A
```

This is a roundabout and somewhat contrived example that assigns the character 'A' to both **ch1** and **ch2**. It does, however, illustrate the effect of the dereference (*) operator. The variable **ch1** is initialized to 'A'. The first **printf()** call displays the address of the pointer variable **p_ch**. In the next step, **p_ch** is assigned the address of **ch1**, which is also displayed. Finally, we display the dereferenced value of **p_ch** and assign it to **ch2**.

The expression ***p_ch** is interpreted as: "take the address value stored in **p_ch** and get the value stored at that address." This gives us a new way to look at the declaration. The data type in the pointer declaration indicates what type of value results when the pointer is dereferenced. For instance, the declaration

```
float *fp;
```

means that when ***fp** appears as an expression, the result will be a **float** value.

The expression ***fp** can also appear on the left side of an expression:

```
*fp = 3.15;
```

In this case, we are storing a value (3.15) at the location designated by the pointer **fp**. Note that this is different from

```
fp = 3.15;
```

which attempts to store the address 3.15 in **fp**. This, by the way, is illegal since addresses are not the same as integers or floating-point values.

When assigning a value through a dereferenced pointer, it is important to make sure that the data types agree. Consider the following case:

```
/* Program name is "ptr_example3". */

#include <stdio.h>

int main( void )
{
  float f = 1.17e3, g;
  int *ip;

  ip = &f;
  g = *ip;
  printf( "The value of f is: %f\n", f );
  printf( "The value of g is %f\n", g );
}
```

The result is:

```
The value of f is: 1170.000000
The value of g is: 1150435328.000000
```

In the preceding example, instead of getting the value of **f**, **g** gets an erroneous value because **ip** is a pointer to an **int**, not a **float**. The Domain C compiler issues a warning message when a pointer type is unmatched. If you compile the preceding program, for instance, you receive the message:

```
(0005)      ip = &f;
******** Line 5: Warning:  Illegal pointer combination:
                          incompatible types.
No errors, 1 warning, C Compiler, Rev x.yy
```

Pointer Arithmetic

The following arithmetic operations with pointers are legal:

- You may add an integer to a pointer or subtract an integer from a pointer.
- You may use a pointer as an operand to the ++ and -- operators.
- You may subtract one pointer from another pointer.

All other arithmetic operations with pointers are illegal.

When you add or subtract an integer to a pointer, the compiler automatically **scales** the integer to the pointer's type. In this way, the integer always represents the number of *objects* to jump, not the number of *bytes*. For example, consider the following program fragment:

```
int x[10], *p1x = x, *p2x;

p2x = p1x + 3;
```

Since pointer **p1x** points to a variable (x) that is 4 bytes long, then the expression **p1x + 3** actually increments **p1x** by 12 (4 * 3), rather than by 3.

It is legal to subtract one pointer value from another, provided that the pointers point to the same type of object. This operation yields an integer value that represents the number of objects between the two pointers. If the first pointer represents a lower address than the second pointer, the result is negative. For example,

```
&a[3] - &a[0]
```

evaluates to 3, but,

```
&a[0] - &a[3]
```

evaluates to -3.

It is also legal to subtract an integral value from a pointer value. This type of expression yields a pointer value. The following examples illustrate some legal and illegal pointer expressions:

```
long *p1, *p2;
int a[5], j;
char *p3;

p1 = a;        /* Same as p1 = &a[0] */
p2 = p1 + 4;   /* legal */
j = p2 - p1;   /* legal -- j is assigned 4  */
j = p1 - p2;   /* legal -- j is assigned -4 */
p1 = p2 - 2;   /* legal -- p2 points to a[2] */
p3 = p1 - 1;   /* ILLEGAL -- different pointer types*/
j = p1 - p3;   /* ILLEGAL -- different pointer types*/
j = p1 + p2;   /* ILLEGAL -- can't add pointers */
```

Arrays and Pointers

Arrays and pointers have a close relationship in the C language. You can exploit this relationship in order to write more efficient code. See the discussion of "array operations" in this chapter for more information.

Casting a Pointer's Type

A pointer to one type may be cast to a pointer to any other type. For example, in the following statements, a pointer to an **int** is cast to a pointer to a **char**. Presumably, the function **func**() expects a pointer to a **char**, not a pointer to an **int**.

```
int i, *p = &i;

func( (char *) p);
```

As a second example, a pointer to a **char** is cast to a pointer to **struct H**:

```
struct H {int q;} x;
char  *genp = &x;

(struct H*)genp->q
```

See the "casting operations" listing of this encyclopedia for more information about the cast operator.

It is always legal to assign any pointer type to a generic pointer, and vice versa, without a cast. For example:

```
float x, *fp = &x;
int j, *pj = &j;
void *pv;

pv = fp;  /* legal */
pj = pv;  /* legal */
```

In both these cases, the pointers are implicitly cast to the target type before being assigned. See Section 3.7.3 for more information about generic pointers.

Assigning an Integer Value to a Pointer

You may assign an integer value to a pointer, but programs that use this feature are not portable. The following statements assign absolute address **0XFFF13000** to a pointer called **abs_address**.

```
char *abs_address;
abs_address = (char *) OXFFF130000;
```

This feature is generally used to map variables to hardware registers whose addresses are fixed.

Null Pointers

The C language supports the notion of a **null pointer**—that is, a pointer that is guaranteed not to point to a valid object. A null pointer is any pointer assigned the integral value zero. For example:

```
char *p;

p = 0;  /* make p a null pointer */
```

In this one case—assignment of zero—you do not need to cast the integral expression to the pointer type.

Null pointers are particularly useful in control-flow statements since the zero-valued pointer evaluates to false, whereas all other pointer values evaluate to true. For example, the following **while** loop continues iterating until **p** is a null pointer:

```
char *p;
    .
    .
while (p)
{
    .
/* iterate until p is a null pointer */
    .
}
```

This use of null pointers is particularly prevalent in applications that use arrays of pointers, as described later in this chapter.

The compiler does not prevent you from attempting to dereference a null pointer; however, doing so may trigger a run-time access violation. Therefore, if it is possible that a pointer variable is a null pointer, you should make some sort of test like the following when dereferencing it:

```
if (px && *px)    /* if px = 0, expression will short-circuit
    .                before dereferencing occurs*/
    .
    .
```

Null pointers are a portable feature.

EXAMPLE 1

```
/* Program name is "pointer_example1". This program shows how
 * to access a one-dim. array through pointers. Function
 * count_chars returns the number of characters in the
 * string passed to it.
 * Note that *arg     is equivalent to a_word[0];
 *           *arg + 1 is equivalent to a_word[1]...
 */
#include <stdio.h>

int count_chars( char *arg )
{
  int    count = 0;

  while (*arg++)
    count++;
  return count;
}

int main( void )
{
  char a_word[30];
  int number_of_characters;

  printf( "Enter a word -- " );
  scanf( "%s", a_word );
  number_of_characters = count_chars( a_word );
  printf( "%s contains %d characters.\n", a_word,
           number_of_characters );
}
```

EXAMPLE 2

```
/* Program name is "pointer_example2".  This program
 * demonstrates two ways to access a two-dim. array.
 */
#include <stdio.h>

int main( void )
{
  int count = 0, which_name;
  char c1, c2;
  static char str[5][10] = {"Phil", "Sandi", "Barry",
                            "David", "Amy"};
  static char *pstr[5] = { str[0], str[1], str[2],
                           str[3], str[4]};
/* pstr is an array of pointers.  Each element in the array
 * points to the beginning of one of the arrays in str.
 */
/* Prompt for information. */
  printf( "Which name do you want to retrieve?\n" );
  printf( "Enter 0 for the first name,\n" );
  printf( "      1 for the second name, etc. -- " );
  scanf( "%d", &which_name );

/* Print name directly through array. */
  while (c1 = str[which_name][count++])
    printf( "%c", c1 );
  printf("\n");

/* Print same name indirectly through an array of pointers. */
  while (c2 = *(pstr[which_name]++))
    printf("%c", c2);
/* We could have also used the following statement instead of
 * the two previous ones:  printf( "%s", pstr[which_name] );
 */
  printf( "\n" );
}
```

USING THESE EXAMPLES

If we execute the first program, we get the following output:

```
Enter a word -- Marilyn
Marilyn contains 7 characters.
```

If we execute the second program, we get the following output:

```
Which name do you want to retrieve?
Enter 0 for the first name,
      1 for the second name, etc. -- 1
Sandi
Sandi
```

predefined macros Provide information about the compiler or compilation environment.

DESCRIPTION

The Domain compilers support a number of predefined macros that provide information about the compiler or about the compilation environment. In addition to the macros described in this section, Domain C also supports the following predefined macros:

__FILE__	Expands to the source file name.
__LINE__	Expands to the current line number in the source file.
__DATE__	Expands to the current date (of compilation).
__TIME__	Expands to the current time (of compilation).
__STDC__	Expands to 1 if ANSI-style function prototyping is in effect.
_BFMT__COFF	Expands to 1 if the compiler is producing COFF object code.

For more information about the **__FILE__** and **__LINE__** macros, see the entry under **__FILE__** in this chapter. For more information about the **__DATE__** and **__TIME__** macros, see the entry under **__DATE__** in this chapter. For more information about **__STDC__** and **_BFMT__COFF**, see the entry under **__STDC__** later in this chapter.

relational operators Compare the values of two expressions.

FORMAT

exp1 > *exp2*	Greater than
exp1 >= *exp2*	Greater than or equal to
exp1 < *exp2*	Less than
exp1 <= *exp2*	Less than or equal to
exp1 == *exp2*	Equal to
exp1 != *exp2*	Not equal to

ARGUMENTS

exp1	Any expression.
exp2	Any expression.

DESCRIPTION

The relational operators perform the same way in C as they do in everything from fourth-grade arithmetic to Advanced Programming II. The two that are slightly unusual are == and !=, but even in these cases the differences are a matter of form, not substance.

The equality operator (==) performs the same function as Pascal's = or FORTRAN's .EQ.; it just looks different. Note that although the equality operator looks similar to the assignment operator (=), the two operators serve completely different purposes. Use the assignment operator when you want to assign a value to a variable, but use the equality operator when you want to test the value of an expression.

Bug Alert: Confusing = with ==

One of the most common mistakes made by beginners and experts alike is to use the assignment operator (=) instead of the equality operator (==). For instance:

```
while (j = 5)
  do_something();
```

What is intended, clearly, is that the do_something() function should only be invoked if **j** equals five. It should been written:

```
while (j == 5)
  do_something();
```

Note that the first version is syntactically legal since all expressions have a value. The value of the expression **j = 5** is 5. Since this is a nonzero value, the **while** expression will always evaluate to true and do_something() will always be invoked.

Note, however, that **#section** directives do not affect variables with fixed duration. Static data that has file scope resides in the module's section regardless of any **#section** directives. All global variables reside in special sections that cannot be affected by **#section** directives. If you are compiling with **/bin/cc**, initialized global variables are placed in **.data** and uninitialized global variables are put in **.bss**. If you are compiling with **/com/cc**, the compiler creates a named section for each global variable. You can override these defaults by using the **#attribute[section]** modifer, which is described in Chapter 3.

The following example illustrates the **#section** directive.

```
#module section_example, psection1 , dsection1
main()
{
.
.
.
}

#section(psection2)  /* dsection1 is still the active data
                        section */
void func1()
{
.
.
.
}

#section(psection1, dsection2)
void func2()
{
.
.
.
}

#section(,,dsection1)
void func3()
{
.
.
.
}
```

The preceding example creates four named sections that contain the program segments shown in the following chart.

Table 4–13. Example of **#section** *Directive*

Named Section	What It Contains
psection1	program instructions from main(), func2(), and func3()
psection2	program instructions from func1()
dsection1	data from main(), func1(), and func3()
dsection2	data from func2()

Note that all of these operators have lower precedence than the arithmetic operators. The expression,

```
a + b * c < d / f
```

is evaluated as if it had been written:

```
(a + (b * c)) < (d / f)
```

Among the relational operators, >, >=, <, and <= have the same precedence. The == and != operators have lower precedence. All of the relational operators have left-to-right associativity. Table 4-11 illustrates how the compiler parses complex relational expressions.

Table 4-11. Examples of Expressions Using the Relational Operators

Given the following declarations:

```
int j = 0, m = 1, n = -1;
float x = 2.5, y = 0.0;
```

Expression	Equivalent Expressions	Result
j > m	j > m	0
m / n < x	(m / n) < x	1
j <= m >= n	((j <= m) >=n)	1
j <= x == m	((j <= x) == m)	1
- x + j == y > n > m	((-x) + j) == ((y > n) >= m)	0
x += (y >= n)	x = (x + (y >= n))	3.5
++j == m != y * 2	((++j) == m) != (y * 2)	1

Relational expressions are often called **Boolean expressions**, in recognition of the nineteenth century mathematician and logician, George Boole. Many programming languages, such as Pascal, have Boolean data types for representing **true** and **false**. The C language, however, represents these values with integers. Zero is equivalent to FALSE, and any nonzero value is considered **true**.

The value of a relational expression is an integer, either 1 (indicating the expression is **true**) or 0 (indicating the expression is **false**). The examples in Table 4-12 illustrate how relational expressions are evaluated.

Table 4-12. Relational Expressions

Expression	Value
-1 < 0	1
0 > 1	0
5 = 5	1
7 != -3	1
1 >= -1	1
1 >10	0

Because Boolean values are represented as integers, it is perfectly legal to write:

```
if (j)
  statement;
```

If **j** is any nonzero value, *statement* is executed; if **j** equals zero, *statement* is skipped. Likewise, the statement,

```
if (isalpha( ch ))
```

is exactly the same as:

```
if (isalpha( ch ) != 0)
```

The practice of using a function call as a Boolean expression is a common idiom in C. It is especially effective for functions that return zero if an error occurs, since you can use a construct such as:

```
if (func())
  proceed;
else
  error handler;
```

Bug Alert: Comparing Floating-Point Values

It is very dangerous to compare floating-point values for equality because floating-point representations are inexact for some numbers. For example, the following expression, though algebraically true, will evaluate to false on most computers:

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0) == 1.0
```

This evaluates to 0 (false) because the fraction **1.0/3.0** contains an infinite number of decimal places (3.33333...). The computer is only capable of holding a limited number of decimal places, so it rounds each occurrence of 1/3. As a result, the left-hand side of the expression does not equal 1.0 exactly.

This problem can occur in even more subtle ways. Consider the following code:

```
double divide( double num, double denom )
{
  return num/denom;
}

int main( void )
{
  double c, a = 1.0, b = 3.0;
  c = a/b;
  if (c != divide(a,b))
    printf("Fuzzy doubles\n");
}
```

Surprisingly, the value stored in **c** will not equal the value returned by **divide()**. This anomaly occurs due to the fact that the computer can represent more decimal places for values stored in registers than for values stored in memory. Because the value returned by **divide()** is never stored in memory, it is not equal to the value **c**, which has been rounded for memory storage.

To avoid bugs caused by inexact floating-point representations, you should refrain from using strict equality comparisons with floating-point types.

EXAMPLE

```
/*  Program name is "relational_example". This program simply
 *  does some mathematical calculations and along the way
 *  shows C's relational operators in action.
 */
#include <stdio.h>

int main( void )
{
  int num, i;

  printf( "\n" );
  num = 5;
  printf( "The number is: %d\n", num );
  for (i = 0; i <= 2; i++)
  {
    if (num < 25)
    {
      num *= num;
      printf( "The number squared is: %d\n", num );
     }
    else if (num == 25)
    {
      num *= 2;
      printf( "Then, when you double that, you get: %d\n", num );
    }
    else if (num > 25)
    {
      num -= 45;
      printf( "And when you subtract 45, you're back where " );
      printf( "you started at: %d\n", num );
    }
  } /*  end for      */

  if (num != 5)
    printf( "The programmer made an error in setting up this \
example\n");
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
The number is: 5
The number squared is: 25
Then, when you double that, you get: 50
And when you subtract 45, you're back where you started at: 5
```

return The mechanism for exiting from a called function.

FORMAT

```
return;          /* first form     */
return exp;      /* second form    */
```

ARGUMENTS

exp	Any valid C expression.

DESCRIPTION

The **return** statement causes a C program to exit from the function containing the **return** and go back to the calling block. It may or may not have an accompanying *exp* to evaluate. If there is no *exp*, the function returns an unpredictable value.

Functions can return only a single value directly via the **return** statement. The return value can be any type except an array or function. This means that it is possible to indirectly return more than a single value by passing a pointer to an aggregate type. It is also possible to return a structure or union directly, though Domain C implements this by returning a pointer to the structure or union.

A function may contain any number of **return** statements. The first one encountered in the normal flow of control is executed, and causes program control to be returned to the calling routine. If there is no **return** statement, program control returns to the calling routine when the right brace of the function is reached. In this case, the value returned is undefined.

The return value must be assignment-compatible with the type of the function. This means that the compiler uses the same rules for allowable types on either side of an assignment operator to determine allowable return types. For example, if **f()** is declared as a function returning an **int**, it is legal to return any arithmetic type, since they can all be converted to an **int**. It would be illegal, however, to return an aggregate type or a pointer, since these are incompatible types. The following example shows a function that returns a **float**, and some legal return values.

```
float f( void )
{
  float f2;
  int a;
  char c;

  f2 = a;       /* OK, quietly converts a to float */
  return a;     /* OK, quietly converts a to float */
  f2 = c;       /* OK, quietly converts c to float */
  return c;     /* OK, quietly converts c to float */
}
```

The C language is pickier about matching pointers. In the following example, **f()** is declared as a function returning a pointer to a **char**. Some legal and illegal **return** statements are shown.

```
char *f()
{
  char **cpp, *cp1, *cp2, ca[10];
  int *ip1, *ip2;

  cp1 = cp2;        /* OK, types match */
  return cp2;       /* OK, types match */
  cp1 = *cpp;       /* OK, types match */
  return *cpp;      /* OK, types match */

/*  An array name without a subscript gets converted
 *  to a pointer to the first element.
 */
  cp1 = ca;      /* OK, types match */
  return ca;     /* OK, types match */

  cp1 = *cp2;    /* Error, mismatched types         */
                 /* (pointer to char vs. char )     */
  return *cp2;   /* Error, mismatched types         */
                 /* (pointer to char vs. char )     */
  cp1 = ip1;     /* Error, mismatched pointer types */
  return ip1;    /* Error, mismatched pointer types */
  return;        /* Produces undefined behavior --  */
                 /* should return (char *)          */
}
```

Note in the last statement that the behavior is undefined if you return nothing. The only time you can safely use **return** without an expression is when the function type is **void**. Conversely, if you return an expression for a function that is declared as returning **void**, you will receive a compile-time error.

EXAMPLE

```
/*  Program name is "return_example". This program finds the
 *  length of a word that is entered.
 */
#include <stdio.h>

int find_length( char *string )
{
  int i;

  for (i = 0; string[i] != '\0'; i++)
    ;
  return i;
}

int main( void )
{
  char string[132];
  int result;

  printf( "This program finds the length of any word you ");
  printf( "enter.\n" );
  printf( "Enter the word: " );
  gets( string );
  result = find_length( string );
  printf( "This word contains %d characters.\n", result );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
This program finds the length of any string you enter.

Enter the string: Copenhagen
The string is 10 characters.
Again? y

Enter the string: galaxy
The string is 6 characters.
Again? n
```

#section (preprocessor directive) Directs the binder to place code and data into the specified sections. (Domain Extension)

FORMAT

#section([*psect_name,*] *dsect_name* **)**

#section(*psect_name* [*,dsect_name*] **)**

ARGUMENTS

psect_name Optional, but you must include a *psect_name* or a *dsect_name*, or both. If you do include a *psect_name*, it must be an identifier. This identifier is the name of the procedure section that the code will go into.

dsect_name Optional, but you must include a *psect_name* or a *dsect_name*, or both. If you do include a *dsect_name*, it must be an identifier. This identifier is the name of the data section that the data will go into.

DESCRIPTION

The **#section** directive instructs the linker to place instructions and data into named sections rather than the default sections. Every object module is composed of at least three sections: a procedure section, a data section, and a debug section. By default, the name of the procedure section is **.text**, and the names of the data sections are **.data** and **.bss**. The **#section** directive, as well as the #module directive, allow you to create additional sections. You can use this capability to group together code or data that is used frequently. This way the system need not swap extra pages in and out of memory to execute a program. For more information about sections and the object file format, see the *Domain/OS Programming Environment Reference* manual.

Note that the following preprocessor directive is illegal:

```
#section()
```

#section directives may appear anywhere in a file except within a function. Section names defined in a **#section** directive are in effect until the end of the file or until another **#section** directive redefines the current section names. By specifying the same section names in different source files, you can ensure that the resulting object code is grouped together in virtual memory.

sizeof Unary operator that finds the size of an object.

FORMAT

sizeof *exp*;
sizeof (*type_name*)

ARGUMENTS

exp	This is any expression.
type_name	This is the name of a predefined or user-defined data type, or the name of some variable. An example of a predefined data type is **int**. A user-defined data type could be the tag name of a structure.

DESCRIPTION

The **sizeof** operator accepts two types of operands: an expression or a data type. However, the expression may not have type function or **void**, or be a bit field. Moreover, the expression itself is not evaluated—the compiler only determines what type the result would be. Any side effects in the expression, therefore, will not have an effect. The result type of the **sizeof** operator is **unsigned int**.

If the operand is an expression, sizeof returns the number of bytes that the result occupies in memory:

```
/*  Returns the size of an int (4 if ints are four
 *  bytes long)
 */
sizeof(3 + 5)

/*  Returns the size of a double (8 if doubles are
 *  eight bytes long)
 */
sizeof(3.0 + 5)
```

For expressions, the parentheses are optional, so the following is legal:

```
sizeof x
```

By convention, however, the parentheses are usually included.

The operand can also be a data type, in which case the result is the length in bytes of objects of that type:

```
sizeof(char)     /* 1 on all machines */
sizeof(short)    /* 2 on Domain machines */
sizeof(float)    /* 4 on Domain machines */
sizeof(int *)    /* 4 on Domain machines */
```

The parentheses are required if the operand is a data type. Note that the results of most **sizeof** expressions are implementation dependent. The only result that is guaranteed is the size of a **char**, which is always 1.

In general, the **sizeof** operator is used to find the size of aggregate data objects such as arrays and structures.

EXAMPLE

You can also use the **sizeof** operator to obtain information about the sizes of objects in your C environment. The following, for example, prints the sizes of the basic data types:

```
/*  Program name is "sizeof_example".  This program
 *  demonstrates a few uses of the sizeof operator.
 */
#include <stdio.h>

int main( void )
{
  printf( "TYPE\t\tSIZE\n\n" );
  printf( "char\t\t%d\n", sizeof(char) );
  printf( "short\t\t%d\n", sizeof(short) );
  printf( "int\t\t%d\n", sizeof(int) );
  printf( "float\t\t%d\n", sizeof(float) );
  printf( "double\t\t%d\n", sizeof(double) );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
TYPE            SIZE

char            1
short           2
int             4
float           4
double          8
```

__STDC__ and _BFMT__COFF (predefined names)

FORMAT

__STDC__	If equal to 1, indicates that this compiler conforms to the ANSI standard. (Note that there are two underscores before and two underscores after this preprocessor symbol.)
_BFMT__COFF	If defined, indicates that this compiler generates COFF

DESCRIPTION

The **__STDC__** macro, if it expands to 1, signifies that the compiler conforms to the ANSI Standard. If it expands to any other value, or if it is not defined, you should assume that the compiler does not conform to the ANSI standard. A common use of **__STDC__** is to choose between the old function declaration syntax and the new ANSI prototyping syntax:

```
#if (__STDC__ == 1)
  extern int foo( char a, float b );
  extern *char goo( char *string );
#else
  extern int foo();
  extern *char goo();
#endif
```

If the compiler conforms to the ANSI standard (**__STDC__** equals 1), we use the prototyping syntax to declare the types of each argument. Otherwise, we use the old function declaration syntax. By default, **__STDC__** *is* defined unless you compile with the **-ntype** option (available with **/com/cc** only).

The **_BFMT__COFF** macro will be defined as 1 for compilers that generate COFF (as opposed to obj) code. For compilers that do not produce COFF, the macro will be undefined. Therefore, you can test the compiler with either an **#if** or an **#ifdef** directive:

```
#ifdef _BFMT__COFF     /* Use #attribute to create overlay
                        * section */
  struct { int a;
           float b;
         } overlay #attribute[section(overlay)];
#else
  struct { int a;
           float b;
         } overlay;
#endif
```

structure and union operations Operations that can be performed on structures and unions, and structure and union members

DESCRIPTION

In Chapter 3, we explained how to define structure and union variables. In this section, we show how to use structure and union variables in the body of a function.

Domain C allows the following uses of structures and unions:

- You can reference a member of a structure or union.
- You can find the address of a structure or union with the address-of operator &.
- You can find the size of a structure or union with the **sizeof** operator.
- You can assign a structure or union to another structure or union of the same type.
- You can define a function that returns a structure or union.
- You can pass a structure or union as an argument to a function.

The following sections detail these uses.

Referencing Structure and Union Members

There are two methods for referencing a member of a structure or union, depending on whether you have the structure itself or a pointer to the structure. Each method uses a special operator. If you have the structure itself, you can enter the structure name and field name separated by the **dot (.) operator**. For instance, suppose you make the following declaration:

```
struct vitalstat
{
   char  vs_name[19], vs_ssnum[11];
   short vs_month, vs_day, vs_year;
} vs, *pvs = &vs;
```

To assign the date, March 15, 1987 to vs, you would write:

```
vs.vs_month = 3;
vs.vs_day = 15;
vs.vs_year = 1987;
```

The referenced field expression is just like any other variable, so you can use **vs.vs_month** anywhere you would normally use a **short** variable.

The following statement, for instance, is perfectly legal:

```
if (vs.vs_month > 12 || vs.vs_day > 31)
  printf( "Illegal Date.\n" );
```

The other way to reference a structure member is indirectly through a pointer to the structure. To reference a member through a pointer, use the **right-arrow operator (->)**, which is formed by entering a dash followed by a right angle bracket. For example:

```
if (pvs->vs_month > 12 || pvs->vs_day > 31)
  printf( "Illegal Date.\n" );
```

The right-arrow operator is actually a shorthand for dereferencing the pointer and using the dot operator. That is,

```
pvs->vs_day
```

is the same as:

```
(*pvs).vs_day
```

The pointer to a struct or union is usually a pointer variable, but Domain C also allows it to be an integer that contains the absolute address of a structure or union. (Using an integer in this context is not a portable feature; trying it triggers a warning.)

Operations on Structure and Union Members

In general, you can perform any operation on a structure member that you can on a normal variable of the same type. The only restriction is that you may *not* take the address of a bit field.

Structure and Union Assignment

Although it is not supported in the original K&R standard, Domain C and the ANSI Standard allow you to assign a structure or union to a structure or union variable, provided they share the same type. The following code extract shows some examples of structure assignments.

```
struct {
         int a;
         float b;
       } s1, s2, sf(), *ps;
       .
       .
s1 = s2;
s2 = sf();
ps = &s1;
s2 = *ps;
       .
       .
```

Referencing Nested Members

Domain C allows you to access nested members of structures and unions without specifying the inner structure name. You need only enter the outer structure name and the member name you want.

Consider the following nested structure:

```
struct {
         int a;
         struct {
                  float b,c;
                } in;
       } out;
```

Domain C provides two ways to access component **b**. First, you can use the traditional C method as shown below:

```
out.in.b
```

Second, you can leave out the inner structure name, as in

```
out.b
```

If the same name appears more than once in a structure with inner structures and you give only the component name, the compiler warns you that the reference is ambiguous. For example, consider the following definition:

```
struct {
     union { int a,b;
           } first_union;
     union { char a,b;
           } second_union;
       } outer_struct;
```

The reference **outer_struct.a** is ambiguous since it is not clear whether it refers to **outer_struct.first_union.a** or **outer_struct.second_union.a**. If you use **outer_struct.a** as a reference, the compiler issues the following warning message:

```
Warning:  Ambiguous reference; more than one member named "a".
```

> **NOTE:** This feature is not supported by the ANSI standard. Use the **-std** compiler option to identify these nonportable usages in source code.

Passing Structures as Function Arguments

There are two ways to pass structures as arguments: pass the structure itself (called **pass by value**) or pass a pointer to the structure (called **pass by reference**). The two methods are shown in the following example.

```
VITALSTAT vs;
     .
     .
func( vs );    /*  Pass by value -- Passes an entire
                *  copy of the structure.
                */
func( &vs );   /*  Pass by reference -- Passes the
                *  address of a structure.
                */
     .
     .
```

Passing the address of a structure is usually faster because only a single pointer is copied to the argument area. Passing by value, on the other hand, requires that the entire structure be copied. There are only two circumstances when you should pass a structure by value:

- The structure is very small (approximately the same size as a pointer).
- You want to guarantee that the called function does not change the structure being passed. (When an argument is passed by value, the compiler generates a copy of the argument for the called function. The called function can only change the value of the copy, not the value of the argument on the calling side.)

In all other instances, you should pass structures by reference.

> **NOTE:** Passing structures by value, though supported in almost all C compilers, is not part of the original K&R standard. It is required by the ANSI Standard.

Depending on which method you choose, you need to declare the argument on the receiving side as either a structure or a pointer to a structure:

```
func( vs )
VITALSTAT vs;  /*  Pass by value -- the argument is a
                *  structure .
                */
```

or

```
func( pvs )
VITALSTAT *pvs;  /* Pass by reference -- the argument
                  * is a pointer to a structure.
                  */
```

Note that the argument-passing method you choose determines which operator you should use in the function body—the dot operator if a structure is passed by value, and the right-arrow operator if the structure is passed by reference.

Bug Alert: Passing Structures vs. Passing Arrays

Passing structures is *not* the same as passing arrays. This inconsistency in the C language can cause confusion.

To pass an array in C, you simply specify the array name without a subscript. The compiler interprets the name as a pointer to the initial element of the array so it really passes the array by reference. There is no way to pass an array by value (except to embed it in a structure and pass the structure by value).

With structures, however, the structure name is interpreted as the entire structure, not as a pointer to the beginning of the structure. If you use the same syntax that you use with arrays, therefore, you will get different semantics. For example:

```
int ar[100];
struct tag st;
        .
        .
func( ar );     /* Passes a pointer to the first
                   element of ar[] */
func( st );     /* Passes an entire structure */
```

The inconsistency follows through to the receiving side. For example, the following two array versions are the same:

```
func( ar )
int ar[];  /* ar is converted to a pointer
              to an int */

func( ar )
int *ar;   /* ar is a pointer to an int */
```

But the following two structure versions are very different:

```
func( st )
struct tag st;  /* st is an entire structure */

func( st )
struct tag *st;  /* st is a pointer to
                    a struct */
```

Returning Structures

Just as it is possible to pass a structure or a pointer to a structure, it is also possible to return a structure or a pointer to a structure. (Returning a structure is not supported in the original K&R standard, but is a common extension supported by most C compilers and by the ANSI standard.) The declaration of the function's return type must agree with the actual returned value. For example:

```
struct tag f()    /* Define a function that returns */
{                 /* a struct */
  struct tag st;
    .
    .
  return st;      /* Return an entire struct */
}

struct tag *f1() /* Define a function that returns */
                 /* a pointer to a struct */
{
  static struct tag pst;
    .
    .
  return &pst;    /* Return the address of a struct */
}
```

As with passing structures, you generally want to return pointers to structures because it is more efficient. Note, however, that if you return a pointer to a structure, the structure must have fixed duration. Otherwise, it will cease to be valid once the function returns.

One situation where returning structures is particularly useful is when you want to return more than one value. The **return** statement can only send back one expression to the calling routine, but if that expression is a structure or a pointer to a structure, you can indirectly return any number of values. The following function, for instance, returns the sine, cosine, and tangent of its argument. The functions **sin()**, **cos()** and **tan()** are part of the run-time library. Each accepts an argument measured in radians and returns the corresponding trigonometric value. If the argument is too large, however, the results will not be meaningful.

```
#include <stdio.h>
#include <math.h>  /* include file for trig */
                   /* functions */
#define too_large 100 /*  Differs from one machine */
                      /*  to another. */
typedef struct
{
  double sine, cosine, tangent;
} TRIG;

TRIG *get_trigvals( radian_val )
double radian_val;
{
  static TRIG result;

/*  If radian_val is too large, the sine, cosine and
 *  tangent values will be meaningless.
 */
  if (radian_val > TOO_LARGE)
  {
    printf( "Input value too large -- cannot return \
meaningful results\n" );
    return NULL; /* return null pointer -- defined in
                  * stdio.h.
                  */
  }

  result.sine = sin( radian_val );
  result.cosine = cos( radian_val );
  result.tangent = tan( radian_val );
  return &result;
}
```

Referencing a Member Through a Pointer to Another Structure

To be compatible with older of versions of C that did not create separate name spaces for every structure and union, Domain C allows you to access members through pointers to other structures and unions. That is, the member name can be a member of *any* structure, not just the structure of which it is a member. The following program fragment demonstrates this unusual feature of C:

```
struct a { int a;
           char b;
         } x = {'ABCD','E'};

struct b { char aa;
           int bb;
         } y;

main()
{
   int *i;
   i = &y;
   printf("%c\n",i->a);
}
```

Note that pointer variable **i** holds the address of structure variable **y**. Further note that **a** is a member of structure **x**, not structure **y**. Yet, we are able to refer to **i->a** in the **printf()** call. When C encounters **i->a**, it looks for any structure member whose name is **a**. If we execute the preceding program, we get the following output:

```
A
```

NOTE: This functionality is not a portable feature since most modern C compilers do not support it.

switch A conditional branching statement that selects among several statements based on constant values.

FORMAT

```
switch ( exp )
{
  case const_exp : [statement...]
[case const_exp : [statement...]]
[default : [statement...]]
}
```

ARGUMENTS

exp	The integer expression that the **switch** statement evaluates and then compares to the values in all the **cases**.
const_exp	An integer expression to which *exp* is compared. If *const_exp* matches *exp*, the accompanying *statement* is executed.
statement	This is zero or more simple statements. (Note that if there is more than one simple statements, you do not need to enclose the statements in braces.)

DESCRIPTION

The expression immediately after the **switch** keyword must be enclosed in parentheses and must be an integral expression. That is, it can be **char, short, int** or **long,** but not **float, double,** or **long double.**

> **NOTE:** the K&R standard requires the **switch** expression to be of type **int.**

The expressions following the **case** keywords must be integral constant expressions, meaning they may not contain variables.

The semantics of the **switch** statement are straightforward. The **switch** expression is evaluated; if it matches one of the **case** labels, program flow continues with the statement that

follows the matching **case** label. If none of the **case** labels match the **switch** expression, program flow continues at the **default** label, if it exists. (Strictly speaking, the **default** label need not be the last label, though it is good style to put it last.) No two **case** labels may have the same value.

An important feature of the **switch** statement is that program flow continues from the selected case label until another control-flow statement is encountered or the end of the **switch** statement is reached. That is, the compiler executes any statements following the selected **case** label until a **break**, **goto**, or **return** statement appears. The **break** statement explicitly exits the **switch** construct, passing control to the statement following the **switch** statement. Since this is usually what you want, you should almost always include a **break** statement at the end of the statement list following each **case** label.

The following **print_error()** function, for example, prints an error message based on an error code passed to it.

```
/*  Prints error message based on error_code.
 *  Function is declared with void because it doesn't
 *  return anything.
 */

#include <stdio.h>
#define ERR_INPUT_VAL 1
#define ERR_OPERAND 2
#define ERR_OPERATOR 3
#define ERR_TYPE 4

void print_error( error_code )
int error_code;
{
  switch (error_code)
  {
    case  ERR_INPUT_VAL:
             printf("Error: Illegal input value.\n");
             break;
    case  ERR_OPERAND:
             printf("Error: Illegal operand.\n");
             break;
    case  ERR_OPERATOR:
             printf("Error: Unknown operator.\n");
             break;
    case  ERR_TYPE:
             printf("Error: Incompatible data.\n");
             break;
    default: printf("Error: Unknown error code %d\n",
                     error_code);
             break;
  }
}
```

The **break** statements are necessary to prevent the function from printing more than one error message. The last **break** after the default case isn't really necessary, but it is a good idea to include it anyway for consistency's sake.

Sometimes you want to associate a group of statements with more than one case value. To obtain this behavior, you can enter consecutive case labels. The following function, for instance, returns 1 if the argument is a punctuation character, or zero if it is anything else.

```
/*  This function returns 1 if the argument is a
 *  punctuation character.  Otherwise, it returns
 *  zero.
 */

is_punc( arg )
char arg;
{
  switch (arg)
  {
    case '.':
    case ',':
    case ':':
    case ';':
    case '!':  return 1;
    default :  return 0;
  }
}
```

Domain C allows the use of **enum** values as the control expressions and case labels of a **switch** statement. However, if you use an **enum** one place, you must use use **enum**s elsewhere. That is, if *expr* is of type **enum**, then all the case labels must also be of type **enum**, while if expr is not an **enum**, none of the case labels may be **enum**s. For example:

```
/* Program name is "enums_in_a_switch". */
#include <stdio.h>

int main( void )
{
  enum AUTHORS { Hemingway, Steinbeck, Twain };
  enum AUTHORS favorite = { Twain };

  switch (favorite)
  {
    case Hemingway: printf( "A Farewell To Arms\n" );
                    break;
    case Steinbeck: printf( "The Grapes of Wrath\n" );
                    break;
    case Twain: printf( "The Adventures of Tom Sawyer\n" );
                break;
/*  case 5 : printf("no author")  THIS WOULD BE ILLEGAL SINCE
 *                                5 IS NOT AN ENUM VALUE.
 */
  }
}
```

EXAMPLE

```
/* program name is "switch_example".  Read a student's grade
 * from the keyboard. Then the switch statement uses the grade
 * to decide which comment should be printed. Notice that the
 * cases allow for uppercase and lowercase letters to be
 * entered.
 */
#include <stdio.h>

int main( void )
{
  char answer, grade;

  answer = 'y';
  printf( "\n\n" );
  while (answer == 'y' || answer == 'Y')
  {
    printf( "Enter student's grade: " );
    fflush( stdin );
    scanf( "%c", &grade );
    printf( "\nComments: " );
    switch (grade)
    {
      case 'A':
      case 'a':
            printf( "Excellent\n" );
            break;
      case 'B':
      case 'b':
            printf( "Good\n" );
            break;
      case 'C':
      case 'c':
            printf( "Average\n" );
            break;
      case 'D':
      case 'd':
            printf( "Poor\n" );
            break;
      case 'F':
      case 'f':
            printf( "Failure\n" );
            break;
      default:
            printf( "Invalid grade\n" );
            break;
    }           /*  end switch */
    printf( "\nAgain? " );
    fflush( stdin );
    scanf( "%s", &answer );
```

```
  } /*  end while */
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
Enter student's grade: B

Comments: Good

Again? y
Enter student's grade: c

Comments: Average

Again? n
```

#systype (preprocessor directive) and the systype() macro Selects the target operating system. (Domain Extension)

FORMAT

#systype *systype_name* Preprocessor directive

systype(*systype_name* **)** Predefined macro

ARGUMENTS

systype_name A string containing the name of an operating system. The string must be one of the following:

- **bsd4.1** Berkeley 4.1BSD (obsolete)
- **bsd4.2** Berkeley 4.2BSD
- **bsd4.3** Berkeley 4.3BSD
- **sys3** AT&T System III (obsolete)
- **sys5** AT&T System V Release 2
- **sys5.3** AT&T System V Release 3
- **any** program is independent of a particular UNIX system

DESCRIPTION

We divide this listing into an explanation of the preprocessor directive and the macro. First, we describe the preprocessor directive.

The #systype Preprocessor Directive

Because C programs are often written to run in UNIX environments, and because not all UNIX environments are the same, Domain C supports the **#systype** preprocessor directive, which allows you to define the version of the UNIX system for which your program is targeted.

The Domain C library contains two sets of routines. One is compatible with the Bell Labs versions of the UNIX system (System V, Release 2 and 3) and the other set is compatible with Berkeley's versions of the UNIX system (4.2BSD, and 4.3BSD). All of the routines in both sets work properly in any Domain/OS environment. However, you may encounter problems if you attempt to mix functions from two sets that interact with each other. In general, it is best to choose one set and stick with it whenever possible.

The two sets of functions overlap to a large extent. It is sometimes the case, however, that while function x exists in both sets, the semantics of the function (and in some cases its arguments) may be subtly different. As an illustration, consider the function **setgrp()**. In the System V version, the function definition is:

```
int setpgrp()
```

It is defined to set the process group ID of the calling process to the process ID of the calling process and return the new process group ID. In the 4.2BSD version of the UNIX system, there is an identically named function with similar semantics but a different calling sequence. The Berkeley function,

```
setpgrp( pid, pgrp )
int pid, pgrp;
```

sets the process group of the specified **pgrp**. Zero is returned if successful; –1 is returned and **errno** is set on failure.

To avoid unexpected behavior, always know which set of functions you are accessing. The system chooses one set of functions over another based on a version selector called the **systype**. The **systype** affects both the compilation and the execution of a program. At compilation time, it determines which include files the compiler uses. At run–time, it determines which set of functions are called and makes sure that the proper calling conventions are employed.

The compiler stamps the object module with the **systype** that was in effect when the module was compiled. When the program is executed, the loader checks this stamp and uses the semantics and calling sequences of the designated **systype** when invoking library functions.

There are several ways to define the **systype**, one of which is to place a **#systype** directive in the source file. You may define the **systype** only once per source file. Any subsequent definitions produce an error. Moreover, the **#systype** directive must be the first non–comment token in the source file.

For instance, to set the **systype** to 4.2BSD, enter the following at the top of your source file:

```
#systype bsd4.2
```

It is also legal to enclose the **systype** in double quotes:

```
#systype "bsd4.2"
```

You also can define the target operating system with the **–systype** compile option (**/com/cc** only), which is described in Chapter 6. If you specify one **systype** on the command line and a different one in the file, the compiler reports an error. If you do not explicitly specify a **systype**, the compiler inherits the **systype** from an environment variable called **COMPILESYSTYPE**. By default, this variable is set to **sys5**. If, for some reason, the **COMPILESYSTYPE** variable does not exist, the systype is inherited from another environment variable called **SYSTYPE**. This variable is always set. These environment variables are described in more detail in the *Using the SysV Environment* and *Using the BSD Environment* manuals.

> **NOTE:** Be especially careful about using **systype any**. Most programs are *not* independent of a particular version. For example, programs running under the Aegis environment are **systype sys5**.

The systype Macro

Domain C supports a macro called **systype** that enables you to find out what the current UNIX **systype** is. By default, the **systype** is "**sys5**", but you can change it with the **#systype** preprocessor directive or with the **-systype** compiler option.

The macro may be used only in an **#if** preprocessor directive. It evaluates to 1 if the argument is the same as the **systype**, and evaluates to zero (0) if the argument differs from the **systype**. The quotes around the argument are optional. For example:

```
#if systype("bsd4.2")
#include "comments.4.2"
#else
#if systype(bsd4.3)
#include "comments.4.3"
#else
#include "comments.bell"
#endif
#endif
```

__TIME__ (predefined symbol) See the __DATE__ and __TIME__ listing earlier in this chapter.

while Executes the statements within a loop as long as the specified condition is true.

FORMAT

while (*exp*)
statement

ARGUMENTS

exp	Any expression.
statement	Any simple or compound statement.

DESCRIPTION

This is one of the three looping constructions available in C. Like the **for** loop, the **while** tests *exp* and if it is true (nonzero), *statement* is executed. Once *exp* becomes false (zero), execution of the loop stops. Since *exp* could be false the first time it is tested, *statement* may not be performed even once.

The following describes two ways to jump out of a **while** loop prematurely (that is, before *exp* becomes false):

- Use **break** to transfer control to the first statement following the **while** loop.
- Use **goto** to transfer control to some labeled statement outside the loop.

EXAMPLE

```
/*  program name is "while_example". */
#include <stdio.h>

int main( void )
{
  int count = 0, count2 = 0;
  char a_string[80], *ptr_to_a_string = a_string;

  printf( "Enter a string -- " );
  gets( a_string );

  while (*ptr_to_a_string++)
    count++;   /* A simple statement loop */
  printf( "The string contains %d characters.\n", count );
  printf( "The first word of the string is " );

  while (a_string[count2] != ' ')
  {           /* A compound statement loop */
    printf ( "%c", a_string[count2] );
    count2++;
  }
  printf( "\n" );
}
```

USING THIS EXAMPLE

If we execute this program, we get the following output:

```
$ while_example.bin
Enter a string -- Four score and seven years ago
The string contains 30 characters.
The first word of the string is Four
$
```

Chapter 5

Functions

The main organizational unit of C is the **function**. Functions can appear in a program in three forms:

Function Definition	A declaration that actually defines what the function does, as well as the number and type of arguments.
Function Allusion	Declares a function that is *defined* elsewhere. A function allusion specifies what kind of value the function returns. (With the new prototyping feature, discussed in Section 5.4, it is also possible to specify the number and types of arguments in a function allusion.)
Function Call	Invokes a function, causing program execution to jump to the invoked function. When the called function returns, execution resumes at the point just after the call.

This chapter discusses function definitions, allusions, and calls, and other topics associated with functions, such as recursion and pointers to functions.

5.1 Function Definitions

The syntax of a function definitions is shown below:

[**static**] [*return_type*] *function_name* ([*arg_name* [, *arg_name...*]])

[*arg_declaration*]

[*arg_declaration...*]

{
function_body
}

You can specify any number of arguments, including zero. The return type defaults to **int** if you leave it blank. However, even if the return type is **int**, you should specify it explicitly to avoid confusion.

We break the discussion of function definitions into two parts—the function's preamble (everything before the left brace) and the function's body (from the left brace to the right brace).

5.1.1 Function Preamble

Domain C supports two forms for a function preamble—the old form specified by the K&R standard and the new form (called prototyping) specfied by the ANSI standard and used in the C++ programming language. This section describes the K&R method; later sections describe the new prototyping feature.

The function's preamble must at the very least consist of the name of the function followed by a pair of parentheses. All other parts of a function are optional. The other parts are:

- The **static** storage class specifier to give the function file scope.
- The data type of the value that the function intends to return. If you do not specify a data type, the compiler assumes that the function returns an **int**.
- The function's argument list, which is a list of identifiers separated by commas.
- One optional parameter declaration for every argument in the argument list. A parameter declaration takes the same format as a variable declaration. If you omit a type in the declaration, the type defaults to **int**. If there are no arguments in the argument list, do not specify any parameter declarations.

> **NOTE:** You must put a semicolon after each parameter declaration, but never put a semicolon after the argument list.

If the function does not return an **int**, you *must* specify the true return type. If the function does not return any value, you should specify a return type of **void**. Before **void** became a common feature of C compilers, it was a convention to leave off the return type when there was no return value. The return type would default to **int**, but the context in which the function was used would usually make it clear that no meaningful value was returned. With modern C compilers such as Domain C, however, there is no excuse for omitting the return type.

5.1.1.1 Argument Declarations

Formal argument declarations obey the same rules as other variable declarations, with the following exceptions:

- The only legal storage class specifier is **register**. (The default duration is automatic, but the **auto** specifier is not legal in this context.)
- **chars** and **shorts** are passed as **ints**; **floats** are passed as **doubles**. (With the new ANSI prototyping feature, you can disable these automatic conversions.)
- A formal argument declared as an array is converted to a pointer to an object of the array type.
- A formal argument declared as a function is converted to a pointer to a function.
- You may not include an initializer in an argument declaration.

It is legal to omit an argument declaration, in which case the argument type defaults to **int**. This is considered very poor style, however.

Let us now examine several sample function preambles:

Example 1

Our first example shows the preamble of a function named **ghost** that accepts no arguments and returns no values; therefore, it simply looks like this:

```
void ghost()
```

The data type **void** ensures that no value will be passed back to the calling function. Notice that we have to put an empty set of parentheses after the name of the function to remind the compiler that this is indeed a function. The fact that the parentheses are empty means that the function has no parameters.

Example 2

Our second example is a function named **analyze** that accepts a single floating-point number as an argument. Here's how to declare it:

```
void analyze( x )
  float x;
```

Notice that we declared *x*'s data type immediately after the function definition. Also notice that we put a semicolon after the parameter declaration but *not* after the argument list.

Example 3

Our third example shows a function that accepts two integer arguments and returns a floating-point result. It looks as follows:

```
float   pythagorean( leg1, leg2 )
     int  leg1;
     int  leg2;
```

The keyword **float** identifies the data type of the returned answer. We declared **leg1** and **leg2** separately for clarity, though we could have written the function preamble like this instead:

```
float   pythagorean(leg1, leg2)
    int  leg1, leg2;
```

Example 4

Our fourth example accepts three arguments and returns a pointer to a character:

```
char *razzmatazz( high, low, precision )
     long int    high;
     short int   low;
     double      precision;
```

5.1.2 The Body of the Function

After the function preamble comes the body of the function. The body of the function takes the following format:

```
{
   local_declaration1
   .
   .
   .
   local_declarationN

   statement1
   .
   .
   .
   statementN
}
```

For example, here is a sample function body:

```
{
    int y, x;                          /* local declarations */

    scanf("%d", &x);                      /* statement */
    y = 10 * x;                           /* statement */
    printf("10 times %d is %d\n", x, y); /* statement */
}
```

You must enclose the function body in braces. Note that a function body can consist of braces and nothing else; for example, the following function body is perfectly legal:

```
{     /* a good place holder for code not yet written */

}
```

Statements within the function body can use the following kinds of variables:

- The function's parameters (that is, the parameters defined in this function's preamble).
- The variables declared within this function (variables having block scope within this function).
- Variables with global scope or file scope.

(See Chapter 2 for a complete discussion of variable scope.)

5.2 Function Allusions

A function allusion is a declaration of a function that is defined elsewhere, usually in a different source file. The main purpose of the function allusion is to tell the compiler what type of value the function returns. With the new prototyping feature, it is also possible to declare the number and types of arguments that the function takes. This feature is discussed in Section 5.4. The remainder of this section describes the old function allusion format. Note that Domain C supports both the old and the new formats.

By default, all functions are assumed to return an **int**. You are only strictly required, therefore, to include function allusions for functions that do not return an **int**. However, it is good style to include function allusions for all functions that you call.

The syntax for a function allusion is shown in Figure 5-1. If you omit the storage class, it defaults to **extern**, signifying that the function definition may appear in the same source file or in another source module. The only other legal storage class is **static**, which indicates that the function is defined in the same source file. The data type in the function allusion should agree with the return type specified in the definition. If you omit the type, it defaults to **int**. Note that if you omit *both* the storage class and the data type, the expression is a function *call* with no arguments if it appears within a block; if it appears outside of a block, it is an allusion:

```
f1();  /* Function allusion -- default type is int */

main()
{
   .
   .
  f2(); /* Function call */
   .
   .
```

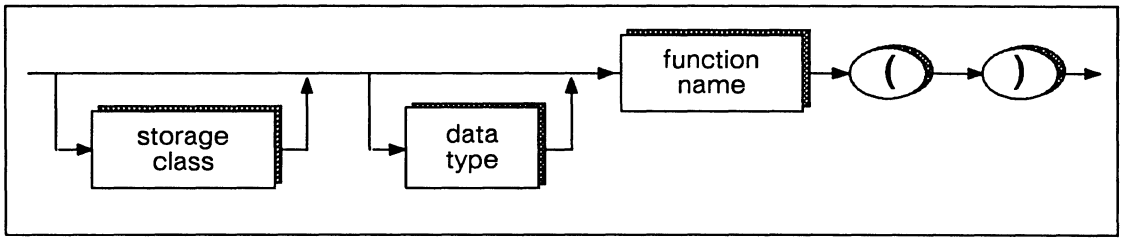


Figure 5–1. Syntax of a Function Allusion

Typically, a function allusion appears at the head of a block with other declarations. The scoping rules for function allusions are the same as for other variables declared with **extern**.

Note, however, that the default storage class rules are different for functions than for other variables. For example, in the following declaration, the storage class of **pflt** and **arr_flt[]** defaults to **auto**, whereas the storage class of **func_flt()** defaults to **extern**.

```
{
  float func_flt();
  float *pflt, arr_flt[10];
                    .
                    .
```

If this declaration appeared outside of a block, **pflt** and **arr_flt[]** would be global definitions, whereas **func_flt()** would still be a function allusion.

5.2.1 Forward References and Backward References

When we make a **forward reference** to a function, we mean that the function call appears in the source code prior to the function's definition or allusion. A **backward reference** to a function means that the function call appears in the source code after the function's definition or allusion. C unconditionally permits backward references, but restricts forward references. You *can* make a forward reference when either of the following conditions is true:

- The called function returns an **int** value
- The caller does not use the value returned by the called function

Stylistically, however, it is best to declare prototypes for all functions before they are invoked.

5.3 Function Calls

A **function call**, also called a **function invocation**, passes control to the specified function. The syntax for a function call is shown in Figure 5–2. A function call is an expression, and can appear anywhere an expression can appear. Unless they are declared as returning **void**, functions always return a value that is substituted for the function call. For example, if **f()** returns 1, the statement

```
a = f()/3;
```

is equivalent to:

```
a = 1/3;
```

It is also possible to call a function without using the return value. The statement

```
f();
```

calls the function **f()**, but does not use the return value. If **f()** returns 1, the statement is equivalent to:

```
1;
```

which is a legal C statement, although it is a **no-op** (no operation is performed, assuming **f()** has no side effects).

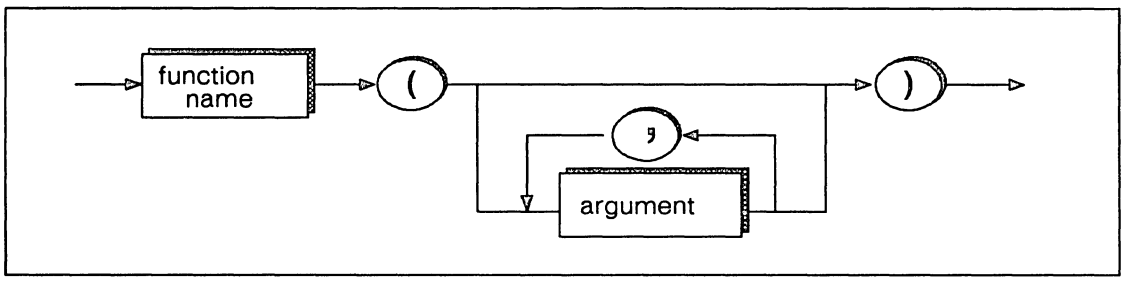


Figure 5–2. Syntax of a Function Call

5.3.1 Call by Value

Arguments to a function are a means of passing data to the function. Many programming languages (notably FORTRAN) pass arguments **by reference**, which means they pass a pointer to the argument. As a result, the called function can actually change the value of the argument. In C, arguments are passed **by value**, which means that a copy of the argument is passed to the function. The function can change the value of this *copy*, but cannot change the value of the argument in the calling routine. (Domain C supports a C++ extension that enables you to pass arguments by reference. This feature is described in Section 5.3.2.)

Figure 5-3 shows the difference. Note that the arrows in the pass-by-reference picture point in both directions indicating that the calling and called function can send information to each other through arguments. In the pass-by-value diagram, the arrows go in only one direction because only the calling function can send information through arguments. The argument that is passed is often called an **actual argument**, while the received copy is called a **formal argument** or **formal parameter**.

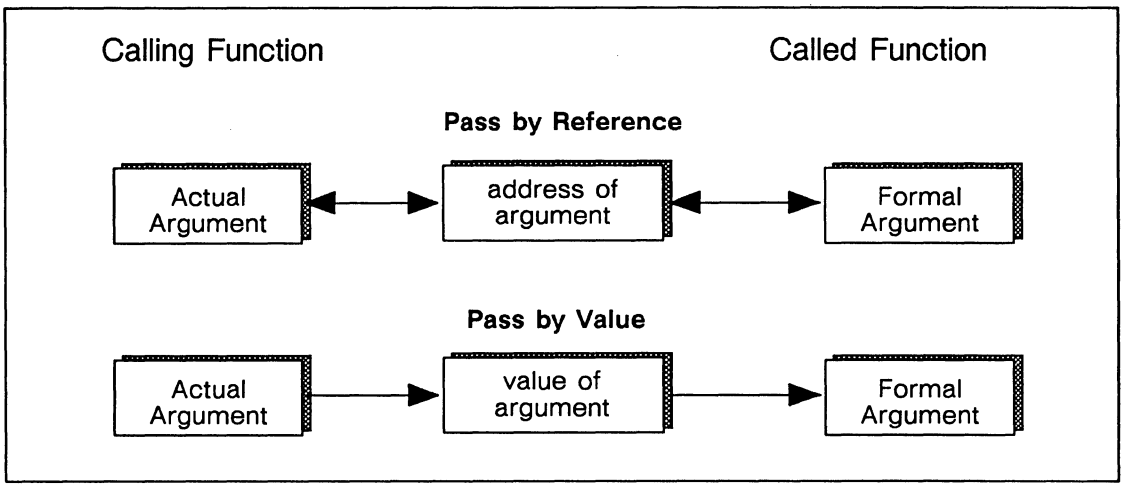


Figure 5-3. Pass by Reference vs. Pass by Value

Because C passes arguments by value, a function can assign values to the *formal* arguments without affecting the *actual* arguments. For example:

```
/* Program name is "pass_by_val_example". */
#include <stdio.h>

int main( void )
{
  extern void f( in );
  int a = 2;

  f (a );  /* pass c copy of "a" to "f()" */
  printf( "Value of \"a\" after return is %d\n", a );
}

void f( int received_arg )
{
  received_arg = 3;  /* Assign 3 to argument copy */
}
```

In the example above, the **printf()** function prints 2, not 3, because the formal argument, **received_arg** in **f()**, is just a copy of the actual argument **a**. The order of the actual arguments matches the order of the formal arguments, regardless of the names used. That is, the first actual argument is matched to the first formal argument, the second actual argu-

ment to the second formal argument, and so on. For correct results, the types of the corresponding actual and formal arguments should be the same.

If you do want a function to change the value of an object, you must pass a pointer to the object, and then make an assignment through the dereferenced pointer. The following, for example, is a function that swaps the values of two integer variables.

```
/* Program name is "pass_by_ref_example". */
#include <stdio.h>

void swap( int *x, int *y )
{
  register int temp;

  temp = *x;
  *x = *y;
  *y = temp;
}
```

To call this function, you need to pass two addresses:

```
int main( void )
{
  int a = 2, b = 3;

  swap( &a, &b );
  printf( "a = %d\t b = %d \n", a, b );
}
```

Executing this program yields:

```
a = 3     b = 2
```

5.3.1.1 Automatic Argument Conversions

In the absence of prototyping, all scalar arguments smaller than an **int** are converted to **int**, and all **float** arguments are converted to **double**. If the formal argument is declared as a **char** or **short**, the receiving function assumes that it is getting an **int**, so the receiving side converts the **int** to the declared type. If the formal argument is declared as a **float**, the receiving function assumes that it is getting a **double**, so it converts the received argument to **float**. This means that every time a **char**, **short**, or **float** is passed, at least one conversion takes place on the sending side where the argument is converted to **int** or **double**. In addition, the argument may also be converted again on the receiving side if the formal argument is declared as a **char**, **short**, or **float**.

Consider the following:

```
{
  char a;
  short b;
  float c;

  foo( a, b, c );    /*  a and b are promoted to ints,
                      *  and c is promoted to double.
                      */
        .
        .
foo( x, y, z )
char x;         /*  Received arg is converted from int
                    to  char. */
short y;        /*  Received arg is converted from int
                    to short. */
float z;        /*  Received arg is converted from
                    double to float */
{
        .
        .
```

Note that these conversions are invisible. So long as the types of the actual arguments match the types of the formal arguments, the arguments will be passed correctly. However, as discussed in Section 5.4, these conversions can affect the efficiency of your program. Prototyping enables you to turn off automatic argument conversions.

5.3.1.2 Passing an Array as an Argument

C does not pass arrays by value because this would involve too much value copying at run time (particularly for a large array). Instead, C passes the address of the first element of the array. For more information about passing arrays as arguments, see the "array operations" section of Chapter 4.

5.3.1.3 Passing Structures and Unions as Arguments

The K&R standard permits you to pass a *member* of a structure or union as a function argument. In conformance with the new ANSI standard, Domain C also further permits you to pass an *entire* structure or union as a function argument. For more information about passing structures and unions as arguments, see the "structure and union operations" section of Chapter 4.

5.3.2 Passing Arguments By Reference

Domain C supports a feature from the C++ language that allows you to declare reference variables. One way in which reference variables can be used is to pass arguments by reference. To pass arguments by reference, all you need to do is declare the formal arguments as reference variables. For example:

```
void incr( int &x )
{
  x++;
}
```

The reference variable **x** becomes an alias for whatever value is passed as an actual argument. For instance, if you call **incr()** from **main()**, as shown below, the actual argument **j** will be incremented.

```
int main( void )
{
  extern void incr( int & );
  int j = 5;

  incr( j );
  printf( "Now, the value of j is: %d\n", j );
}
```

Note that this same behavior can be obtained using pointers:

```
void incr( x )
int *x;
{
  (*x)++
}

int main( void )
{
  extern void incr( int * );
  int j;

  incr( &j );   /* pass the address of j explicitly */
  printf( "Now, the value of j is: %d\n", j );
}
```

The principal difference between these two methods is that in the pointer version, you must explicitly pass the address of **j**. In the reference variable version, the address of **j** is obtained implicitly.

The actual argument to a reference variable can be an lvalue or an rvalue. If you pass a constant, however, the called function may not modify the value. Although the compiler will not report this error, the program will abort with a run-time error when it attempts to access read-only memory. For example, if you pass a constant to **incr()**

```
incr( 5 );
```

the program will issue the following run-time error when it attempts to increment the constant:

```
?(sh) "./incr.bin" - access violation (OS/fault handler)
In routine "incr".
```

These semantics differ somewhat from the semantics described in *The C++ Programming Language,* which states that the compiler treats reference arguments as if they are normal reference variables initialized with the values passed as actual arguments. This implies that if an rvalue is passed, the compiler should produce a temporary variable. This is, in fact, how Domain C works if you pass an expression rather than a constant. For example, the following invocation of **incr()** works because the compiler generates a temporary variable for the expression 2+3.

```
incr( 2 + 3 );
```

5.4 Function Prototypes

Function prototyping is a feature introduced to the C language by Bjarne Stroustrup, the designer of the C++ programming language, and adopted by the ANSI X3J11 Technical Committee. Function prototypes in Domain C behave exactly as documented in the ANSI standard.

Function prototypes allow function declarations to include data type information about arguments. This has two main benefits:

- Function prototyping enables the compiler to check that the types of the actual arguments in the function call match the types of the formal arguments specified in the function declaration.
- Function prototyping turns off automatic argument conversions. Floating types are not converted to **double** and small integers are not converted to **int**. This can significantly speed up algorithms that make intensive use of small integer or floating-point data.

The format for declaring function prototypes is the same as the old function allusion syntax except that you enter types for each argument. For example, the function allusion

```
extern void func( int, float, char * );
```

declares a function that accepts three arguments—an **int**, a **float**, and a pointer to a **char**. The argument types may optionally be followed by variable names. For example, the previous declaration could be written:

```
extern void func( int a, float b, char *pc );
```

The variable names have no meaning other than to make the type declarations easier to read and write. No storage is allocated for them, and the names do not conflict with real variables that have the same name. You may include the storage class **register** in a prototype but it is has no meaning.

Prototyping ensures that the right number of arguments are passed, and it prohibits you from passing arguments that cannot be quietly converted to the correct type. On the other hand, it does quietly convert arguments when it can. As a result, you may actually pass the wrong type of argument without receiving a compile–time error. However, you will receive a warning if the types do not match.

If you attempt to call this function with

```
func( j, x );
```

the compiler will report an error because the call contains only two arguments whereas the prototype specifies three arguments. Also, if the argument types cannot be converted to the types specified in the prototype, a compilation error occurs. The rules for converting arguments are the same as for assignments. The following, for example, should produce an error because the compiler cannot automatically convert a **float** to a pointer.

```
{
  extern void f( int * );
  float x;

  f( x );  /*  ILLEGAL -- cannot convert a float
            *  to a pointer
            */
  ...
```

If the compiler *can* quietly convert an argument to the type of its prototype, it does so. In the following example, for instance, **j** is converted to a **float** and **x** is converted to a **short** before they are passed.

```
{
  extern void f( float, short );
  double x;
  long j;
  ...
  f( j, x );    /* OK -- long is converted to float,
                 *       and double is converted to
     ...         *       short.
                 */
```

Without prototyping, this example would produce erroneous results because **f()** would treat **j** as a **float** and **x** as a **short**, even though it is receiving a **long** and a **double**.

To declare a function that takes no arguments, use the **void** type specifier:

```
extern int f( void )  /*  This function takes no
                       *  arguments.
                       */
```

5.4.1 Function Definitions

The new protyping feature also includes an alternative syntax for declaring arguments in a function **definition**. The old style, which is still supported, requires you to declare arguments after the function header. For example:

```
int foo( x, y, z )
int x;
float y;
char *z;
{
  ...
}
```

The new syntax allows you to declare the arguments within the function header:

```
int foo( int x, float y, char *z )
{
  ...
}
```

Note that the new syntax makes it easy to create prototype declarations from new-style function definitions—all you need to do is copy the function definition, optionally precede it with **extern**, and end it with a semicolon. A prototype declaration of **foo()**, for example, would be:

```
extern int foo( int x, float y, char *z );
```

Moreover, when you use the new syntax for declaring arguments in a function definition, the definition also serves as a prototype of the function for the remainder of the source

file. That is, the compiler uses the type information specified in the definition to check the types of the arguments in all invocations of the function throughout the remainder of the source file. This type-checking does not occur if you use the old syntax. For instance:

```
int foo( x, y, z )
int x;
float y;
char *z;
{
  ...
}

main()
{
  char *a;
  float b;
  ...
  foo( a, b );  /* Will NOT produce a compile-time error */
  ...
}
```

On the other hand:

```
int foo( int x, float y, char *z )
{
  ...
}

main()
{
  char *a;
  float b;
  ...
  foo( a, b );  /* Will produce a compile-time error */
  ...
}
```

5.4.2 Prototyping a Variable Number of Arguments

If a function accepts a variable number of arguments (**printf()** for example), you can use the ellipsis token "..." in the prototype declaration. For example, the prototype for **printf()** is:

```
int printf( char *format, ... );
```

This indicates that the first argument is a character string, and that there are an unspecified number of additional arguments. The ellipsis token may appear only as the last argument type in a prototype declaration. See the description of **varargs** in the *Domain Pro-*

grammer's Reference for SysV or BSD for more information about writing functions that accept a variable number of arguments.

5.4.3 Backwards Compatibility

The Domain C compiler continues to support the old syntax and semantics for function declarations and definitions. However, unless the **-ntype** switch is used (supported with **/com/cc** only), the compiler will issue an informational message whenever it encounters a function that is not prototyped. (Note, however, that the compiler reports informational messages only if you compile with **-info 1** or a larger value.) The message informs you that the compiler is using the default prototype:

```
func_name( ... );
```

The ellipsis notation represents an indeterminate number of arguments with indeterminate types.

When the compiler encounters a prototype allusion and an old-style definition for the same function, it expands the formal argument types using the old rules before checking the types against the prototype. The following example, for instance, produces a compile-time error because the expanded argument types in the definition are **int** and **double**, whereas the prototype specifies **char** and **float**.

```
main()
{
  extern void foo( char, float );
        .
        .

void foo( x, y )
char x;
float y;
        .
        .
```

Note the distinction between this example and the following example which uses the new-style definition.

```
main()
{
  extern void foo( char, float );
        .
        .

void foo( char x, float y )
        .
        .
```

In this case, no argument expansions take place so the prototype matches the definition.

5.4.4 Using Prototypes to Write More Efficient Functions

The following example shows how prototypes can be used to write more efficient functions by turning off the automatic conversion of **floats** to **doubles**. The **sum_of_squares**() function, shown below, is allowed to pass **floats** and perform **float** arithmetic, which can lead to significant savings in calculation time for large floating-point programs.

```
#include <stdio.h>

int main( void )
{
  extern float sum_of_squares( float x, float y, float z);

  printf("Enter three floating-point numbers: ");
  scanf("%f%f%f", &x, &y, &z );
  printf("The sum of the squares of x, y, and z\ is: %f",
          sum_of_squares(x, y, z);
}

float sum_of_squares( float a, float b, float c )
{
 return (a*a)+(b*b)+(c*c);
}
```

Without prototyping, all three arguments would be converted to **double** before they were passed and then converted back to **floats** on the receiving side, making the function slower.

5.5 Returning a Value Back to the Caller

Here are C's rules for returning a value from the calling function back to the caller:

1. Use the **return** statement to pass a value back to the caller.
2. A function can directly return at most one value to the calling function. However, the value can be a structure, union, or pointer, so it is possible to indirectly return more than one value.
3. If you specify the function's data type (in the function definition), then **return** passes back a value of this data type. However, if you specified the function's data type as **void**, then **return** passes back no value.
4. If you did not specify the function's data type, then **return** passes back an **int** value.

For more information about returning from a function, see the "return" section in Chapter 4.

5.5.1 Returning Values By Reference

Just as you can pass function arguments by reference, you can also return function values by reference. To do this, you need to declare the return type as a reference variable, as shown below:

```
int &foo( int x, float y)
{
  static int j;
     .
     .
  return j;  /* returns the address of j */
}
```

When you return from a function by reference, what gets returned is actually the *address* of the returned value. This can be a dangerous practice if the returned value is a constant, an expression, or an automatic variable. If the returned value is a constant, automatic variable, or temporary variable, there is no guarantee that its memory location will will remain unchanged before the calling function accesses it. Constants are stored in read–only memory, which the compiler can use for other purposes as soon as the constant has been referenced. Automatic variables live on the stack and can be overwritten as soon as their defining blocks are exited. There is no guarantee, therefore, that the address returned by a function that returns by reference will point to meaningful data at a later point in the program. For *expressions* returned by reference, the compiler creates a temporary variable to store the expression value. For this reason, you should return only fixed duration variables by reference.

5.5.2 The #options Specifier — Domain Extension

The **#options** specifier gives you some control over the use of registers within function calls. The syntax is:

function_declaration **#options(** *option* [,*option*...] **)**

where *function_declaration* is an old-style declaration or a function prototype, and *option* is one of the following:

a0_return — Forces the function to place the return value in A0, in addition to D0. You must specify this option for Pascal routines that return pointers. See Chapter 6 for more information about cross-language communication.

abnormal — Warns the compiler that the function can produce an abnormal transfer of control. The compiler takes this warning into account when optimizing any routines that invoke this function. The **abnormal** option, however, does not affect the function to which it is applies (unless it calls itself recursively). The abnormal option is particularly useful for writing cleanup handlers.

noreturn — Indicates that the program terminates after invocation. The optimizer may remove any code following a call to a **noreturn** function since it is unreachable.

nosave — Indicates that the function will not save the contents of any registers. The **nosave** option should only be specified when declaring an assembly language program that does not follow the normal conventions for preserving registers. Routines written in C or other Domain high-level programming languages always preserve these registers. Note that assembly-language routines must preserve registers A5 and A6, which contain pointers to the current stack area and stack frame, respectively.

5.6 Recursive Functions

The C language supports **recursive functions**, which are functions that call themselves. The following example demonstrates a recursive method for calculating factorials:

```
/* Program name is "recursive_example". */
#include <stdio.h>

int   factorial( int n )
{
  int   result;

  if (n == 0)
    result = 1;
  else
    result = n * factorial(n - 1);

  return result;
}

int main( void )
{
  int  a_positive_integer, answer;

  printf( "This program finds a factorial.\n" );
  printf( "Enter an integer from 0 to 16 -- " );
  scanf( "%d", &a_positive_integer );
  answer = factorial( a_positive_integer );
  printf( "The factorial of %d is %d.\n", a_positive_integer,
          answer );
}
```

5.7 Pointers to Functions

Pointers to functions are a powerful tool because they provide an elegant way to call different functions based on the input data. Before discussing pointers to functions, however, we need to describe more explicitly how the compiler interprets function declarations and invocations.

The syntax for declaring and invoking functions is very similar to the syntax for declaring and referencing arrays. In the declaration,

```
int ar[5];
```

the symbol **ar** is a pointer to the initial element of the array.

When the symbol is followed by a subscript enclosed in brackets, the pointer is indexed and then dereferenced. An analogous process occurs with functions. In the declaration,

```
extern int f();
```

the symbol **f** by itself is a pointer to a function. When a function is followed by a list of arguments enclosed in parentheses, the pointer is dereferenced (which is another way of saying the function is called). Note, however, that just as **ar** in,

```
int ar[5];
```

is a *constant* pointer, so too, **f** in,

```
extern int f();
```

is a *constant* pointer. Hence, it is illegal to assign a value to **f**. To declare a *variable* pointer to a function, you must precede the pointer name with an asterisk. For example,

```
int (*pf)();  /* pf is a pointer to a function
               * returning an int.
               */
```

declares a pointer variable that is capable of holding a pointer to a function that returns an **int**. The parentheses around ***pf** are necessary for correct grouping. Without them, the declaration,

```
int *pf()
```

would make **pf** a function returning a pointer to an **int**.

5.7.1 Assigning a Value to a Function Pointer

To obtain a pointer to a function, you merely enter a function name, without the argument list enclosed in parentheses. For example:

```
{
  extern int f1();
  int (*pf)();

  pf = f1;  /* assign pointer to f1 to variable pf */
       .
       .
```

If you include the parentheses, then it is a function call. For example, if you write

```
pf = f1();  /* ILLEGAL -- f1 returns an int,
             * but pf is a pointer */
```

you will get a compiler error because you are attempting to assign the returned value of **f1()** (an **int**) to a pointer variable, which is illegal. If you write

```
pf = &f1();  /* ILLEGAL -- cannot take the address
              * of a function result. */
```

the compiler will attempt to assign the address of the returned value. This, too, is illegal. Lastly, you could write:

```
pf = &f1;  /* ILLEGAL -- &f1 is a pointer to
            * a pointer, but pf is a pointer to
            * an int.
            */
```

On older C compilers, this would also cause a compile error (or warning) because the compiler would interpret **f1** as an address of a function, and the **address-of (&)** operator attempts to take the address of an address. C does not permit this. Even if it did, the result would be a pointer to a pointer to a function which is incompatible with a simple pointer to a function. Domain C allows this syntax by ignoring the & operator, but the compiler does issue a warning message.

5.7.2 Return Type Agreement

The other important point to remember about assigning values to function pointers is that the return types *must* agree. If you declare a pointer to a function that returns an **int**, you must assign the address of a function that returns an **int**, not the address of a function that returns a **char**, a **float**, or some other type. If the types don't agree, you will receive a compile-time error. The following example shows some legal and illegal function pointer assignments.

```
extern int if1(), if2(), (*pif)();
extern float ff1(), (*pff)();
extern char cf1(), (*pcf)();

main()
{
  pif = if1;   /* Legal -- types match */
  pif = cf1;   /* ILLEGAL -- type mismatch */
  pff = if2;   /* ILLEGAL -- type mismatch */
  pcf = cf1;   /* Legal -- types match */
  if1 = if2;   /* ILLEGAL -- Assignment to a constant
                */
}
```

5.7.3 Calling a Function Using Pointers

To dereference a function pointer, thereby calling a function, you use the same syntax you use to declare the function pointer, except this time you include parentheses, and possibly arguments. For example:

```
{
  extern int f1();
  int (*pf)();
  int answer;

  pf = f1;
  answer = (*pf)(a); /*  Calls the function f1() with
                     *  argument a
                     */
       .
       .
```

As with the declaration, the parentheses around ***pf** in the function call are essential to override default precedence rules. Without them, **pf** would be a function returning a pointer to an **int**, rather than a pointer to a function. Note that the value of a dereferenced function pointer is whatever it was declared to be. In our case, we declared **pf** with the statement,

```
int (*pf)();
```

signifying that when it is dereferenced, it will evaluate to an **int**.

One peculiarity about dereferencing pointers to functions is that it does not matter how many asterisks you include: For example,

```
(*pf)(a)
```

is the same as:

```
(****pf)(a)
```

This odd behavior stems from two rules: first, that a function name by itself is converted to a pointer to the function; and second, that parentheses change the order of evaluation. The parentheses cause the expression,

```
****pf
```

to be evaluated before the argument list. Each time **pf** is dereferenced, it is converted back to a pointer because the argument list is still not present. Only after the compiler has exhausted all of the indirection operators does it move on to the argument list. The presence of the argument list makes the expression a function call.

It follows from this logic that you can dereference a pointer to a function *without* the indirection operator. That is,

```
pf(a)
```

should be the same as:

```
(*pf)(a)
```

This is, in fact, the case according to the ANSI standard. Older compilers, however, may not support this syntax. We recommend the second version because it is more portable, and reminds us that **pf** is a pointer variable.

5.7.4 Passing a Pointer to a Function as an Argument

You will sometimes want to pass a function pointer as an argument to another function. In this manner, you can call a function that can in turn call another function.

We demonstrate this technique in the program that follows. Consider the **main** function of this program. Notice that we assign the address of either function **max** or function **min()** to variable **pointer_to_a_function**. Therefore, when we call function **initial_checking()**, we pass the address of one of these functions.

Function **initial_checking()** copies the address of either **max()** or **min()**. Then it does some checking regardless of whether **max** or **min** was passed. Finally, **initial_checking()** calls either **max()** or **min()** by dereferencing variable **pf**.

```
/*  Program name is "pointers_to_functions".  This program
 *  shows how to pass a function pointer as an argument to
 *  another function.
 */
#include <stdio.h>

void initial_checking( int (*pf)(), int int1, int int2)
{
  int answer;

  if ((int1 <= 0) || (int2 <= 0))
  {
    printf( "You entered an illegal value.\n" );
    exit();
  }
  else
  {
    answer = (*pf)(int1, int2);
    printf( "\nThe result is %d\n", answer );
  }
```

```
}

/* find the maximum of two integers */
int max( int arg1, int arg2 )
{
   if (arg1 > arg2)
      return arg1;
   else
      return arg2;
}

/* find the minimum of two integers */
int min( int arg1, int arg2 )
{
  if (arg1 < arg2)
    return arg1;
  else
    return arg2;
}

int main( void )
{
  int  (*ptr_to_a_function)(), value1, value2, reply;

  printf( "Enter two positive integers -- " );
  scanf( "%d%d", &value1, &value2 );

  printf( "\nEnter 0 to find the max of the two integers,\n" );
  printf( "Enter 1 to find the min of the two integers. -- " );
  scanf( "%d", &reply );
  if (reply)
    ptr_to_a_function = &min;
  else
    ptr_to_a_function = &max;
  initial_checking( ptr_to_a_function, value1, value2 );
}
```

5.8 The main() Function

All C programs must contain a function called **main()**, which is always the first function executed in a C program. When **main()** returns, the program is done. The compiler treats the **main()** function like any other function, except that at run time, the host environment is responsible for providing two arguments. The first, usually called **argc** by convention, is an **int** that represents the number of arguments that are present on the command line when the program is invoked; the second, called **argv** by convention, is an array of pointers to the command line arguments.

The following program uses **argc** and **argv[]** to print out the list of arguments supplied to it when it is invoked:

```
/* Program name is "echo".  It prints the command line
 * arguments on stdin.
 */
#include <stdio.h>

int main( int argc, char *argv[] )
{
  while (--argc > 0)
    printf( "%s ", *++argv );
  printf( "\n" );
}
```

In UNIX systems, there is a program like this called **echo**. So, if you write at the command line,

```
echo Alan Turing was a father of computing.
```

the system prints:

```
Alan Turing was a father of computing.
```

Note that a pointer to the command itself is stored in **argv[0]**. This is why we use the prefix increment operator rather than the postfix operator to increment **argv**. Otherwise, the name of the command, **echo**, would be printed first.

When you invoke a program, each command line argument must be separated by one or more spaces. Note that the command line arguments are always passed to **main()** as character strings. If the arguments are intended to represent numeric data, you must explicitly convert them. Fortunately, there are several functions in the run-time library that convert a string into its numeric value. The function **atoi()**, for example, converts a string into an **int**, and **atof()** converts a string into a **float**. The following program takes two arguments, and returns the first to the power of the second:

```
#include <math.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  float x, y;

  if (argc < 3)
  {
    printf( "Usage: power <number>\n" );
    printf( "Yields arg1 to arg2 power\n" );
    return;
  }
  x = atof( *++argv );
  y = atof( *++argv );
  printf( "%f\n", pow( x, y ) );
}
```

The **pow()** function is part of the run-time library.

Chapter 6

C Program Development

This chapter describes how to produce an executable object file (that is, a finished program) from Domain C source code. There are three Domain/OS environments in which you can develop programs: Aegis, SysV, and BSD. Where the development process differs depending on the environment, we describe each environment separately.

6.1 Program Development in a Domain/OS Environment

Briefly, you create an executable object file in the following steps:

1. Compile each file of source code that makes up the program. The compiler creates one object file for each file of source code.

2. Debug program if it contains errors.

3. Link (bind) the object files if necessary. Linking is necessary if your program consists of more than one object file. The linker resolves external references; that is, it connects the different object files so that they can communicate with one another. Before linking, you may wish to package related object files into a library file with the UNIX archiver utility.

Figure 6–1 illustrates the general program development process. As described in later sections, the details differ somewhat depending on whether you are developing programs in an Aegis or UNIX environment.

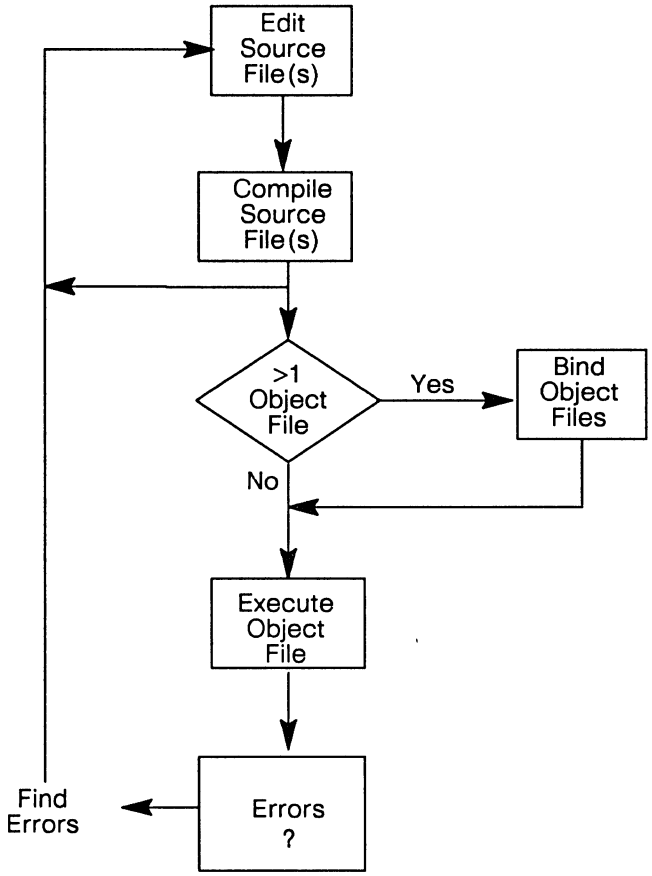


Figure 6-1. Program Development in a Domain/OS System

This chapter details the compiler and provides brief overviews of the binder (linker), archiver, and debugger utilities.

In addition to the traditional programming development scheme shown in Figure 6-1, you can also use the Domain Software Engineering Environment (DSEE™) system to develop C programs. This chapter also contains a brief description of Domain/Dialogue™, which is a product that simplifies the writing of user interfaces.

6.2 Compiling

There are two **cc** commands: one resides in **/com/cc** and the other resides in **/bin/cc**. Ultimately, both commands invoke the same compiler, which we refer to as Domain C. The syntaxes of the two **cc** commands, however, are somewhat different. The **/bin/cc** command is the traditional UNIX command for compiling C source code. When you type **cc** in a UNIX shell, the system invokes **/bin/cc**.

The **/com/cc** command is the traditional Aegis command for compiling C source code.

The behavior of each of these commands is described in the following sections.

6.2.1 Compiling with /bin/cc

To invoke the Domain C compiler with the **/bin/cc** command, type **cc** in a UNIX shell, or **/bin/cc** in an Aegis shell. The **/bin/cc** command has the following format:

$ **/bin/cc** [*option1...optionN*] *pthnm1* [... *pthnmN*]

where *pthnm* is a pathname and *option* is a command line option for **cc**, **cpp**, or **ld**.

The **/bin/cc** command is actually a driver for other commands. If the command line contains files with **.c** suffixes (source files), **/bin/cc** begins by invoking the UNIX preprocessor (**cpp**). It passes along any options that are supported by the **cpp** utility.

After **cpp** has finished processing the source files, **/bin/cc** sends them to the Domain C compiler. This is the same compiler that is invoked by **/com/cc**. However, **/bin/cc** implicitly passes along the **-bss** option so that the compiler will produce **.bss** sections rather than overlay sections (see the description of **-bss** in section 6.3.5). The other principal difference between **/bin/cc** and **/com/cc** is that **/bin/cc** compiles without optimizations by default, whereas **/com/cc** compiles with optimization level 3 by default.

Finally, the **/bin/cc** command is capable of invoking the link editor to bind object modules. It will do so automatically unless you specify the **-c** option. (The **-c** option suppresses the linking stage and saves all object modules in files with a **.o** suffix.) The link editor links together all the object modules, including those just produced by **/bin/cc**, and creates an executable program named **a.out**. Note, however, that **a.out** is created only if all global symbols are resolved. After creating **a.out**, the **/bin/cc** command deletes all of the **.o** object files produced by the compilation. By specifying options to the **/bin/cc** command, you can prevent deletion of these files. You can also have the executable binary written to a file other than **a.out**.

6.2.1.1 Some Compilation Examples

Let us consider a few compilation examples.

Example 1

Consider a C program consisting of only one file called **complete_program.c**. If we compile as follows from a UNIX shell

```
$ cc complete_program.c
```

the compiler produces an executable object file named **a.out**. We can use the **-o** option to produce an executable with a different name:

```
$ cc -o comp_program complete_program.c
```

Alternatively, we could use the **-c** option to suppress the linking stage:

```
$ cc -c complete_program.c
```

In this case, compilation would produce an object file named **complete_program.o**.

Example 2

Now consider a program broken into two files of source code—**m.c** and **f.c**. Assume that **m.c** contains the **main()** function. Further assume that somewhere in **m.c**, a call is made to a function stored in file **f.c**. Probably the easiest way to create an executable object file is to compile like this:

```
$ cc m.c f.c
```

Assuming no errors, the preceding command creates three files—two object files (**m.o** and **f.o**) and an executable file named **a.out**.

Example 3

Suppose that we discovered a mistake in file **m.c** from Example 2. After changing the source code in **m.c**, we can recompile with the following command:

```
$ cc m.c f.o
```

The preceding command creates a new **m.o** and a new **a.out**, but it does not affect **f.o**.

Example 4

Let us now use the same source files as in Example 2, except that this time, we will compile with the **-c** option as follows:

```
$ cc -c m.c f.c
```

As before, this compiler compiles both **m.c** and **f.c** to produce **m.o** and **f.o**. This time, though, the **-c** option suppresses the linking of **m.o** and **f.o**. Later, you can optionally link **m.o** and **f.o** with the **cc** command:

```
$ cc m.o f.o
```

6.2.1.2 /bin/cc Compiler Errors

The compiler *does not* produce a **.o** file if there is an error in the source code or if compilation ends prematurely for some other reason (you type a CTRL/Q, for example). In addition, if you compile and one or more of the files contain errors, then the linking phase will be suppressed, and consequently, no **a.out** file will be produced.

For instance, consider the following compilation:

```
$ cc t1.c t2.c t3.c
```

If all three source files are error-free, then the compiler creates the following four files:

```
t1.o
t2.o
t3.o
a.out
```

However, if **t2.c** had an error, then the compiler would have created the following two files only:

```
t1.o
t3.o
```

Unlike many UNIX systems, the Domain **/bin/cc** command renames any existing **.o** files to **.o.bak** before compiling a **.c** file. For example, suppose that you compile file **test.c** to produce file **test.o**. Before beginning the compilation, the system will rename any existing **test.o** file to **test.o.bak**. If compilation succeeds, you will have two files—**test.o** and **test.o.bak**. If compilation fails, you will still retain the previous object module, but it will be renamed to **test.o.bak**.

If errors occur during compilation, the compiler writes diagnostic messages in **stderr** and flags the incorrect statements in the listing file. See Chapter 9 for a complete list of compiler error and warning messages.

6.2.1.3 Overview of /bin/cc Options

The **/bin/cc** command is really an interface to the preprocessor (cpp), the Domain C compiler, and the link editor (**ld**). Not all standard UNIX options are available. Furthermore, some unique options are provided by the Domain C compiler. The compiler will interpret any command line argument beginning with a dash (-) as a compiler option. If the compiler doesn't recognize an option, it will assume that it is an option for the link editor (**ld**) and will pass it along. The options it recognizes as preprocessor options are: **-C, -D, -H, -I,** and **-U**. The link editor options are: **-a, -l, -m, -o, -r, -s, -t, -u, -x, -z, -L, -M,** and **-V**. See the *Domain/OS Programming Environment Reference* manual for more information about **ld**.

The supported options are described in Table 6-1. Some of the Domain-specific options are described in more detail in section 6.3. Note that unlike options to the **/com/cc** command, these options must be entered in the correct case. Also, you can enter multiple options with a single dash (-). For example:

```
$ /bin/cc -almc test.c
```

Table 6-1. /bin/cc Command Options

Option	Default	Description
-a	**-a**	(**ld** option) Produces an object file for execution. This is the default. Use **-r** to retain relocation information in the object module. If you specify both **-a** and **-r**, the link editor will retain relocation information for all data except common symbols, which will be allocated.
-B *name*		Assigns a prefix pathname to **cc** and **ld** for substitute compiler and linker passes. If *name* is not specified, the default is **/usr/lib/o**.
-c		Suppresss the linking phase of the compilation and force an object file to be produced, even if only one program is compiled.
-C		(**cpp** option) Prevents the preprocessor from stripping comments.

(Continued)

Table 6-1. /bin/cc Command Options (Cont.)

Option	Default	Description
-D *name* [*=def*]		(**cpp** option) Defines *name* to the preprocessor, as if by **#define**. If no definition is given, defines *name* as 1. This is the same as the Aegis **-def** option described in Section 6.3.10. The **-D** option has lower precedence than **-U**; if both are used for the same name, the name will be undefined, regardless of the order in which the options appear.
-E		Runs only the macro preprocessor on the named C programs, and sends the result to the standard output. This is similar to the Aegis **-es** option described in Section 6.3.11. Note, however, that **-E** passes the source file through the UNIX preprocessor (**cpp**), whereas **-es** processes the source file with the Domain preprocessor that is part of the Domain compiler.
-f		Not supported.
-F		Not supported.
-g		Generates full run-time debugger information. See the Aegis **-dbs** option described in Section 6.3.9.
-H		(**cpp** option) Prints out to **stderr** the pathname of every file included during this compilation. The map lists the name of each section, its starting address, and its size.

(Continued)

Table 6–1. /bin/cc Command Options (Cont.)

Option	Default	Description
-I *dir*		(**cpp** option) Changes the search path for **#include** files with names not beginning with a slash (/) and enclosed in double quotes rather than angle brackets. Look first in the directory of the source file in which the **#include** directive occurs; then in directories named in this option; and finally, in directories on a standard list. Note that this option does not affect filenames enclosed in angle brackets. It is also similar to the Aegis **-idir** option (Section 6.3.14), though the search rules are somewhat different. See the description of **#include** in Chapter 4 for more information.
-l*x*		Searches the library named **lib***x***.a**. Libraries are searched in the order that they appear on the command line. The link editor searches for libraries in the directories specified by the environment variables **LIBDIR** and **LLIBDIR** (these generally resolve to **/lib** and **/usr/lib**). You can specify additional library directories with the **-L** option.
-L*dir*		(**ld** option) Changes the search path for libraries. By default, the compiler looks for **lib***x***.a** libraries in the directories specified by **LIBDIR** and **LLIBDIR**. This option allows you specify a different directory before searching these standard directories. This is useful if you have different versions of a library and you want to specify which one the link editor should use. Note that this option is only effective if it precedes a **-l** option.
-m		(**ld** option) Produces a map or listing of the input/output sections on standard input.

(Continued)

Table 6–1. /bin/cc Command Options (Cont.)

Option	Default	Description
-M *id*	**-Many**	Generates code for a particular class of processor. Legal values for *id* are: **any** standard M68000 code **160** DSP160 code **460** DN460 code **660** 660 code **90** DSP90 code **330** DN330 code **560** DN560 code **570** DN570 code **580** DN580 code **3000** DN3000 and DN400 code **FPX** Floating–Point Accelerator Board **PEB** Performance Enhancement Board This is the same as the **-cpu** option described in Section 6.3.8.
-o *output*	**-o a.out**	Names the final output file *output*. By default, *output* is **a.out**. If you specify a different name, the system leaves any existing **a.out** file undisturbed. This is similar to the Aegis **-b** option described in Section 6.3.4.
-O		Turns on compiler optimizations. This is the same as the Aegis **-opt** option described in Section 6.3.21. The default when you compile with **/bin/cc** is **-opt 0**.
-p		Produces code that, when executed, creates a **mon.out** file that can be used by the **prof** utility to evaluate the program's performance. This is the same as the **-prof** option described in Section 6.3.23. This is the same as the **-qp** option available in SysV environments.

(Continued)

Table 6–1. /bin/cc Command Options (Cont.)

Option	Default	Description
-P		Runs only the macro preprocessor on the named C programs, and leaves the result on corresponding files suffixed with **.i**. This is similar to the Aegis **-esf** option described in Section 6.3.11.
-pg		**(BSD only)** Produces code that, when executed, creates a **gmon.out** file for use by the **gprof** utility. This is the same as the **-qg** option available in SysV environments.
-qg		**(SysV only)** Produces code that, when executed, creates a **gmon.out** file for use by the **gprof** utility. This is the same as the **-pg** option available in BSD environments.
-qp		**(SysV only)** Produces code that, when executed, creates a **mon.out** file that can be used by the **prof** utility to evauluate the program's performance. This is the same as the **-prof** option described in Section 6.3.23. This is the same as the **-p** option available in BSD environments.
-r	**-a**	**(ld** option) Retains relocation entries in the output object module. Relocation entries must be preserved if the object file will be specified in a future **ld** or **bind** command. **-a** is the default.
-s		**(ld** option) Strips line number entries and symbol table information from the output object file. The option is equivalent to using the **strip** utility and is useful if you want to reduce the size of the object module. Note, however, that removing this information from a program makes it impossible to debug the program with a source level debugger (**dbx** or **dde**).

(Continued)

Table 6–1. /bin/cc Command Options (Cont.)

Option	Default	Description
-t		Not supported. Use the **-Y** option.
-t[*p0l*]		Finds only the preprocessor (**p**), compiler passes(**0**), or binder (**l**) in the files whose names are constructed by a **-B** option. In the absence of a **-B** option, the name is taken to be **/usr/lib/n**. The value **-t ""** is equivalent to **-t0l**. The **-Y** option performs the same function and is easier to use.
-T *systype*		Defines the target system type (*systype*) for the compiled object. *systype* may be one of **any** version independent **bsd4.1** Berkeley version 4.1BSD (obsolete) **bsd4.2** Berkeley version 4.2BSD **bsd4.3** Berkeley versions 4.3BSD **sys3** System III (obsolete) **sys5** System V **sys5.3** System V, Release 3 This is the same as the Aegis **-systype** option described in Section 6.3.26.
-u *symname*		Enters *symname* as an undefined symbol in the symbol table. This option is useful if you are using the **cc** command to load a library. The symbol table is initially empty and needs an unresolved reference to force **ld** to load the first routine.
-U *name*		(**cpp** option) Removes any initial definition of *name*.
-V		(**ld** option) Outputs a message giving information about the version of **ld** being used.

(Continued)

Table 6-1. /bin/cc Command Options (Cont.)

Option	Default	Description
-w		(**BSD only**) Suppresses warning diagnostics. This is the same as the Aegis **-nwarn** option described in section 6.3.30.
-W*c*,*arg1*,[*arg2...*]		Hands off the arguments *argi* to pass *c* where *c* is one of **p**, **0**, or **l**, indicating the preprocessor, compiler or the binder. Using **-W0** enables you to use **/com/cc** options that are not available with **/bin/cc**. For instance, to specify the **-exp** option, you could write: `$ /bin/cc -W0,-exp foo.c`
-x		(**ld** option) Does not preserve local symbols in the output symbol table; enter external and static symbols only. This saves space in the object module, but still enables the link editor to resolve global references.
-Y[**p0lSILU**], *dir*		Specifies a new pathname and directory for the locations of the tools and directories designated by the first argument. You can include only one letter or number per **-Y** option, but there is no limit to the number to **-Y** options per compilation. The valid letters and numbers, and their meanings, are as follows:

(Continued)

Table 6–1. /bin/cc Command Options (Cont.)

Option	Default	Description
		p Preprocessor **0** Compiler **l** Link editor **S** Directory containing the start–up routine **I** Default include directory searched by the preprocessor **L** First default library directory searched by the link editor (**ld**) **U** Second default library directory searched by the link editor If the location of a tool is being specified, the new pathname for the tool will be **/dir/tool**. If more than one **–Y** option is applied to any one tool or directory, the last occurrence holds.
–z		(**ld** option) Does not bind anything to address zero. This option enables the run–time system to detect null pointers.

6.2.2 Compiling with /com/cc

To compile a file of C source code using the **/com/cc** command, type **cc** from an Aegis shell or **/com/cc** from a UNIX shell. The **/com/cc** command has the following format:

$ cc *source_file* [*option*1...*optionN*]

For *source_file*, specify the pathname of the source file to be compiled. By convention, C source files usually end with a **.c** suffix, though the suffix is not required. Filenames may contain up to 256 characters, including the **.c** suffix. If the filename includes the **.c** suffix, you may omit the suffix on the command line. For example, to compile C source code stored in file **test.c**, you can enter either of the following commands:

```
$ cc test
$ cc test.c
```

Following the source filename, you can optionally enter one or more of the C compiler options listed in Table 6–2, and detailed in Sections 6.2.3 to 6.2.21. Be sure to separate each option with at least one space.

If there are no errors in the source code and the compilation proceeds normally, the C compiler creates an object file and, optionally, a listing file. By default, the compiler

gives the object file the **.bin** suffix and the listing file the **.lst** suffix. For example, in response to the command

```
$ cc plot_data -l
```

the C compiler reads the file **plot_data.c**, and produces an object file named **plot_data.bin** and a listing file named **plot_data.lst**.

The **/com/cc** command preprocesses and compiles a single source file (plus any included header files), and produces a single object file. If your program contains more than one module, you must link the object files together with the **/com/bind** or **/bin/ld** command. These two commands perform similar operations—**/com/bind** invokes the Aegis binder; **/bin/ld** invokes the UNIX link editor. You can use either one to link object modules together.

You also need to use the **/com/bind** or **/bin/ld** commands if your program accesses routines in a user-supplied library. If your program consists of a single module that does not access user-supplied library routines, you do not need to explicitly invoke a linker. For more information about the **bind** and **ld** commands, see the *Domain/OS Programming Environment Reference* manual.

6.2.3 /com/cc Compiler Errors

The compiler *does not* produce an object module if there is an error in the source code or if compilation ends prematurely for some other reason (you type a CTRL/Q, for example). Rather, it looks in the appropriate directory for a binary object module with the same name as the one it would have created, had it been successful. If such a file exists, the compiler changes its name by appending the additional suffix **.bak** (*filename*.**bin.bak**.) For example, suppose your working directory contains the following files:

```
abc.c     (the source file)
abc.bin   (the object file)
```

Now suppose you recompile **abc.c**:

```
$ cc abc
```

If the source file contains an error, the compiler does not create a new version of **abc.bin**. Instead, the compiler changes **abc.bin**'*s* name to **abc.bin.bak**. If the compilation completes successfully, the compiler creates the new *filename*.**bin** file and deletes any previous *filename*.**bin.bak** file.

If errors occur during compilation, the compiler writes diagnostic messages in **errout** and flags the incorrect statements in the listing file. See Chapter 9 for a complete list of compiler error and warning messages.

6.2.3.1 Overview of /com/cc Compiler Options

Domain C supports the compiler options summarized in Table 6–2. You cannot abbreviate option names.

The optional "n" prefix negates the effect of some options. For example, the **-b** compiler option causes the compiler to produce an object file; conversely, the **-nb** option prevents the compiler from producing an object file.

Table 6–2. C Compiler Options

Option	Default	Description
-ac	**-ac**	Produces absolute code. This is the default. Another option is **-pic**, which forces the compiler to produce position-independent code.
-alnchk **-nalnchk**	**-alnchk**	Display messages about alignment of structures Suppresses alignment messages.
-b [*pathname*] **-nb**	**-b**	Produces a binary output file. The operational pathname specifies a name for the output file. If you omit the pathname, the compiler appends the **.bin** suffix to the source file's name. **-nb** inhibits production of a binary output file.
-bss **-nbss**	**-nbss**	Put uninitialized global variables in the **.bss** section of the object file. By default, all global variables are put in separate, named sections.

(Continued)

Table 6–2. C Compiler Options (Cont.)

Option	Default	Description
-comchk **-ncomchk**	**-ncomchk**	Checks to see if comment delimiters are balanced and generates a warning if they are not.
-cond **-ncond**	**-ncond**	Compiles lines begining with the **#debug** preprocessor directive.
-cpu *id*	**-cpu any**	Specifies the cpu type on which the program will run. The *id* argument can be any of the following: **90, 160, 330, 460, 560, 570, 580, 660, 3000, fpx, peb,** and **any**. Using **any** causes the compiler to produce universal machine code that can run on any of the CPUs. (Note: This option replaces the **-peb** option supported in earlier releases.)
-db **-ndb**	**-db**	Generates minimal debugging information. When you debug a program compiled with this option, you can set breakpoints but you cannot examine variables.
-dbs	**-db**	Generates full run-time debugging information and optimizes the generated object file (implies the **-opt** option).
-dba	**-db**	Generates full run-time debugging information, but prevents optimization of the generated object file (implies **-opt 0**).
-def *name* [= *value*]		Defines a name (works like the **#define** preprocessor directive). Each compilation command supports up to 128 **-define** options.
-es		Causes the compiler to run only as a preprocessor. Writes the expanded source code to **stdout**.

(Continued)

Table 6–2. C Compiler Options (Cont.)

Option	Default	Description
-esf [*pathname*]		Causes the compiler to run only as a preprocessor. Writes the expanded output to *pathname* or to **stdout** if *pathname* is omitted.
-exp **-nexp**	**-nexp**	Expands the code listing in the listing file to include the generated assembly-language code. This option implies the **-l** option.
-frnd		Forces the compiler to write all floating-point operands to memory so that floating-point comparisons produce correct results.
-idir *pathname*		Specifies a list of directories for the compiler to search to find **#included** filenames. Each compilation command supports up to 63 **-idir** options.
-indexl **-nindexl**	**-nindexl**	Produces a 32-bit index for all array references.
-info *level*	**-info 0**	Controls the output of informational messages. There are four possible levels: 0, 1, 2, and 3. Each higher level causes the compiler to output additional informational messages to indicate potential errors in the source file. **-info 0** suppresses informational messages.
-inlib *pathname*		Specifies one or more libraries that are not currently installed but should be installed when the program is executed. These libraries are searched at compile-time to determine whether indirect or absolute references should be generated.

(continued)

Table 6–2. C Compiler Options (Cont.)

Option	Default	Description
-l *pathname* **-nl**	**-nl**	Writes a listing file to **filename.lst** or to *pathname*.**lst** if *pathname* is specified. By default, this option is off, but is automatically turned on by the **-map** and **-exp** options.
-map **-nmap**	**-nmap**	Inserts a symbol map in the listing file. This option implies the **-l** option.
-natural		Makes natural alignment the default for this compilation.
-mgbl **-nmgbl**		Obsolete. As of SR10, this option is a no-op.
-msgs **-nmsgs**	**-msgs**	Controls output of the warning and error summary line. If **-nmsgs** is specified, the final message from the compiler is suppressed.
-opt **-nopt**	**-opt**	Causes the compiler to perform global program optimizations. The **-nopt** option suppresses optimizations.
-pic	**-ac**	Produces position-independent object code. The default is to produce absolute code.
-prof		Produces a **.mon** file that can be used by the **prof** utility to evaluate performance of the program.
-runtype *systype*	**sys5**	Causes the compiler to use the runtime semantics of the specified *systype* regardless of the current environment setting. The possible *systypes* are: **bsd4.2**, **bsd4.3**, **sys5**, **sys5.3**, and **any**.
-std **-nstd**	**-nstd**	Causes the compiler to issue warning messages when nonstandard language elements are encountered.

(Continued)

Table 6–2. C Compiler Options (Cont.)

Option	Default	Description
-systype *systype*	**-systype sys5**	Causes the compiler to stamp the object module for execution under a specific version of the UNIX system. The possible *systypes* are: **bsd4.2**, **bsd4.3**, **sys5**, **sys5.3**, and **any**.
-type **-ntype**	**-type**	Causes the compiler to recognize function prototypes and reference variables. Also defines **__STDC__** to be 1.
-uline **-nuline**	**-uline**	Causes the compiler to recognize **#line** preprocessor directives. **-nuline** forces the compiler to ignore **#line** directives.
-version		Causes the compiler to print its version number.
-warn **-nwarn**	**-warn**	Causes the compiler to display warning messages. The **-nwarn** option suppresses warning messages.

6.3 Domain Compiler Options

The following sections describe the **/bin/cc** and **/com/cc** options in more detail.

6.3.1 Absolute Code in User Space: -ac (/com/cc)

The **-ac** option is the default. It forces the compiler to produce absolute code, which generally executes faster than position-independent code. Unlike code produced with the **-abs** option, however, code produced with **-ac** uses indirect referencing for all global variables that are defined in global libraries. This includes global libraries currently installed as well as libraries specified with the **-inlib** option.

Refer to the *Domain/OS Programming Environment Reference* and *Domain Assembler Reference* manuals for more information about absolute and position-independent code.

6.3.2 Longword Alignment: -align and -nalign (/com/cc)

The **-align** and **-nalign** options are obsolete.

6.3.3 Displaying Messages about Alignment: -alnchk and -nalnchk (/com/cc)

When you use the **-alnchk** option, the compiler displays messages telling you whether your data is naturally aligned. Naturally aligned data increases efficiency at least slightly on any workstation, but the increase in efficiency is very significant on Series 10000 workstations.

Use the **-nalnchk** option to suppress messages about alignment. The **-alnchk** option is the default.

6.3.4 Binary Output: -b|-nb (/com/cc) -o (/bin/cc)

The **-b** option (**/com/cc**) produces a binary object module file as output. This option takes the format:

-b [*pathname*]

If you specify a pathname following **-b**, and your program compiles without errors, a binary file is created with the specified pathname and the suffix **.bin**. If you omit the pathname, the binary file is given the same name as the source file, except that **.bin** replaces **.c** as the suffix.

Specify **-nb** to suppress creation of an object module. This option is useful if you are compiling only to check for errors in your program. **-b** is the default.

The **-o** option (**/bin/cc**) allows you to direct the resulting object file to a file other than **a.out**, which is the default. This option takes the following format:

-o *pathname*

Note that you must specify a *pathname*.

6.3.5 Global Variables in .bss Section: -bss|-nbss (/com/cc)

By default (**-nbss**), the **com/cc** compiler creates a separate, named section for each global variable. The name of the section is the same as the name of the variable. In contrast, the **/bin/cc** compiler puts all initialized global variables in **.data** and all uninitialized global variables in **.bss**. The **-bss** option causes the **/com/cc** compiler to mimic the behavior of the **/bin/cc** compiler. This is useful if your program uses many global variables and you don't want a named section for each one. Even if you use the **-bss** option, you can still create named sections for global variables by using the **#attribute[section]** modifier. See Chapter 2 for more information about this modifier.

6.3.6 Comment Checking: -comchk|-ncomchk (/com/cc)

The **-comchk** option causes the compiler to check that comment pairs are balanced—that there are no extra left comment delimiters (/*) before a right comment delimiter (*/). When **-comchk** is specified, the compiler returns a warning for every additional left comment delimiter. Using **-comchk** can help you identify a place in the program where some code was not compiled because the compiler assumed that it was part of a comment. The **-ncomchk** option inhibits this extra check. **-ncomchk** is the default.

For example, consider the following program fragment:

```
/*This comment should be closed, but I forgot to do it!
  crash_flag = 10;    /* MUST occur or else disaster */
```

If we compile with **-comchk**, then the preprocessor will report the following warning:

```
******** Line 8: Warning: Unbalanced comment; another comment start
found before end.
```

If we compile with **-ncomchk** (or simply without **-comchk**), then the preprocessor will not report the warning.

> **NOTE:** Using **-comchk** only identifies a problem area in the source code; the option has absolutely no affect on the machine code generated.

6.3.7 Conditional Compilation: -cond|-ncond (/com/cc)

The **-cond** option invokes conditional compilation. When this option is on, lines marked with the **#debug** preprocessor directive are treated as source code lines and are compiled. If you compile with the **-ncond** option, the compiler treats the marked lines as comments.

-ncond is the default.

6.3.8 Target Node Selection: -cpu *cpu* (/com/cc) -M *cpu* (/bin/cc)

Use the **-cpu** or **-M** option to select the target workstations that the compiled program can run on. If you choose an appropriate target workstation, your program might run faster; however, if you choose an inappropriate target workstation, the run-time system will issue an error message telling you that the program cannot execute on this workstation. The Domain C compiler can generate code in five possible modes:

- Code that will run on a DSP160, DN460, or DN660 workstation
- Code that will run on a workstation with the M68020 microprocessor and the M68881 floating-point coprocessor
- Code that will run on a workstation with a Performance Enhancement Board (PEB)
- Code that will run on any Apollo workstation
- Code that will run on a DN5xx-T with a floating-point accelerator (FPX) unit

You select the code generation mode through the argument that you specify immediately after **-cpu** or **-M**. Table 6-3 shows the possible arguments and the code generation mode that they select.

Note that there are many possible arguments to **-cpu** and **-M**; however, many of them are synonyms. For example, **-cpu 330** produces exactly the same code as **-cpu 560**.

The advantage of compiling with **-cpu any** is that the resulting program can run on any Apollo workstation. This is how Apollo compiles the programs that appear in your **/com** or **/bin** directories. **-cpu any** and **-Many** are the defaults.

Table 6–3. Arguments to the **-cpu** *and* **-M** *Options*

Argument	What It Does
-cpu 160 -cpu 460 -cpu 660	Generates code for the DSP160, DN460, and DN660 workstations.
-cpu 90 -cpu 330 -cpu 560 -cpu 570 -cpu 580 -cpu 3000	Generates code for workstations with a M68020 processor and a M68881 floating-point unit (includes the DSP90, DN330, DN560, DN570, DN580, DN3000, and DN4000).
-cpu fpx	Generates code for workstations with a floating-point accelerator (FPX) unit (includes DN5xx-T's)
-cpu peb	Generates code for workstations with a PEB (includes the DN100, DN320, DN400, and the DN600, when equipped with an optional PEB).
-cpu any	Generates code for any workstation.

The advantage of the processor-specific code generation modes is that the compiler generates code optimized for that particular processor, which makes the programs so compiled run faster. The advantage is seen mostly in programs that make heavy use of floating-point. Programs that make heavy use of 32-bit integer multiply and divide might also show significant improvement.

NOTE: There is one caveat concerning programs compiled with the **-cpu fpx** option. The address of an instruction for a floating-point fault is not stored in the Instruction Address register (IADDR) as it is for programs compiled with the **-cpu 330** and **-cpu 3000** options. Consequently, fault handlers should not rely on this address when code is compiled with **-cpu fpx**. This warning applies only to assembly language fault handlers.

6.3.9 Debugger Output: -db|-ndb|-dbs|-dba (/com/cc) -g (/bin/cc)

The **-db**, **-dba**, **-dbs**, and **-g** options generate output for later use by **dde** and **dbx**, the language-level debuggers. These debuggers allow you to search for program errors using the program's variables, parameters, statement labels, and other program-defined symbols. The output generated by the four compiler options allows the debuggers a particular level of access to the program. The **-ndb** option specifies no debugger access. Table 6–4 summa-

rizes the access granted to the debuggers by each option. For an overview of the debuggers, see Section 6.8.

Table 6-4. DEBUG Compilation Options

Compiler Option	Debugger Access
-ndb	None.
-db	Source line numbers (except lines optimized out) and functions.
-dbs (or **-g**)	Same access as -db with the addition of local and global variables.
-dba	Same access -dbs but without any code optimization.

If you use the **-db** option, the compiler puts minimal debugger preparation information into the **.bin** file. This preparation is enough to enter the debugger and set breakpoints, but not enough to access symbols, such as variables and constants.

If you use the **-dbs** option, the compiler puts full debugger preparation information into the **.bin** file. This preparation allows you to set breakpoints and access symbols. When you use the **-dbs** option, the compiler sets the optimization level to 3. (You can override this by specifying a different optimization level with the **-opt** option.)

The **-g** option to **/bin/cc** is the same as **-dbs**.

The **-dba** option is identical to the **-dbs** option except that when you use the **-dba** option, the compiler sets the **-nopt** option (even if you specify **-opt**).

For more complete details on these four options, see the *Domain Distributed Debugging Environment Reference* manual.

6.3.10 Name Definition: **-def** *name* [= *value*] (/com/cc)

-D*name*[=*value*] (/bin/cc)

The **-def** option lets you define a name and, optionally, its value at compilation time. It takes the format:

-def *name* [= *value*]

This option has the same effect as the **#define** preprocessor directive. You may use as many as 128 **-def** options in a compile command line. If you do not use the optional =*value* component, the default value of the name is 1. For example, consider the following simple program stored in file **test.c**:

```
#include <stdio.h>
int x = 0;

int main( void )
{

#if env1
  x = 500;
#else
  #if env2
    x = 1000;
  #endif
#endif

  printf("x = %d\n", x);
}
```

Tables 6–5 and 6–6 shows the varying effects of three different compilation command lines:

Table 6–5. The Effect of -def

Compilation Command	Result
$ cc test	x = 0
$ cc test -def env1	x = 500
$ cc test -def env2	x = 1000

Table 6–6. The Effect of -D

Compilation Command	Result
$ cc test.c	x = 0
$ cc -Denv1 test.c	x = 500
$ cc -Denv2 test.c	x = 1000

If there are spaces in the value, be sure to surround the *entire* definition with quotes. For example,

```
$ /com/cc -def "rev_string=@"Revision 1.23 1-JAN-85@""
```

or

```
$ /bin/cc '-Drev_string="Revision 1.23 1-Jan-85"'
```

is the same as

```
#define rev_string "Revision 1.23 1-Jan-85"
```

Note that in an Aegis shell, any embedded quotation marks must be preceded with "@".

The **-D** option behaves just like the **-def** option. Note, however, that unless you enclose the entire option in single quotes, you cannot put a space between the defined name and the equal sign or between the equal sign and the value.

6.3.11 Preprocessor Options: -es|-esf (/com/cc) -E|-P (/bin/cc)

Compilation actually consists of two phases—preprocessing and processing. During preprocessing, the preprocessor obeys all the preprocessor directives (such as **#define, #if, #include**) in your source code. It is not until processing that the compiler actually generates executable code. By default, when you issue the **cc** command, you invoke both the preprocessor and the processor. However, by using the **-es** or **-esf** options (**-E** or **-P** option with **/bin/cc**), you invoke only the preprocessor. The output (known as the expanded source) from the preprocessor can be studied and run through the processor if desired.

The two **/com/cc** options produce the exact same expanded source file; the only difference is in the pathname of the expanded source file. The options take the following format:

-es
-esf [*pathname*]

The **-es** option directs the expanded source to standard output.

The **-esf** option takes an optional *pathname* as an argument. If you omit a *pathname*, the expanded source file gets the same name as the source file, but the **.c** suffix is replaced by the suffix **.i**. If you specify a *pathname*, the compiler uses that name and automatically appends the **.i** suffix, unless it is already present.

> **NOTE:** You cannot use **-es** or **-esf** in a command line that also contains **-l, -b,** or **-exp.**

The **/bin/cc -E** option behaves exactly like the **-es** option; the **-P** option functions exactly like the **-esf** option without a *pathname* argument.

6.3.12 Expanded Code Listing: -exp|-nexp (/com/cc) -S (/bin/cc)

If you use the **-exp** or **-S** option, the compiler produces an expanded listing file that contains a representation of the generated assembly-language code, interleaved with the source code. The listing also shows all macro expansions.

Note that using **-exp** causes the compiler to produce a listing file even if you did not use the -l option. However, if **-nl** appears on the command line after **-exp**, then the expanded code listing will be suppressed.

-nexp is the default.

6.3.13 Floating-Point Accuracy: -frnd (/com/cc only)

The **-frnd** option forces the compiler to write all floating-point operands to memory and then fetch the memory contents before evaluating the expression. This ensures that each operand will have the same amount of precision so that floating-point comparisons will produce correct results. If you do not compile with **-frnd**, floating-point operands may be kept in registers, which support more accuracy than memory. Consequently, when a register operand is compared with a memory operand, the result may not be what is expected. This is particularly true of equality comparisons. Consider the following C program:

```
double fetch( void )
{
  return 1.1;
}

int main( void )
{
  double x;

  x = fetch();

  if (x - 0.1 == 1.0)
    printf(" Pass\n" );
  else
    printf( " Fail\n" );
}
```

If you compile with **-cpu 3000**, and without **-frnd**, this program fails because the values 0.1 and 1.1 cannot be represented exactly in base 2 floating-point. Thus, the quantity **(x - 0.1)** can only be approximated. This value is calculated in an 80-bit register, and then a compare is generated to see if this value is *exactly* equal to 1.0, which is stored in memory. Since the register has more accuracy than memory, the comparison fails.

If you compile with **-frnd**, the 80-bit register is stored (and rounded) in a single-precision 32-bit temporary memory location. Now when it is compared with 1.0, which is also stored in memory, the comparison passes.

6.3.14 Include Directories: -idir (/com/cc)

The **-idir** option tells the compiler to look for include files in the directories specified by the pathname. (Include files are detailed in the "#include" listing of Chapter 4.)

This option allows you to postpone until compilation naming the directories for include files. Suppose, for example, that different versions of an include file have the same name but reside in different directories. You might enter the filename in an **#include** command in your code, and then select the appropriate directory with **-idir**. You may use as many as 63 **-idir** options in each compilation command.

When you enclose the include filename in quotes in the **#include** control line, the compiler first searches the working directory, then the directories specified by **-idir** (if any), and finally, the directory **/usr/include**. When you enclose the include filename in angle brackets (< >), the compiler searches the **-idir** directories first, and then **/usr/include**.

Suppose that your source file contains these **#include** statements:

```
#include "local_include_file"
#include <global_include_file>
```

You then compile the program with the following options while in the same directory as the local include file:

```
cc test -idir /personal -idir \impersonal
```

The C compiler resolves the include files by searching for filenames in the following order:

For **local_include_file**:

1. local_include_file
2. /personal/local_include_file
3. \impersonal/local_include_file
4. /usr/include/local_include_file

For **global_include_file**:

1. /personal/global_include_file
2. \impersonal/global_include_file
3. /usr/include/global_include_file

Note that the **-idir** directories are searched in the order in which they appear in the compilation command.

6.3.15 Array Reference Index: -indexl|-nindexl (/com/cc)

The **-indexl** option disables some optimizations and forces the compiler to use 32-bit indexing in subscript calculations for all array references. **-nindexl**, the default, causes the compiler to use the source code's array dimension information to determine whether to use 16-bit or 32-bit indexing.

6.3.16 Informational Messages: -info | -ninfo (/com/cc)

The Domain C compiler produces three types of messages:

informational Identifies aspects of the source file that will compile correctly, but could be rewritten to be more efficient or more portable.

warning Identifies aspects of the source file that may be correct, but are suspect. The compiler makes a "best guess" as to what the source means and produces an object file.

error Identifies syntactical or semantic errors that prevent the compiler from producing an object file.

By default, the compiler outputs warning and error messages but not informational messages. (You can suppress warning messages with the **-nwarn** option.)

The **-info** option causes the compiler to output informational messages. However informational messages are divided into four levels, where each higher level represents additional messages. The four levels are as follows:

0 No messages (this is the default).

1 Messages about old-style function definitions and allusions and messages indicating that members of a structure are not naturally aligned.

2 Messages indicating that the program could be written more efficiently (for example, a variable is declared but never used).

3 Reserved for future use.

4 Reserved for future use.

Note that each level includes the messages in all lower levels. For instance, if you specify **-info 3**, you will receive level 1, 2, and 3 messages, but not level 4 messages.

6.3.17 Installed Libraries: -inlib (/com/cc)

The **-inlib** option allows you to specify additional libraries that are not currently installed, but will be installed when you execute the program. The compiler needs this information to determine whether to use indirect or long absolute addressing modes. If you are producing absolute code (the default), you must use this option to specify any library that is not currently installed, but should be installed when the program is executed. If you use the **-pic** option to produce position-independent code, you do not need to specify libraries that are not yet installed. See the *Domain/OS Programming Environment Reference* and the *Domain Assembler Reference* manuals for more information about absolute and position-independent code.

The **-inlib** option has a format similar to the **-idir** option. Instead of specifying the pathnames of directories, however, you specify the pathnames of files that you want to **inlib**. The following command line, for example, tells the compiler that the object files **-libs/my_lib** and **//node/libs/master_lib** will be installed when **examp.bin** is loaded:

```
$ cc examp -ac -inlib -libs/my_lib //node/libs/master_lib
```

If you specify a library with the **-inlib** option, the compiler writes a **library record** into the object file so that the loader automatically **inlib**s the library when it loads the object file. If the library is not available at load time, an error occurs.

6.3.18 Listing File: -l|-nl (/com/cc)

The **-l** option causes the compiler to produce a listing file. The listing file contains the following information:

- The source code complete with line numbers. Note that line numbers start at 1 and increment by 1 (even if there is no code at a particular line in the source code). In addition, note that the listing file separately numbers lines from include files.
- Compilation statistics.
- An object module section summary.
- A list of the compiler options affecting code generation.
- Errors and warnings generated during the compilation.
- A count of the error and warning messages produced by compilation.

The format for the **-l** option is

-l [*pathname*]

If you specify a *pathname* following **-l**, the listing file is created with the specified pathname and the suffix **.lst**. If you omit *pathname*, the listing file is given the same name as the source file, except that **.lst** replaces **.c** as the suffix.

The **-nl** option is the default, but note that **-map** and **-exp** contain an implicit **-l**.

6.3.19 Symbol Map: -map|-nmap (/com/cc)

If you use the **-map** option, Domain C creates a map file. A map file contains everything in the listing file (produced by using **-l**) plus a special symbolic map. The special symbolic map consists of two sections.

The first section describes all the routines in the compiled file; for example, here is a sample first page:

```
001 TEST_C module(Psect = procedure$,Dsect = data$)

002 q function(Proc = 000000,Ecb = 000040,Stack Size = 16,
               Psect = my_proc,Dsect = data$)

003 main function(Proc = 000000,Ecb = 00002C,Stack Size = 4,
                  Psect = procedure$,Dsect = data$)

004 <apollo_c_startup> program(Proc = 00004C,Ecb = 000018,
               Stack Size = 0,Psect = procedure$,Dsect = data$)
```

Let us consider this information on a line-by-line basis.

The first line

```
001 TEST_C module(Psect = procedure$,Dsect = data$)
```

tells us the name of the module (**TEST_C**) and the names of the head-of-file procedure and data sections.

The second and third lines

```
002 q function(Proc = 000000,Ecb = 000040,Stack Size = 16,
               Psect = my_proc,Dsect = data$)

003 main function(Proc = 000000,Ecb = 00002C,Stack Size = 4,
                  Psect = procedure$,Dsect = data$)
```

tell us the names (**q** and **main**) of the two user-supplied functions in the source code. The map supplies five pieces of information for each function. The first piece of information is the starting address of the function measured in bytes from the beginning of the procedure section. The section piece of information is the offset in bytes of the ECB (Entry Control Block). The third piece of information is the stack size measured in bytes. The fourth and fifth pieces are the names of the procedure and data sections that the function is stored in.

The fourth line

```
004 <apollo_c_startup> program(Proc = 00004C,Ecb = 000018,
            Stack Size = 0,Psect = procedure$,Dsect = data$)
```

tells us the same information as the second and third lines, but for a special startup function provided by the compiler.

The second section of the special symbolic map contains an alphabetic listing of all the variables used in the source code. For example, here is a sample second page:

```
002 arg1    var(+000014/S): long int
002 c       var(-000006/S): char
001 g       var(+000000/g): float
003 j4      var(-000008/S): long int
002 m       var(-000004/S): long int
001 student  var(+000000/student): array[0..9] of char
001 x       var(extern): long int
001 y       var(+000000/y): long int
```

The preceding data tells us that the program referred to eight variables. Let us consider the second variable

```
002 c       var(-000006/S): char
```

in greater detail:

002 — The number to the far left tells us where within the program that the variable was declared. Top-level declarations get the number 001. A number higher than 001 indicates a variable declared in a function. For example, 002 means that this variable was declared in the first function of the file, 003 identifies a variable declared in the second function of the file, and so on.

c — The name of the variable.

(-000006/S) — This number and identifer tells us where the variable is actually stored at run time. If the identifier is "S", it means that the variable is stored on the stack. Otherwise, the identifier tells you the name of the section in which the variable is stored. The numerical part of the data is the offset (in bytes) from the beginning of the stack or the section.

char — This tells us the data type of the stored variable.

Note that variable **x** does not have an offset or section name since it is a declaration, but not a definition.

The **-nmap** option suppresses creation of the special symbol map. **-nmap** is the default.

6.3.20 Error and Warning Summary: -msgs|-nmsgs (/com/cc)

The **-msgs** and **-nmsgs** options control the output of a summary message from the compiler. If **-nmsgs** is given, the final message from the compiler (shown below) is suppressed:

```
XX errors, YY warnings, C Compiler, Rev n.nn
```

The default is **-msgs**.

6.3.21 Optimization Levels: -opt [*n*] (/com/cc)
-O [*n*] (/bin/cc)

For **/com/cc**, the **-opt 3** option is the default. For **/bin/cc**, the default is no optimization.

The **-opt** option allows you to specify the kinds of optimization performed on your source program, by means of an "optimization level." The syntax for the **-opt** option is:

-opt [*n*]

where *n* is an integer between 0 and 4 that represents the optimization level. At **-opt 0**, very few optimizations are performed. For each higher optimization level, more optimizations are performed. If you specify **-opt** and omit the optimization level, the level defaults to **-opt 3**. If you omit the **-opt** option completely, the default option, **-opt 3**, is assumed.

The obsolete option **-nopt** is equivalent to **-opt 0**. Each higher level of optimization includes all optimizations performed at the lower levels of optimization. Because the compiler does more work at each higher level of optimization, it may take longer to compile your program at higher optimization levels.

It is important to note that the **-dba** option overrides anything you specify for the **-opt** option. If you want your code to be optimized, and want to use the debugger on your program, you should use the **-dbs** option rather than **-dba**. When you specify **-dba**, the **-opt** option is set to **-opt 0**, regardless of what you specified for **-opt** on the command line for the compilation. In addition, **-dba** represents an even lower level of optimization than **-opt 0**.

NOTE: If you wish to use the Debugger (described in the Section 6.8) to debug a program compiled at **-opt 3** or **-opt 4**, you may find that you get inaccurate values for some local variables at points in the source code where those variables are not actively in use. This happens because the value of the variable is assigned to a machine register, rather than being kept in the computer's main memory. The optimizer may decide that the main memory location for this variable does not need to be updated, because all uses of the variable in the source program can legally use the value of the variable that is retained in the machine register. In addition, the optimizer may merge some source statements together, or eliminate source statements entirely, when legal to do so. When you are debugging with these optimizations, you may see what appear to be strange jumps in the control flow of the program. In addition, you may be unable to set a breakpoint at a particular source line because the generated code for that source line has been optimized away or merged with the code from another source line. It can be slightly more difficult to use the debugger with optimized code, but there is no reason to avoid using **dde** or **dbx** with the optimization levels discussed here. See the *Domain Distributed Debugging Environment Reference* manual for more details concerning the use of the debugger.

The following is a brief description of the optimizations performed at each optimization level. For a detailed discussion of compiler optimization techniques, consult a general compiler textbook.

-dba Represents the lowest possible optimization level. It forces the -opt option to be **-opt 0**, and additionally suppresses some optimizations that are normally performed at **-opt 0**. In particular, the **-dba** option forces the compiler to store machine registers in main memory after every statement. Even with the **-dba** option, however, some optimizations are still performed. For example, the compiler may:

- Rearrange expressions to minimize the number of registers needed to compute the expression.
- Generate faster short range branch instructions in place of long branches where possible.
- Compute constant expressions that appear in the source code, such as 2*3, rather than generating code to compute them.
- Compute multiple occurrences of the same expression only once.

Another example of simple constant folding performed at this level is shown by the following example:

```
unsigned char small_range;
        .
        .
if (small_range < 0)
        .
        .
```

In this example, **small_range** can never be less than zero because of its type. The compiler will therefore substitute the value **FALSE** for the expression "small_range < 0". The expression

```
if FALSE
```

means that the statements following **if** cannot be executed, so the compiler will not generate code for them.

-opt 0 Performs the optimizations described above. If **-dba** is not also set, the compiler will permit values to remain in registers across statements where it is legal to do so. Additionally, a sequence of generated code that is identical to another sequence may have all its instructions replaced by a branch to the other identical sequence of instructions.

-opt 1 performs the following optimizations:

- Eliminates limited global "common subexpression."
- Eliminates "dead code."
- Transforms integer multiplication by a constant into shift and add instructions rather than using direct multiply, where appropriate.
- Performs simple transformations for speed.
- Merges assignment statements where possible.

A **common subexpression** is an expression that appears two or more times in the program, with no intervening assignments to any component of the expression. In such cases, the expression need only be computed once, and the other occurrences of the expression can be replaced with the resulting value. **Dead code** is code that cannot be executed because there is no execution path of the program that leads to the code.

-opt 2 Performs the following optimizations:

- Substitutes constants for "reaching" definitions.
- Folds global constants.

Assigning to a variable, or using the variable as parameter in a function call, produces a **definition** of the variable. A particular definition of a variable is

said to "reach" later uses of the variable if there are no other definitions between the original definition and the use of the variable. If the definition is an assignment of a constant to the variable, uses of the variable that are "reached" by the definition can be replaced with the constant value. As constants are substituted for variable uses, the expressions in which the variable uses occurred are sometimes transformed into constant expressions that can be evaluated during compilation. This eliminates the need to generate code to compute the value of the expression. For example, in the statements

```
a = 3;
c = 2 * a;
```

there are no other definitions of the variable **a** between the original assignment and the use of **a** in the expression **2*a**. So the compiler can substitute the value 3 in the expression **2*a**. The expression then becomes **2*3**, which is computed during compilation. As a result, the program does not perform a multiply when it executes. Instead, it merely assigns the already computed value 6 to **c**.

-opt 3 This is the default optimization level. At this level, the compiler performs the following optimizations:

- Live analysis on local variables.
- Redundant assignment statement elimination.
- Global register allocation.
- Instruction reordering.
- Removal of invariant expressions from loops.
- Exhaustive searches through each routine for global common subexpressions to eliminate.

The **-opt 1** and **-opt 2** levels make only limited searches through the code for global common subexpressions.

Performing **live analysis** of local variables involves determining the areas of a routine where a variable is actively used. For example,

```
    .
    .
j = k;
if (i = 0)
{
  i = 2;
  j = 3 * j;
  printf( "%d\n", j );
}
else
{
  k =  i * 4;
  printf("%d", k);
}
    .
    .
```

In the **else** clause of the example, there are no uses of **j**. If there are no further uses of the variable **j** on any execution path from the **else** clause to the end of the program, **j** is not actively used in the **else** clause and on execution paths from the **else** to other parts of the routine. **j** is therefore considered "dead" from the statement following the **else** to the end of the routine. Within the **if** clause, there is a use of **j**. Therefore, **j** is actively used within the **if** clause, and is considered "live" within the **if** clause. If there are other uses of **j** that can be reached from the **if** clause, **j** is considered "live" along the paths that lead to those uses. Live analysis is important because it allows the compiler to allocate local variables to machine registers for exactly as long as the variable's value is needed. When the variable becomes "dead," the register can be used for other variables or expression values. In general, the CPU can reference a value in a register faster than a value in the computer's main memory. Efficient use of registers increases the execution speed of your program.

Redundant assignment elimination performed at this optimization level may result in warning messages such as the following:

```
******** Line 14: [Warning 279]  Value assigned to
SMALL_RANGE is never used; assignment is eliminated by op-
timizer.
```

Consider the following example:

```
main()
{
  int i, j;
  fscanf("%d%d", &i, &j);
  if (i == 0)
    j = 3;
  printf("%d", i);
}
```

There are no uses of the variable **j** after the assignment **j=3**. Since the value assigned to **j** is not used, the compiler can eliminate the assignment completely without changing the result computed in the program. In fact, once the assignment is eliminated, the **if** portion of the statement isn't needed either, and can be eliminated. If we change the example so that **j** is used after the assignment,

```
main()
{
  int i, j;
  fscanf("%d%d", &i, &j);
  if (i == 0)
    j = 3;
  printf("%d\t%d", i, j);
}
```

the assignment is no longer eliminated.

Global register allocation allows variables that are local to a routine to have their values placed in machine registers for faster access. In many cases, all

definitions and uses of a local variable may occur in a register, and the copy of the variable in the computer's main memory is never used or updated. Keeping variables in registers makes your program execute faster. The global register allocator treats the register variable declaration as advice, not as a directive. Variables declared as register receive special consideration for allocation to registers. However, if a variable is declared as register, but is not used, it will not be allocated to a register.

Instruction reordering changes the order in which some instructions are executed to take advantage of overlaps that are possible in some instruction sequences. Some integer instructions can execute at the same time as some floating-point instructions, as long as the integer instructions do not depend upon the result computed by the floating-point instruction.

A **loop invariant expression** is an expression whose value does not change during the execution of a loop. When invariant expressions are computed outside a loop, they are only computed once, instead of needlessly being computed on each pass through the loop. This makes the loop execute faster, and generally increases program execution speed. For example:

```
        .
        .
for (i=1; i <= 10; i++)
{
  j = k * m;
  j = i + j;
}
        .
        .
```

The expression **k * m** is invariant in the above example. The compiler can safely transform this loop as follows:

```
        .
        .
temp = k * m;
for (i=1; i <= 10; i++)
{
  j = temp;
  j = i + j;
}
        .
        .
```

After the invariant expression is removed from the loop, the example does only one multiply instead of 10 to make the assignment to **j**.

-opt 4 This is identical to **-opt 3** in the present compiler. Future releases may use this level to perform additional optimizations.

6.3.22 Position-Independent Code: -pic (/com/cc)

By default, Domain compilers produce **absolute** or **fixed position** code. Absolute code programs are loaded at a fixed (determined prior to load time) address. By default, absolute code programs are loaded at the low end of user virtual memory (hexadecimal address 8000). If the loader cannot load your program at the pre-determined address (because there is already a resident program), it reports an error.

The **-pic** option enables you to produce **position-independent code** (pic). Pic code can be loaded and run anywhere in virtual address space without relocating (modifying at load-time) the procedure text. The procedure text is mapped at load time, which is a much faster operation than copying and relocating.

In general, absolute code runs faster than position-independent code so you will not use the **-pic** option often. However, there are a few instances where you must use the -pic option. In particular, you should produce pic code for all routines that are to be entered into an installed library. In addition, you should produce pic code for the following:

- Programs that invoke other absolute code programs in-process with the **pgm_$invoke** system call in **pgm_$wait** mode.
- Programs that are dynamically loaded, such as IOS type managers, GPIO drivers, and shared libraries.

Refer to the *Domain/OS Programming Environment Reference* and *Domain Assembler Reference* manuals for more information about absolute and position-independent code.

6.3.23 Profiling: -prof (/com/cc) -p (/bin/cc)

The **-prof** and **-p** options force the compiler to produce code that, when executed, produces a **.mon** file that can later be used by the **prof** utility to identify bottlenecks in the program. For example, if you compile a program called **test.c** with the command,

```
$ /com/cc test -prof
```

the compiler will produce a file called **test.mon** in the working directory. To get performance statistics, execute the command:

```
$ prof test.bin
```

will display the number of calls to each function and the amount of time spent in each function. If you don't specify the program name on the **prof** command line, **prof** assumes **a.out**. For example:

```
$ /bin/cc -p test.c
$ prof
```

For more information about the **prof** utility, see the *SysV Command Reference* manual. Note that you can also use the **dpak** utility to obtain more detailed statistics about program performance. For details about **dpak**, refer to *Analyzing Program Performance with DPAK*.

6.3.24 Nonportable References: -std|-nstd (/com/cc)

The **-std** option causes the compiler to issue warning messages for nonportable language elements (that is, extensions to the K&R standard). If portability is an issue, pay attention to the warnings; otherwise, ignore them.

The **-nstd** option suppresses reporting of nonstandard elements. **-nstd** is the default.

6.3.25 Run-Time UNIX Version Selection: -runtype *systype* (/com/cc)

When you execute a C program, the run-time environment uses the semantics of the systype stamped on the object module. By default, this is **sys5**, but you can change it with the **#systype** preprocessor directive or the **-systype** compile option. Use the **-runtype** option to override the systype that is stamped on the object module. That is, you can use **-runtype** when you compile with one systype setting but want to execute the program with a different systype setting. Suppose, for example, that you want to use the C shell in a SysV environment. Because the C shell is a BSD program, you need to compile it in a BSD environment. When you actually run the program, however, you want all filenames to resolve to the SysV tree. To accomplish this, you need to compile with **-systype BSD4.3** and **-runtype SysV.3**. Note that the **-runtype** setting only affects the run-time semantics for library calls—it does not affect the resolution of **#include** pathnames. See the discussion of **-systype** for more information.

6.3.26 UNIX Version Selection: -systype *systype* (/com/cc) -T *systype* (/bin/cc)

Because C programs are often written to run in UNIX environments, and because not all UNIX environments are the same, Domain C supports the **#systype** preprocessor directive and the **-systype** compilation option, which allow you to define the version of the UNIX system for which your program is targeted.

The Domain C library contains two sets of routines. One is compatible with Bell Labs versions of the UNIX system (System V, Releases 2 and 3) and the other set is compatible with Berkeley's versions of the UNIX system (4.2BSD, and 4.3BSD). All of the routines in both sets work properly in any Domain environment. However, you may encounter problems if you attempt to mix functions from the two sets that interact with each other. In general, it is best to choose one set and stick with it whenever possible.

The two sets of functions overlap to a large extent. It is sometimes the case, however, that while function x exists in both sets, the semantics of the function (and in some cases its arguments) may be subtly different. As an illustration, consider the function **setgrp()**. In the Bell Labs version, the function definition is:

```
int setpgrp()
```

It is defined to "set the process group ID of the calling process to the process ID of the calling process and return the new process group ID." In the Berkeley versions of UNIX systems, there is an identically named function with similar semantics but a different calling sequence. The Berkeley function,

```
setpgrp( pid, pgrp )
int pid, pgrp;
```

"sets the process group of the specified pgrp. Zero is returned if successful; -1 is returned and **errno** is set on failure."

To avoid unexpected behavior, always know which set of functions you are accessing. The system chooses one set of functions over another based on a version selector called the **systype**. The **systype** can affect both the compilation and the execution of a program. At compilation time, it determines which include files the compiler uses. At run time, it determines which set of functions are called and makes sure that the proper calling conventions are employed. However, it is possible to compile with one **systype** and execute the program with a different **systype** by using the **-runtype** option.

To affect the execution of a program, the compiler stamps the object code with the **systype** that was in effect when the module was compiled. This is either the **systype** specified by the **-systype** option, the **#systype** directive, or the **-runtype** option. Note that the **-runtype** option overrides all other systype specifications for determining how the object module is stamped. When the program is executed, the loader checks this stamp and uses the semantics and calling sequences of the designated **systype** when invoking library functions.

There are several ways to define the **systype**, one of which is to place a **#systype** directive in the source file. You may define the **systype** only once per source file. Any subsequent definitions produce an error. Moreover, the **#systype** directive must be the first non-comment token in the source file.

You also can define the target operating system with the **-systype** compile option. The format of **-systype** is as follows:

-systype *systype*

where *systype* can be any of the following:

- **bsd4.1** Berkeley 4.1BSD (obsolete)
- **bsd4.2** Berkeley 4.2BSD
- **bsd4.3** Berkeley 4.3BSD
- **sys3** Bell System III (obsolete)
- **sys5** System V Release 2
- **sys5.3** System V Release 3
- **any** program is independent of a particular UNIX system

If you specify one *systype* on the command line and a different one in the file, the compiler reports an error. If you do not explicitly specify a *systype*, the compiler inherits the

systype from an environment variable called **COMPILESYSTYPE**. By default, this variable is set to **sys5**. If, for some reason, the **COMPILESYSTYPE** variable does not exist, the systype is inherited from another environment variable called **SYSTYPE**. This variable is always set. These environment variables are described in more detail in the *Using Your BSD Environment* and *Using Your SysV Environment* manuals.

6.3.27 Function Prototypes: -type|-ntype (/com/cc)

By default (**-type**), Domain C expects function prototypes for all functions. If the compiler encounters an old-style function declaration or a function invocation prior to a prototype, it will issue an informational message (assuming **-info** is set to level 1 or above). If you are compiling older source files that do not contain prototypes, you should use the **-ntype** option, which suppresses these messages.

The **-type** option also turns on the reference variable feature. If you compile a file with **-ntype**, the compiler will issue errors when it encounters declarations of reference variables.

Finally, **-type** sets the predefined macro **__STDC__** to 1. If **-ntype** is specified, this macro expands to zero.

6.3.28 Line Numbers: -uline|-nuline (/com/cc)

Use **-uline** and **-nuline** to enable or disable any **#line** preprocessor directives in your program.

The **#line** and **#line_number** preprocessor directives establish nondefault line numbers. If you specify **-uline**, the compiler honors these preprocessor directives. However, if you specify **-nuline**, the compiler ignores these preprocessor directives, and therefore, numbers statements according to its normal scheme.

For details on **#line**, see the "#line" listing of Chapter 4.

-uline is the default.

6.3.29 Version Number: -version (/com/cc)

The **-version** option causes the compiler to print the current version number of the compiler. Use this number when reporting APRs (Apollo Product Reports) to Customer Service. If you specify **-version**, you should not specify any other options, nor should you specify a source file. For example:

```
$ cc -version
cc (C compiler), revision 4.89
```

6.3.30 Warning Messages: -warn|-nwarn (/com/cc) -w (/bin/cc)

If you specify **-warn**, the compiler issues any warning messages generated by compilation. If you specify **-nwarn**, the compiler suppresses these warning messages. Note that **-warn** and **-nwarn** do not affect the warning summary; the warning summary is controlled by the **-msgs** and **-nmsgs** options described earlier in this section.

The default is **-warn**.

6.4 Linking in a Domain Environment

There are two commands that enable you to link object modules to form an executable image. The **/bin/ld** utility is the standard UNIX link editor with some Domain enhancements. The **bind** command is the traditional Aegis binder. You can use either command regardless of whether the modules were compiled with **/bin/cc** or **/com/cc**.

6.4.1 The /bin/ld Utility

Use the UNIX link editor, **/bin/ld**, to combine several object modules into one executable program. You can invoke the link editor with the **ld** command or with the **/bin/cc** command. In fact, the link editor is automatically invoked by a **/bin/cc** command if the command line contains **.o** files or a source file containing the **main()** function. The input object modules can come from the following sources:

- Libraries created by **ar** (the UNIX archiver)
- Object modules created by the Domain C, Domain Pascal, or Domain FORTRAN compilers, or the Domain assembler.
- Object modules previously created by **ld**.
- Object modules created by **bind** (the Aegis binder).

One of the primary purposes of **ld** is to resolve external references. If there are any unresolved external references, **ld** will report them. The UNIX utility **nm** can also be used to perform a check of resolved and unresolved global symbols.

When the link editor is called by **/bin/cc**, a startup routine named **/lib/crt0.o** is linked with the program. This routine invokes **main()**. Assuming **main()** returns normally, **crt0.o** finishes by invoking **exit(2)**.

Note that **ld**'s output can either be executed (assuming that there is a start address) or used as input for a further **ld** run. For syntax details on **ld** and its options, see the *BSD Command Reference* manual and the *SysV Command Reference* manual.

6.4.2 The bind Command

The format for the **bind** command is as follows:

$$\$ \textbf{bind} \quad pthnm1 \left[...pthnmN \right] \quad \left[option1 \left[...optionN \right] \right]$$

A *pthnm* must be the pathname of an object file (created by a compiler) or a library file (created by the librarian). Your bind command line must contain at least one *pthnm*.

The available options are described in the *Domain/OS Programming Environment Reference* manual. For example, suppose you write a program consisting of the source files named **test_main.c**, **mod1.c**, and **mod2.c**. To compile the source files using **/com/cc**, you issue the following three commands:

```
cc test_main
cc mod1
cc mod2
```

The **/com/cc** command creates three object files named **test_main.bin**, **mod1.bin**, and **mod2.bin**. To create an executable file named **complete_program** with the **com/bind** utility, enter the following command:

```
bind test_main.bin mod1.bin mod2.bin -b complete_program
```

6.5 Archiving in a Domain Environment

Use the UNIX archiver, **ar**, to create and update library files. Once created, a library file can be used as input to the link editor, **/bin/ld**. As with most linkers, **/bin/ld** will optionally bind only those modules in a library file that resolve an outstanding external reference. For syntax details on **ar** and its options, see the *BSD Command Reference* manual and the *SysV Command Reference* manual.

6.6 System Libraries

There are a number of libraries that come automatically with your operating system. One of these libraries—known as the **standard C library**—is available regardless of whether you run in an Aegis or UNIX environment. The standard library enables you to perform buffered I/O, memory management, double-precision math, and other functions.

Though it is known as the "standard" library, there is no real standard for it. The ANSI C Subcommittee has proposed a standard for the C library, which is expected to be approved by the full ANSI Committee in 1988. In the meantime, the *de facto* standard is the UNIX library, which also agrees with the subset of library functions described by K&R. Domain/OS systems support the UNIX version of the standard library.

In addition to the standard library, Domain/OS also supports lower-level libraries that enable you to perform systems-type operations, such as creating and deleting directories, changing protection codes, and creating new processes. For more information about these library routines, see the *BSD Programmer's Reference* manual, the *SysV Programmer's Reference* manual, and the *Aegis Programmer's Reference* manual. In addition, there are several libraries for performing graphics operations, I/O through streams, and for manipulating windows. For a complete list of manuals that describe these libraries, see the *Technical Publications Master Index*.

6.6.1 The Standard C Library

Although the standard C library exists in a single object file (**/lib/clib**), it is really a conglomeration of many special-purpose libraries. Each sub-library contains routines that cover a particular area of functionality, such as I/O or memory management. Each sub-library has an associated header file. The header file contains the declarations for any related functions, macros, or data types needed to execute a set of library functions. Table 6-7 lists the standard header files. All header files for the standard library reside in **/usr/include** and can be included in your source file by surrounding the filename with angle brackets. For example,

```
#include "/usr/include/stdio.h"
```

and

```
#include <stdio.h>
```

are equivalent in a Domain/OS environment. The second method is preferred because it is more portable. In some cases, the header file is not required but we recommend that you include them anyway. Because they contain prototypes, they enable the compiler to perform type checking of arguments, and they also inhibit unnecessary argument type conversions.

Both the loader and the linkers (**ld** and **bind**) automatically search through **clib** for unresolved references. It is not necessary, therefore, to explicitly link routines from the standard library.

Table 6–7. Header Files

Header File	Functions
assert.h	Diagnostic functions.
ctype.h	Character testing and mapping functions.
curses.h	The curses screen control utility.
errno.h	The **errno** global variable.
malloc.h	Memory management functions.
math.h	Double–precision math functions.
setjmp.h	The **setjmp()** and **longjmp()** functions, which enable you to bypass the normal function call and return discipline.
signal.h	Functions that handle signals.
stdio.h	Buffered I/O functions.
string.h	String manipulation functions.
strings.h	(**BSD only**) String manipulation functions.
time.h	Time functions.
varargs.h	Variable argument list macros.

6.6.2 Built-in Routines

Domain C supports **built-in code** (also called **in-line code**) for many of the routines declared in **string.h**, **strings.h**, and **math.h**. To obtain the built-in versions of these functions, you must include the **<builtins.h>** header file. The functions for which built-in versions exist are as follows:

atan()
atan2()
cos()
exp()
fabs()
log()
sin()
sqrt()
strcat()
strncat()
strcpy()
strcmp()
strlen()
strncpy()
tan()

Normally, when you invoke a library function, the compiler produces code to pass control to the specified function at run time. This requires some overhead since local variables must be preserved and arguments must be passed. When you include **<builtins.h>**, the compiler simply inserts the function's object code wherever it is invoked. While this results in somewhat longer object files, it can produce much faster executable code, particularly when double-precision math functions are used heavily.

> **NOTE:** The built-in functions do not support the error-checking and recovery that normally accompanies library routines. This is particularly important for the **math.h** functions, which check for overflow and assign meaningful values to **errno**. If your programs rely on this error handling, do not use the built-in routines.

6.7 Executing Programs in a Domain/OS Environment

The following sections describe how to execute a program in a UNIX or Aegis environment.

6.7.1 Executing in a UNIX Environment

To execute a program, simply enter its full pathname (including any suffixes). For example, to execute an object file named **a.out**, just enter

```
$ a.out
```

By default, standard input and standard output for the program are directed to the keyboard and display, respectively. You can redirect standard input and output by using the shell's redirection notation (described in *Using Your SysV Environment* and *Using Your BSD Environment*). For example, to redirect standard input when you invoke **a.out**, type

```
$ a.out <trading_data
```

The "<" character redirects standard input from the file **trading_data**. You can redirect standard output in a similar fashion, for example:

```
$ a.out >results
```

This command uses the character ">" to redirect standard output for **a.out** to the file **results**.

6.7.2 Executing in an Aegis Environment

To execute a program, simply enter its full pathname (including any suffixes). For example, to execute an object file named **complete_program**, just enter

```
$ complete_program
```

The operating system searches for a file named **complete_program** according to its usual search rules, then calls the loader utility. The loader utility is user transparent. It binds unresolved external symbols in your executable object file with global symbols in the language and system libraries. Then, it executes the program.

By default, standard input and standard output for the program are directed to the keyboard and display, respectively. You can redirect standard input and output by using the shell's redirection notation (described in the *Using Your Aegis Environment*). For example, to redirect standard input when you invoke **complete_program**, type

```
$ complete_program <trading_data
```

The "<" character redirects standard input to the file **trading_data**. You can redirect standard output in a similar fashion, for example:

```
$ complete_program >results
```

This command uses the character ">" to redirect standard output for **complete_program** to the file named **results**.

> **NOTE:** If the executable object has a suffix (such as **.bin**), you must not forget to type this suffix.

6.8 Debugging Programs in a Domain Environment

The Domain systems support two source level debuggers—**dde** and **dbx**. The following sections describe these sections briefly. For more information about **dde**, refer to the *Domain Distributed Debugging Environment Reference* manual. For information about **dbx**, refer to the *Domain/OS Programming Environment Reference* manual.

6.8.1 The dde Utility

The Domain Distributed Debugger (**dde**) utility is a powerful screen-oriented debugger. To prepare a file for debugging with **dde**, you do not have to do anything special at bind time but you do have to compile with the **-db**, **-dba**, or **-dbs** compiler options. **-db** provides minimal debugger preparation, **-dba** and **-dbs** provide full debugger preparation. Use the following syntax to invoke **dde**:

$$\$\ \textbf{dde}\ \left[\textit{-dde\_options}\right]\ \textit{target\_program\_name}\ \left[\textit{target\_program\_options}\right]$$

where *target_program_name* is the pathname of the program you want to debug. For example, issue the following command to debug the executable object stored in file **complete_program**:

```
$ dde complete_program
```

For complete details on **dde** and its commands, refer to the *Domain Distributed Debugging Environment Reference* manual. Note that **dde** works somewhat differently for C programs than for Pascal programs.

6.8.2 The dbx Utility

dbx is the traditional Berkeley UNIX source language debugger. Although it is usually available only on BSD systems, the Domain/OS version is available regardless of what environment you are running. Note also that, like **dde**, **dbx** can be used on programs compiled with **/com/cc** as well as with programs compiled with **/bin/cc**. The command syntax for invoking **dbx** is:

% dbx [*options*] [*object_file* [*coredump*]]

where *object_file* is the name of the program you want to debug. If you omit the *object_file* name, **dbx** attempts to debug the file **a.out**. If you specify a *coredump* filename, or if a file named **core** exists in the working directory, you can use **dbx** to examine the state of a program that has aborted prematurely.

For complete details about the **dbx** utility, refer to the *Domain/OS Programming Environment Reference* manual.

6.9 Program Development Tools

Domain/OS supports several programming tools that aid in program development, debugging, and source management. Some of these tools are listed below. Most of these utilities are described in detail in the *Domain/OS Programming Environment Reference* manual, although the DSEE facility has its own documentation set. Refer to the appropriate manual for more information about these tools.

cb — Formats a C source file according to user-supplied rules so that it is consistent and readable.

lint — Examines C source files and attempts to detect obscure bugs and nonportable usages. The **lint** utility is described in detail in Appendices C and D of this manual.

make — Creates a program from input object modules according to a list of dependencies that the programmer supplies in a **makefile**. The **make** utility is described in the *Domain/OS Programming Environment Reference* manual.

sccs — **sccs** stands for Source Code Control System, which is a collection of programs that help you maintain a record of versions of a program. The **sccs** utility is described in the *Domain/OS Programming Environment Reference* manual.

DPAK Package — DPAK is a collection of three programs—**DSPST**, **DPAT**, and **HPC**—that allows you to analyze the performance of a program. It is particularly useful for isolating bottlenecks. The DPAK package is described in *Analyzing Program Performance with DPAK*.

DSEE Facility The DSEE (Domain Software Engineering Environment) package is a support environment for software development. DSEE helps engineers develop, manage, and maintain software projects; it is especially useful for large-scale projects involving a number of modules and developers.

Domain/Dialogue Package
The Domain/Dialogue package is a tool for designing the interface to an application program and specifying how the interface should be presented to users of the application. The primary advantage of the Domain/Dialogue package is that it lets you create interfaces separately from the application code.

6.9.1 tb (Traceback)

If you execute a program and the system reports an error, you can use the **tb** utility to find out what routine triggered the error. You invoke **tb** by entering the command

```
$ tb
```

immediately after a faulty execution of the program. (To execute **tb** in a Bourne shell, you must set the **inprocess** environment variable before executing the program.)

For example, suppose you execute object file **complete_program**, encounter an error, and then invoke **tb**. The whole sequence might look like the following:

```
$ complete_program
Enter a value -- 2
?(sh) "./test.bin" - access violation (OS/fault handler)
In routine "_doscan" line 320.
$ tb
access violation (from OS / fault handler)
In routine "_doscan" line 320
Called from "scanf" line 53
Called from "my_rout" line 12
Called from "main" line 6
```

tb first reports the error, which in this case is

```
access violation (from OS / fault handler)
```

Then, **tb** shows the chain of calls leading from the routine in which the error occurred all the way back to the main program block. For example, the error was picked up at line 320 of routine **_doscan**, which was called by routine **scanf** which was called by routine **my_rout** which was called by routine **main**. Given this information, it is probable, though not certain, that there is a problem at line 12 of routine **my_rout**. We make this presumption because **my_rout** is the deepest user-defined routine shown in the traceback.

The *Aegis Command Reference* manual details the **tb** utility.

NOTE: If you compile a file with the **-ndb** option (**/com/cc**), then the functions stored in this file will not be included in the traceback.

Chapter 7

Cross-Language Communication

This chapter describes how to call Pascal and FORTRAN routines from a C program and how to share data between a C program and a FORTRAN or Pascal program. Because many Domain system routines are written in Pascal, the information in this chapter also applies to invoking system routines from C. Briefly, this chapter covers the following topics:

- Using function prototypes to declare parameters
- Understanding data type agreement of Domain C, Pascal, and FORTRAN
- Using reference variables to declare Pascal IN parameters
- Calling Pascal routines from a C program
- Calling FORTRAN routines from a C program
- Sharing data between routines written in different languages
- Using global names
- Calling system service routines

For detailed information about system calls, see the *Domain/OS Calls Reference* manual.

7.1 Suppressing Automatic Type Promotions of Arguments

When you call a C function without a prototype for that function being in scope, the compiler automatically converts the data types of the parameters according to the rules shown in Table 7-1. For communication among C functions, these conversions are usually invisible because the arguments are converted back to the type declared in the formal argument declaration. When calling routines written in other languages, however, it is important to suppress these conversions. The simplest way to suppress these conversions is to declare the external routine with a function prototype. For instance, consider the following program:

```
main()
{
  short j = 3;
  float x = 3.141;

  ex_func( j, x );
}
```

Because there is no prototype for **ex_func()**, the C compiler implicitly casts **j** to an **int** and **x** to a **double** before passing them to **ex_func()**. There is no problem if **ex_func()** is a C routine that expects two arguments of type **short** and **float** because the necessary conversions will occur on the receiving side. However, if **ex_func()** is a Pascal routine that expects arguments of type **INTEGER16** and **REAL** passed by value, the function call will fail. This problem can be avoided by prototyping **ex_func()**:

```
int main( void )
{
  extern void ex_func( short, float );
  short j = 3;
  float x = 3.141;

  ex_func( j, x );
}
```

The prototype causes the C compiler to suppress the automatic argument type promotions. Note that the prototype should be used even if the function is a C routine because it turns on type checking which can identify bugs that would otherwise go unnoticed. For more information about function prototypes, see Chapter 5.

NOTE: Prior to SR10, the Domain C compiler did not support function prototyping. Instead, Domain C supported the reserved word **std_$call**, which turned off automatic type promotions of arguments. Domain C continues to support **std_$call** but it is viewed as an obsolete and inferior means for cross-language communication. We strongly urge you not to use **std_$call** for new programs and to convert your older programs that use **std_$call** to the new prototyping syntax. **std_$call** is described in Appendix E.

Table 7-1. C Function Argument Conversions without Prototypes

Data Type of Argument	Data Type Actually Passed
char	**int**
short	**int**
unsigned char	**unsigned int**
unsigned short	**unsigned int**
float	**double**

7.2 Data Type Agreement in C, Pascal and FORTRAN

Table 7-2 shows equivalences among the three languages' data types. To call a Pascal or FORTRAN routine, make sure that the types declared in the C prototypes are compatible with the types in the definition.

7.2.1 Non-C Data Types

As Table 7-2 shows, the C language has no equivalent types for Pascal's **BOOLEAN** and **SET** types or for FORTRAN's **LOGICAL** and **COMPLEX** types. Section 7.5.4 shows how to simulate the **BOOLEAN** type in C, and Sections 7.6.6 and 7.6.7 show how to simulate FORTRAN's **LOGICAL** and **COMPLEX** data types. It is also possible to simulate the **SET** type in C, but a description of this technique is beyond the scope of this manual. However, the *Programming with Domain/OS Calls* manual describes how to simulate sets in C. (For an interesting discussion of implementing SET functions in C, see *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele Jr.)

7.2.2 Non-FORTRAN Data Types

There are a few C types that have no FORTRAN equivalents. Most of these, however, can be simulated in FORTRAN. *Programming with Domain/OS Calls* describes how to simulate C's structure, union, and enumerated data types. Section 7.6.5 describes how to pass pointers from Domain C to Domain FORTRAN.

There is no easy way to simulate C's **unsigned** types in FORTRAN. Therefore, if you pass an **unsigned** value to a FORTRAN routine, it will be interpreted as a **signed** value. This will only make a difference when the high-order bit is set.

Table 7-2. Domain C, Pascal, and FORTRAN Data Types

C	Pascal	FORTRAN
char, char enum	CHAR	CHARACTER*1
short	INTEGER, INTEGER16	INTEGER*2
int, long	INTEGER32	INTEGER, INTEGER*4
float	REAL, SINGLE	REAL, REAL*4
double	DOUBLE	DOUBLE PRECISION, REAL*8
short enum	enumerated types	INTEGER*2
long enum, enum	INTEGER32	INTEGER*4
struct	record	none
union	variant record	none
pointer (*)	pointer (^)	none
unsigned char	none	none
unsigned short	0..65335	none
unsigned long	0..4295967295	none
none	BOOLEAN	none
none	SET	none
none	none	LOGICAL
none	none	LOGICAL*2
none	none	LOGICAL*1
none	none	COMPLEX
none	none	COMPLEX*16

7.3 Data Type Agreement of Return Value

Just as the parameters must agree in type, so must the function return value. For example, if a Pascal function returns an **INTEGER16** value, you must declare it in your C program as a function that returns a **short**. That is, if the Pascal declaration is

```
FUNCTION func1 : INTEGER16;
```

then the C declaration should be:

```
extern short func1( void );
```

All C declarations of Pascal procedures and FORTRAN subroutines should use the **void** type since these routines do not return a value. For instance, the Pascal procedure defined by

```
PROCEDURE proc1;
```

should be declared as:

```
extern void proc1( void );
```

7.3.1 Functions Returning Pointers

When Pascal returns the value of a function, it places it in one of two registers: a data register (D0) if the value being returned is not a pointer or an address register (A0) if the value is a pointer. C normally expects values to be returned in a data register (D0). Therefore, when you prototype a Pascal function that returns a pointer, you need to tell the C compiler to fetch the returned value from the address register rather than the data register. You do this by appending **#options[a0_return]** to the prototype. For instance, if **pass_point()** is a Pascal function that returns a pointer to an **int**, the prototype would be:

```
extern int *pass_point() #options[a0_return];
```

FORTRAN has no syntax for declaring a function that returns a pointer. All FORTRAN functions return their values in a data register as do C programs so no special syntax is required.

7.4 Argument Passing Conventions

In addition to ensuring that arguments agree in type, you also need to compensate for different passing conventions. Domain FORTRAN passes all arguments by reference, and Domain Pascal passes most arguments by reference. This means that they pass the address of the argument rather than the value of the argument. In contrast, C passes all arguments (except arrays and functions) by value.

Although Domain C passes arguments by value, it provides two mechanisms to simulate passing by reference. The first is to explicitly pass the address of the argument. For example, if **pas_func()** is a Pascal procedure that expects an **integer16** argument, you could invoke it from C with the following statements:

```
int main( void )
{
  extern void pas_func( short * );
  short x;
     .
     .
  pas_func( &x );
     .
     .
```

Note that in the prototype of **pas_func()**, we declare the argument as a pointer. There are two drawbacks with this method. First, it does not provide an easy means for passing constants or expressions since it is illegal to take the address of these. For example, if you want to pass the constant 5 to **pas_func()** you need to store the value in a variable first:

```
int main( void )
{
  extern void pas_func( short * );
  short x;
      .
      .
/*  pas_func( &5 );  ILLEGAL */
  x =5;
  pas_func( &x );  /* Legal */
      .
      .
```

Likewise, if you want to pass the product of two numbers, you must again store the product in a variable before passing it:

```
int main( void )
{
  extern void pas_func( short * );
  short x, y;
      .
      .
/*  pas_func( &(x*y) );  ILLEGAL */
  x *= y;
  pas_func( &x );  /* Legal */
      .
      .
```

The other problem with passing addresses explicitly is that the prototype gives no indication of whether the argument is an IN or OUT parameter. That is, the declaration of **pass_func()** does not reveal whether **pas_func()** will modify the value of the argument or not. You cannot, therefore, assume that the value of **x** will be the same after the call as it was before the call.

Both of these limitations can be avoided by using reference variables, a Domain extension borrowed from C++. Reference variables are described in Sections 3.15 and 5.3.2.

Declaring a parameter as a reference variable in a prototype causes the compiler to pass the argument by reference when the function is invoked. For example:

```
int main( void )
{
  extern void pas_func( short & );
  short x;
      .
      .
  pas_func( x );
      .
      .
```

Note that reference variables make it legal to pass constants and expressions by reference:

```
int main( void )
{
  extern void pas_func( short & );
     .
     .
  pas_func( 5 ); /* Legal */
     .
     .
```

Although this will work, the receiving routine, **pas_func()**, may not modify the constant value passed. If it attempts to modify this value, a run-time access error will occur.

Because there are two ways to pass arguments by reference—explicitly passing addresses or declaring arguments as reference variables—you can set up conventions to use one method in certain situations and the other method in different situations. Domain/OS system calls, for example, use the two methods to distinguish between IN variables and all other type of parameters. In the insert files, all IN variables are declared as reference variables and all other parameters are declared as pointers. A single function might include a combination of pointers and reference variables.

7.5 Pascal Examples

Pascal can pass by reference or by value depending on how a parameter is declared. In Domain Pascal, there are five ways to declare a formal parameter: **IN**, **OUT**, **IN OUT**, **VAR**, or without a keyword. In the first four cases, the parameters are passed by reference. The Pascal keywords control what operations are legal within the Pascal routine. Consult the *Domain Pascal Language Reference* for information about these declaration specifiers.

When you declare a variable in Pascal without one of the declaration specifiers, it directs the compiler to use call-by-value semantics. This means that the Pascal routine will use a local copy of the parameter so that the formal and actual parameters are different objects. The actual parameter in the calling routine will remain unchanged despite any modifications that the called routine makes to the formal parameter.

Domain Pascal uses two methods to achieve call-by-value semantics:

1. For nested routines (routines that are visible only within a single source file), Domain Pascal passes arguments by value just like C.

2. If the routine is globally visible, Domain Pascal assumes that it may be called by routines written in other languages, such as FORTRAN, that only support pass by reference. Therefore, the Pascal routine expects an address of the actual argument and then generates a local copy on the receiving side.

From a Pascal programmer's perspective, the two methods are equivalent since both achieve the call-by-value semantics (that is, the routine operates on a local copy of the

argument). From a C programmer's perspective, however, it is important to know which method is being used. If the first method is being used, you should declare and pass arguments as though you were invoking a function written in C. If the second method is used, you need to pass arguments by reference by either explicitly passing a pointer or by declaring the arguments as reference parameters. (You can force the Pascal compiler to use method 1 by declaring the globally visible routine with the **val_param** option.)

The following examples show how to pass various objects of different types and sizes to Pascal routines.

7.5.1 Passing Integers and Floating-Point Numbers

Passing characters, integers and floating-point values to Pascal programs is fairly straightforward. The prototype for the Pascal function should be type compatible with Pascal function definition. The actual arguments passed must be assignment compatible. To conform with Domain conventions, you should declare IN parameters as reference arguments and all other parameters as pointers. Consider the following Pascal function that raises its first argument to the power specified by the second argument:

```
MODULE power_p;

    FUNCTION power (IN arg1 : SINGLE;
                    IN pow  : INTEGER16) : DOUBLE;

    VAR
      temp : INTEGER16;
      count : INTEGER16;

    BEGIN
      temp := arg1;
      count := pow;
      WHILE count > 1 DO
        BEGIN
          temp := arg1*temp;
          count := count-1;
        END;
      power := temp;
    END;
```

The C program below shows various ways to call **power()**:

```
/* Program name is "call_power_p".  To execute it, you
 *  need to compile this program and the pascal routine
 * in file "power_p.pas", and then bind the two binaries.
 */
#include <stdio.h>

int main( void )
{
  extern double power( float &, short & );
  float x = 2.5;
  short j = 5;
  double z;

  z = power( x, j );
  printf(" %f to the power of %d is %f\n", x, j, z );
  z = power( 3.0, 2 );
  printf(" %f to the power of %d is %f\n", 3.0, 2, z );
  z = power( 2, 3.0 );
  printf(" %f to the power of %d is %f\n", 2, 3.0, z );
}
```

Note that both arguments are declared as reference variables in the prototype because they are declared as Pascal IN parameters. Because they are reference variables, it is legal to pass constants, as illustrated in the second and third invocations. In the third invocation, note that the types of the actual arguments do not match the types declared in the prototype. This is acceptable so long as the actual arguments are assignment-compatible with the prototype parameters. The compiler implicitly casts the first argument to **float** and the second argument to **short** before passing them.

7.5.2 Passing Character Arrays

Pascal supports both fixed-length and variable-length character arrays. In C, strings are fixed-length, but C's convention of ending string with a null character makes them behave like variable-length strings. Having allocated an array of characters, you can store strings of any length in that array as long as they do not exceed the total length of the array.

To facilitate passing strings between the two languages, Domain Pascal supports two run-time functions—**ptoc()** and **ctop()**. The **ptoc()** function appends a null character to a Pascal variable-length string to make it a C-style string. The **ctop()** function helps convert a C-style null-terminated string into a Pascal-style variable-length string. These functions are primarily designed to simplify calling C functions from Pascal. As shown in the example in this section, though, they can also be used when a C function passes a string to a Pascal routine. For more information about these functions, and about Pascal variable-length strings, see the *Domain Pascal Language Reference* manual.

Unlike other type of variables, C arrays are automatically passed by reference. Therefore, if a Pascal routine expects an array argument, you should prototype and invoke the routine as though it were written in C. Do not use reference parameters for array arguments. If you do, you will need to dereference the array before passing it, which will produce very unusual-looking code.

The Pascal program in our example takes one argument: a string with a maximum size of 256 characters. It copies the string to a variable-length string in order to find the length

and then reverses the string. Because s is an IN OUT parameter, the reversed string is available to the calling C routine.

```
MODULE reverse_string;

TYPE
    str = ARRAY[1..256] of CHAR;

VAR
    var_string : VARYING[256] of CHAR;
    temp       : CHAR;
    j          : INTEGER;
    len        : INTEGER;

PROCEDURE reverse_string( IN OUT s : UNIV str );

BEGIN
    j := 1;
    var_string.body := s; {copy s to variable-length string }
    CTOP( var_string );   {set length of var-length string }
    len := var_string.length;
    WHILE j <= len/2 DO
    BEGIN
        temp := s[j];
        s[j] := s[len+1-j];
        s[len+1-j] := temp;
        j := j+1;
    END;
END;
```

The following **main**() function calls **reverse_string**(). C automatically passes arrays by reference so there is no need to precede the array name with an ampersand.

```
/* Program name is "call_reverse_string".  To execute this
 * program, you need to compile this source file with cc,
 * and the "reverse_string.pas" source file with pas, and then
 * link the two object modules.
 */
#include <stdio.h>

int main( void )
{
  extern void reverse_string( char * );
  static char s[] = "reverse this string";

  reverse_string( s );
  printf("%s\n", s );
}
```

The output is:

```
gnirts siht esrever
```

7.5.3 Passing Pointers

In both C and Pascal, pointers are 4-byte entities. The example below shows a simple linked-list application. The C program creates the first element of the list and then calls the Pascal routine **append()** to add new elements to the list. The function **printlist()** is a C routine that prints the entire list. In addition to illustrating how to pass pointers, this example also shows the correspondence of Pascal **records** to C **structures**.

The Pascal program is:

```
MODULE pointer_example;

TYPE
    link = ^list;
    list =
        RECORD
           nex : link;
           data : char;
        END;

PROCEDURE append (firstrec : link;
                  IN val   : CHAR);
    VAR
      newdata : link;

    BEGIN
       new(newdata);          {allocate memory for new element.}

       WHILE firstrec^.nex <> NIL DO
          firstrec := firstrec^.nex;

       firstrec^.nex := newdata;
       newdata^.data := val;
       newdata^.nex := NIL;
    END;
```

The C program is shown below. Note that C's NULL pointer (defined in **<stdio.h>**) is equivalent to Pascal's NIL pointer. The Pascal function expects the first argument to be a pointer to a structure. Because Pascal passes by reference, however, the function is really expecting a pointer to a pointer to a structure, which is how we declare it in the C prototype. When we make the call, we pass the address of **base**, which is a pointer to the first element of the linked list. The second argument is declared as an IN parameter in the Pascal routine so we can declare it as a reference variable in the C prototype. This enables us to pass a character constant as the second argument.

```
/*  Program name is "pass_pointer_c".  To execute this
 *  program, you must also obtain the Pascal program named
 *  "pass_pointer_p". After compiling pass_pointer_p and
 *  pass_pointer_c, you must bind  them together.
 */
#include <stdio.h>

static struct list {
         struct list *next;
         char data;
       };

int main( void )
{
  extern void append( struct list **, char & );
  extern void printlist( struct list * );
  struct list first, *base;

  char ch='z';
  first.data = 'a';  /* assign 'a' to first element of list */
  first.next = NULL;
  base = &first;
  append( &base,'b' );
  append( &base, ch );
  printlist( base );
}

/* printlist() prints the data in each member of the list. */

void printlist( struct list *base )
{
  while (base != NULL )
  {
    printf( "%c\n",base->data );
    base = base->next;
  }
}
```

After compiling and binding these routines, the output is:

```
a
b
z
```

7.5.4 Simulating the BOOLEAN Type

The Pascal **BOOLEAN** type is an 8-bit entity that evaluates to TRUE when its numeric value is -1 and to FALSE when its numeric value is 0. (In a packed record, a **BOOLEAN** uses only one bit.) The **BOOLEAN** type can be simulated in C with the **char** data type. Suppose that you want to call the Pascal routine shown below. This routine takes a **BOOLEAN** argument and returns a **BOOLEAN** result (the opposite of the argument).

```
MODULE pass_boolean_p;

FUNCTION log_not( IN bool_arg : BOOLEAN) : BOOLEAN;
    BEGIN
      writeln('Pascal value of argument:',bool_arg);
      bool_arg := NOT bool_arg;
      writeln('Pascal value returned:   ',bool_arg);
      log_not := bool_arg;
    END;
```

The C program below shows several ways to invoke **bool()**.

```
/*  Program name is "pass_boolean_c".  To execute this
 *  program, you must also obtain the Pascal program named
 *  "pass_boolean_p". After compiling pass_boolean_p and
 *  pass_boolean_c, you must bind them together.
 */
#include <stdio.h>

#define TRUE -1
#define FALSE 0

int main( void )
{
  extern char bool( char & );
  char x;

  printf( "Numeric value of argument: %d\n", TRUE );
  x = bool( TRUE );
  printf( "Numeric value returned: %d\n\n", x );

  printf( "Numeric value of argument: %d\n", FALSE );
  x = bool(FALSE);
  printf( "Numeric value returned: %d\n\n", x );
}
```

The output after compiling, binding and executing is:

```
Numeric value of argument: -1
Pascal value of argument:           TRUE
Pascal value returned:             FALSE
Numeric value returned: 0

Numeric value of argument: 0
Pascal value of argument:          FALSE
Pascal value returned:              TRUE
Numeric value returned: -1
```

7.6 FORTRAN Examples

The following examples show how to pass various objects of different types and sizes to FORTRAN routines. Remember that FORTRAN does not make local copies of parameters—all arguments are passed by reference. Unlike Pascal, FORTRAN does not include syntax to control whether a parameter can or can't be modified within the called function. To be safe, you should assume that the called function may modify any arguments you pass from C. Therefore, you should be careful about passing constants and expressions. If the FORTRAN routine attempts to modify constants or expressions, a run–time error will occur.

There is one restriction concerning the types of data that you can pass to, or return from, a FORTRAN routine:

- You cannot return a character array of any size, including 1, from a FORTRAN function. For instance, a FORTRAN function declared as

  CHARACTER FUNCTION char_func()

  cannot be called from a C program.

As with Pascal, there are two methods for passing arguments from C to FORTRAN: explicitly pass addresses or declare the arguments as reference variables so that the compiler will implicitly pass the address. Either method will work, although only the reference variable method enables you to pass constants and expressions. The choice of which to use is largely a question of style. Using reference variables provides a cleaner interface since the implicit addressing is hidden. On the other hand, this cleanliness can be misleading. Someone reading the code must look at the prototype to realize that the arguments are being passed by reference rather than by value. The examples in Section 7.6.2 through 7.6.7 illustrate both methods.

7.6.1 Names of FORTRAN Routines

The Domain system supports two FORTRAN compile command: **/bin/f77** and **/com/ftn**. Both commands compile FORTRAN source files, but the resulting object files differ slightly. One of the differences is that **/bin/f77** appends an underscore to all global names. This includes names of functions and subroutines as well as names of common blocks.

When you invoke a FORTRAN routine from C, you need to know whether the routine has an appended underscore or not. For example, consider the following FORTRAN function definition:

```
REAL*8 FUNCTION hypot( side1, side2 )
REAL*4 side1, side2
```

If the function is compiled with **/com/ftn**, the C prototype would be:

```
extern double hypot( float &, float & );
```

On the other hand, if the function is compiled with **/bin/f77**, the C prototype would be:

```
extern double hypot_( float &, float & );
```

If you don't know how the function was compiled, you need to look at the object file to see whether the function name has an appended underscore. One way to look at the object file is with the **nm** command, described in the *BSD Command Reference* manual.

7.6.2 Passing Integers and Floating-Point Data

Passing integers and floating-point values to FORTRAN programs is fairly straightforward. The prototype for the FORTRAN function should be type compatible with the FORTRAN function definition. The actual arguments passed must be assignment compatible. The example below shows a FORTRAN subroutine that accepts the values of the two sides of a right-angle triangle and returns the length of the hypotenuse. The parameters are REAL*4 and the result is REAL*8.

```
REAL*8 FUNCTION hypot(side1,side2)
REAL*4 side1, side2

hypot = SQRT((side1*side1) + (side2*side2))

END
```

The first C program below shows how to declare and invoke **hypot()** using pointers as parameters. The second example illustrates the function call using reference variables.

```
/* Passing floats using pointers */
int main( void )
{
  extern double hypot( float *, float * );
  float x = 3.0, y = 4.0;
  double z;

  z = hypot( &x, &y );

/* Note that you cannot pass constants -- the following is
 * illegal

  z = hypot( &3.0, &4.0 )
 */
}
/* Passing floats using reference variables */
int main( void )
{
  extern double hypot( float &, float & );
  float x = 3.0, y = 4.0;
  double z;

  z = hypot( x, y );

/* Note that it is legal to pass constants */

  z = hypot( 3.0, 4.0 )
}
```

7.6.3 Passing Character Data

Passing character data is the same as passing integers, with two exceptions:

- FORTRAN routines expect an additional argument for every character parameter specifying the size of the character array. (For a single character, the size is only one.)
- A FORTRAN routine cannot return character data. To return a character value, create a subroutine and return the character value in a parameter.

Consider the following FORTRAN case-inversion routine that takes two character arguments. The routine inverts the case of the first argument and returns the result through the second argument.

The FORTRAN routine is:

```
SUBROUTINE UPPER_LOWER( in_char, inverted )
CHARACTER in_char,inverted

IF (ICHAR(in_char) .LE. 97) THEN
    inverted = CHAR(ICHAR(in_char) + 32)
ELSE
    inverted = CHAR(ICHAR(in_char) - 32)
END IF

END
```

The following C program shows how to call **upper_lower()**. Note that the first character is declared as a reference variable to allow us to pass character constants; the second parameter is declared as a pointer to prevent us from passing a constant. (Passing a constant would produce a run-time error when the FORTRAN routine attempts to modify its value.) Note also that the size parameters come at the end of the argument list. Both size paramters are declared as reference variables so that we can pass them as constants.

```
/*  Program name is "pass_char_cf".  To execute this
 *  program, you must also obtain the FORTRAN program named
 *  "pass_char_f". After compiling pass_char_cf and
 *  pass_char_f, you must bind them together.
 */
#include <stdio.h>

int main( void )
{
  extern void upper_lower( char &, char *, short &,
                                              short & );
  char out_char,result;       /* 8-bit variables */
  short long_char;            /* 16-bit variable  */

  out_char = 'A';
  long_char = 'b';
  printf( "Original Char\t\tCase-Inverted\n\n" );

  upper_lower( out_char, &result, 1, 1 );
  printf( "\t%c\t\t\t\t\t%c\n", out_char, result );

  upper_lower( 'b', &result, 1, 1 );
  printf( "\t%c\t\t\t\t\t%c\n",'b', result );

  upper_lower( 81, &result, 1, 1 );
  printf( "\t%c\t\t\t\t\t%c\n", 81, result );
}
```

The result of program execution is:

```
Original Char     Case-Inverted

    A                   a
    b                   B
    Q                   q
```

Because the hidden size parameters come at the end of the argument list, you can omit them without affecting your program.

7.6.4 Passing Arrays

There are three points to remember when passing arrays from C to FORTRAN:

- FORTRAN expects the size of each character array to be passed implicitly. In the C prototype for the FORTRAN routine, you should declare this extra argument as a **short**. Size arguments always come at the end of the argument list.
- FORTRAN and C access multidimensional arrays in a different order. In C, the rightmost subscript varies fastest while in FORTRAN the leftmost subscript varies fastest.
- Unlike other variables, C arrays are passed by reference. Therefore, if a FORTRAN routine expects an array argument, you should prototype and invoke the routine as though it were written in C. Do not use reference parameters for array arguments. If you do, you will need to dereference the array before passing it, which will produce very unusual-looking code.

The following example illustrates how to pass a character array from C to FORTRAN. Note that you can declare the array in FORTRAN as a character string or as an array of type

CHARACTER. The two FORTRAN routines shown here return the last character of a string and the next–to–last character, respectively.

```
C Pass a string and get the last char.

        SUBROUTINE pass_char_array(ca, clen, outchar)
        CHARACTER ca(256)
        INTEGER*2 clen
        CHARACTER outchar

C Test for null string.

        IF (clen .LT. 1) THEN
            outchar = ' '
            RETURN
        ENDIF

        outchar = ca(clen)
        RETURN
        END

C Pass a string and get the next-to-last char.

        SUBROUTINE pass_char_string(ca, clen, outchar)
        CHARACTER*256 ca
        INTEGER*2 clen
        CHARACTER outchar

C Test for null string.

        IF (clen .LT. 1) THEN
            outchar = ' '
            return
        ENDIF

        outchar = ca(clen-1:clen-1)
        RETURN
        END
```

The following C program calls these FORTRAN routines.

```
/*  Program name is "pass_char_array_c".  To execute this
 *  program, you must also obtain the FORTRAN program named
 *  "pass_char_array_f".  After compiling pass_char_array_c
 *   and pass_char_array_f, you must bind them together.
 */
#include <stdio.h>

int main( void )
{
  extern void pass_char_string( char &, short &, char &,
                                short &, short & );
  extern void pass_char_array( char *, short &, char &,
                               short &, short & );
  char result;
  static char s1[] = "This is the first string";
  static char s2[] = "This is the second string";

/*  To pass an array declared as a reference variable, you
 *  need to dereference the array.  This is the WRONG way
 *  to pass arrays.
 */
  pass_char_string( *s1, strlen(s1), result, sizeof(s1),
                    sizeof(result) );
  printf( "The second to last character is %c\n", result );

/* To pass an array declared as a pointer, you just pass the
 * array name, as you would in a C-to-C function invocation.
 * This is the RIGHT way to pass arrays.
 */
  pass_char_array( s2, strlen(s2), result, sizeof(s2),
                   sizeof(result) );
  printf( "The last character is %c\n", result );
}
```

The result is:

```
The second to last character is n
The last character is g
```

Note that we need to pass the length of the string twice. The first string length is for the **clen** argument explicitly declared in the FORTRAN routines. The second length is the implict array size that FORTRAN expects for every character argument. The last argument, "1", is the length of the **outchar** parameter.

7.6.4.1 Passing Adjustable Arrays

The following example illustrates how to pass an **adjustable array** from C to FORTRAN. The C program passes two arguments: an array of integers and the size of the array. The FORTRAN routine uses the second argument to declare the size of the array. The routine then returns the average value of the array elements.

```
C  Pass an array of long int and return the average.

        INTEGER*4 FUNCTION pass_int_array(larray, array_len)
        INTEGER*4 array_len
        INTEGER*4 larray(array_len)
        INTEGER*4 i, tot

        tot = 0
        DO i = 1,array_len
          tot = tot + larray(i)
          print *,'larray(',i,') = ',larray(i)
        END DO

        pass_int_array = tot / array_len
        RETURN
        END
```

The C program is:

```
/*  Program name is "pass_int_array".  To execute this program,
 *  you must also obtain the FORTRAN program named
 *  "pass_int_array_f". After compiling pass_int_array_c and
 *  pass_int_array_f, you must bind them together.
 */
#include <stdio.h>

int main( void )
{
  extern int pass_int_array( int *, int & );
  static int average, pass_array[]={ 325, 478, 982,331, 21,
                                     56, 79
                                   };

  average = pass_int_array( pass_array,
                  sizeof(pass_array)/sizeof(pass_array[0]) );
  printf( "The average is: %d\n", average );
}
```

Note that the array is declared as a pointer rather than a reference parameter so that we can pass the array name without dereferencing it; the length is declared as a reference variable so that we can pass the expression that computes the array's length. The result of executing the program is:

```
larray( 1) =  325
larray( 2) =  478
larray( 3) =  982
larray( 4) =  331
larray( 5) =  21
larray( 6) =  56
larray( 7) =  79
The average is: 324
```

7.6.4.2 Passing Multidimensional Arrays

When you pass a multidimensional array, it is important to remember that in C the rightmost subscript varies fastest while in FORTRAN the leftmost subscript varies fastest. The example below shows the consequences of this difference.

The FORTRAN routine is:

```
SUBROUTINE dyn_dim(arr, x, y)
INTEGER*4 x, y
INTEGER*4 arr(x, y)
INTEGER*2 i, j

WRITE(*,*)
WRITE(*,*) 'This is the FORTRAN array:'
DO i = 1, x
  DO j = 1, y
     WRITE(*,*) 'arr(',i,',',j,') = ',arr(i,j)
  END DO
END DO
END
```

The C program is shown below. Note that the array is declared as a pointer to an **int**, just as it would be declared if **dyn_dim()** was a C function. The **x** and **y** arguments are declared as reference parameters so that we can pass constants.

```
/*  Program name is "multi_dim_array_c".  To execute this
 *  program, you must also obtain the FORTRAN program named
 *  "multi_dim_array_f". After compiling multi_dim_array_c
 *  and multi_dim_array_f, you must bind them together.
 */
#include <stdio.h>
#define DIM1 2
#define DIM2 3

int main( void )
{
  extern void dyn_dim( int *, int&, int& );
  static int arr[DIM1][DIM2] = { { 1, 2, 3 },
                                 { 4, 5, 6 }
                               };
  short i,j;

  printf("This is the C array:\n");

  for (i = 0; i<=1; i++)
    for (j=0; j<=2; j++)
      printf( "arr(%d,%d) = %d\n", i, j, arr[i][j] );

  dyn_dim( arr, DIM1, DIM2 );
}
```

The result is:

```
This is the C array:
arr(0,0) = 1
arr(0,1) = 2
arr(0,2) = 3
arr(1,0) = 4
arr(1,1) = 5
arr(1,2) = 6

 This is the FORTRAN array:
 arr( 1, 1) =  1
 arr( 1, 2) =  3
 arr( 1, 3) =  5
 arr( 2, 1) =  2
 arr( 2, 2) =  4
 arr( 2, 3) =  6
```

7.6.5 Passing Pointers

As an extension to the ANSI standard, Domain FORTRAN enables a FORTRAN routine to dereference pointers passed from C or Pascal programs. For complete details, consult the *Domain FORTRAN User's Guide*.

In the following example, the C program passes the FORTRAN subroutine a pointer to a structure that contains four **short** integers. By using the the **POINTER** statement, the FORTRAN subroutine is able to modify the structure elements.

Pay special attention to the C prototype for the FORTRAN routine. We declare the parameter as a pointer to a structure of type **S**, passed by reference. What actually gets passed, therefore, is a pointer to a pointer. We declare the parameter as a reference parameter so that we can pass a constant (the result of the address-of operator).

The FORTRAN subroutine is:

```
SUBROUTINE pass_point(p1)
INTEGER*4 p1
INTEGER*2 a,b,c,d
POINTER/p1/a,b,c,d

a=a+1
b=2**a
c=3**a
d=4**a

END
```

The C program is:

```
/*  Program name is "pass_point_c".  To execute this program,
 *  you must also obtain the FORTRAN program named
 *  "pass_point_f". After compiling pass_point_c and
 *  pass_point_f, you must bind them together.
 */
#include <stdio.h>

typedef struct {
                  short s1,s2,s3,s4;
                } S;

int main( void )
{
  extern void pass_point( S *& );  /* Parameter is a pointer to
                                    * S, passed by reference.
                                    */
  static S struct_pass = { 1, 1, 1, 1 };

  pass_point( &struct_pass );
  printf( "%d\n%d\n%d\n%d\n", struct_pass.s1, struct_pass.s2,
          struct_pass.s3, struct_pass.s4);
}
```

The result is:

```
2
4
9
16
```

7.6.6 Simulating the LOGICAL Types

Domain FORTRAN supports three **LOGICAL** types:

- **LOGICAL** and **LOGICAL*4**
- **LOGICAL*2**
- **LOGICAL*1**

The numbers refer to the length, in bytes, of the type. Note that the default is four bytes long.

Each of these types describes an object that evaluates to TRUE when its numeric value is -1 and to FALSE when its numeric value is 0. In C you can simulate the logical types with integer types of the same size. The following FORTRAN function accepts two arguments: a **LOGICAL*1** and a **LOGICAL*2**, and returns a **LOGICAL*4**. Note that **out_arg** is modified, so we need to be careful not to pass an address of a constant.

```
LOGICAL*4 FUNCTION pass_logical(in_arg, out_arg)
LOGICAL*1 in_arg
LOGICAL*2 out_arg

PRINT *,'FORTRAN value of in_arg:',in_arg
PRINT *,'FORTRAN value of out_arg:',out_arg
out_arg = .NOT. out_arg
pass_logical = in_arg .EQV. out_arg
PRINT *,'FORTRAN value returned:', pass_logical

END
```

The C program below shows how to invoke **pass_logical()**.

```
/*  Program name is "pass_logical_c". To execute this program,
 *  you must also obtain the FORTRAN program named
 *  "pass_logical_f". After compiling pass_logical_c and
 *  pass_logical_f, you must bind them together.
 */
#include <stdio.h>

#define TRUE -1
#define FALSE 0

int main( void )
{
  extern int pass_logical( char &, short * );
  char arg1 = TRUE;
  char arg2 = TRUE;
  int result;

  printf( "C numeric value of arg1: %d\n", arg1 );
  printf( "C numeric value of arg2: %d\n", arg2 );
  result = pass_logical( arg1, &arg2 );
  printf( "C numeric value of arg2 after function call: %d\n",
          arg2 );
  printf( "C numeric value returned: %d\n\n", result );

  printf( "C numeric value of arg1: %d\n", arg1 );
  printf( "C numeric value of arg2: %d\n", arg2 );
  result = pass_logical( arg1, &arg2 );
  printf( "C numeric value of arg2 after function call: %d\n",
          arg2 );
  printf( "C numeric value returned: %d\n\n", result );
}
```

The output after compiling, binding, and executing is:

```
C numeric value of arg1: -1
C numeric value of arg2: -1
 FORTRAN value of in_arg: T
 FORTRAN value of out_arg: T
 FORTRAN value returned: F
C numeric value of arg2 after function call: 0
C numeric value returned: 0

C numeric value of arg1: -1
C numeric value of arg2: 0
 FORTRAN value of in_arg: T
 FORTRAN value of out_arg: F
 FORTRAN value returned: T
C numeric value of arg2 after function call: -1
C numeric value returned: -1
```

7.6.7 Simulating the COMPLEX Types

Domain FORTRAN supports two sizes of complex data types. The FORTRAN **COMPLEX** data type is stored as two 4-byte floating-point numbers, the first representing the real part and the second representing the imaginary part of a complex value. The **COMPLEX*16** type is stored as two 8-byte floating-point numbers.

It is easy to simulate both types in C via structures containing two **float**s or two **double**s. In the following example, the FORTRAN function accepts a **COMPLEX** argument, and returns the square of the argument.

```
COMPLEX FUNCTION sqr_comp( com_param )
COMPLEX com_param

sqr_comp = com_param * com_param

END
```

The C program is:

```
/*  Program name is "pass_complex_c".  To execute this program,
 *  you must also obtain the FORTRAN program named
 *  "pass_complex_f". After compiling pass_complex_c and
 *  pass_complex_f, you must bind them together.
 */
#include <stdio.h>

typedef struct {
                  float real;
                  float imag;
                } COMPLEX;

int main( void )
{
    extern COMPLEX pass_complex( COMPLEX * );
    static COMPLEX result, arg = { 2.5, 3.5 };

    printf( "Complex Number\t\t\tSquare of Number\n\n" );
    result = pass_complex( &arg );
    printf( "(%f,%f)\t\t(%f,%f)\n", arg.real, arg.imag,
             result.real, result.imag );
}
```

The result is:

```
Complex Number          Square of Number
(2.500000,3.500000)     (-6.000000,17.500000)
```

7.7 Data Sharing

As the previous sections illustrated, one way to share data between routines is by passing arguments. The following sections describe two other methods:

- Explicitly define and allude to global variables in the C and Pascal routines.
- Create overlay sections.

Before describing these two techniques, it will be helpful to explain how declarations of global variables get entered into the object file. This is especially important in C because the **/bin/cc** command and the **/com/cc** command handle global declarations differently.

7.7.1 Global Variable Declarations Using /com/cc

> **NOTE:** The description in this section assumes that you do not use the **-bss** switch. If you specify this switch, the compiler will handle global variables as described in Section 7.7.2.

When the **/com/cc** compiler encounters a global definition, it creates a new section in the object file to hold the variable. The name of the new section is the same as the name of the variable. These sections are called **overlay sections** because the linker is allowed overlay sections with the same name. If you include the file scope declaration:

```
int x;
```

in three different source files, the compiler will produce an overlay section named **x** in each of the three resulting object files. When you link these object files together, the compiler overlays the three sections with the same name so that there is only one section for the variable in the resulting executable file.

Because of this overlay technique, it is possible to initialize a global variable in more than one source file (although this is not recommended). The variable gets whatever initial value was overlaid last. (Sections are overlaid in the order in which the files are listed in the link command.) If none of the source files contain an initialization value, the linker initializes the variable to zero.

Note that this discussion refers to global definitions, not global allusions. If you allude to a global variable (precede the declaration with **extern**), the compiler enters the variable into the symbol table as an undefined name. It is up to the linker to resolve this reference by

finding the definition in another object module. If the linker can't resolve an allusion, it reports an error.

7.7.2 Global Variable Declarations Using /bin/cc

Unlike the **/com/cc** compiler, the **/bin/cc** compiler makes a distinction between global definitions that contain an initializer and those that don't. If a compiler encounters a global definitions *with* an initializer, it allocates space for the variable in the **.data** section of the object file, which is where local static data is also kept. If the definition does not contain an initializer, the compiler treats the variable as "weakly defined"—it enters the variable into the symbol table, but does not allocate any storage for it.

When the linker attempts to resolved undefined references, it recognizes these "weakly defined" variables as a special case. If the linker cannot find memory allocated for a weakly defined variable in any of the other object modules, it allocates memory for it in a section named **.bss**. Eventually, therefore, all uninitialized global variables are placed in **.bss**. At run time, the entire section is initialized to zero.

To put a global variable in a named section, as is done with **/com/cc**, you must declare the variable with the **#attribute[section]** specifier, described in Section 3.16.6.

> **NOTE:** When the **.bss** section is used, you must pass object modules through the linker before you can execute them. If you compile with the **/bin/cc** command, the linker is automatically invoked. However, if you compile with **/com/cc** and the **-bss** switch, you must explicitly invoke the linker yourself.

7.7.3 Case Sensitivity and Global Names

Unlike C, Pascal and FORTRAN are case-insensitive, which means that names written in lowercase are the same as names written in uppercase. By convention, both Pascal and FORTRAN export global variables to the linker as lowercase names. Therefore, all C global names that are accessed by FORTRAN or Pascal routines must also be lowercase. C global names that are shared between C modules may use only uppercase and lowercase letters.

7.7.4 Data Sharing Between C and Pascal

There are two ways to declare global variables in Pascal and C such that the linker can resolve references:

- Declare the variables so that they are placed in the **.data** or **.bss** sections.
- Declare the variables so that they are placed in named overlay sections.

7.7.4.1 Declaring .data and .bss Global Variables

In Pascal, an external variable is defined with the **DEFINE** keyword and alluded to with the **EXTERN** keyword. All variables defined with **DEFINE** are placed in the **.data** section

of the the object file. Variables declared as **EXTERN** are listed as unresolved references in the symbol table. For compatible behavior in C, you must compile with **/bin/cc** or use the **-bss** switch with **/com/cc**.

There are several scenarios for declaring and defining variables in Pascal and C. The three most common are described below:

- **Define a variable in Pascal and allude to it in C.** For example, the Pascal source file might contain the following:

  ```
  VAR x: DEFINE INTEGER32 := 0;
  ```

  and the C file would contain:

  ```
  extern int x;
  ```

  In this case, the definition in the Pascal module causes the compiler to allocate space for **x** in the **.data** section. The C declaration produces an undefined reference to **x** in the symbol table, which is resolved by the linker.

- **Define a variable in C (initialized) and allude to in Pascal.** For example, the C file would contain:

  ```
  int x = 10;
  ```

  and the Pascal source file would declare **x** as:

  ```
  VAR x: EXTERN INTEGER32;
  ```

  In this case, the definition of **x** in the C module forces the C compiler to allocate space for **x** in the **.data** section. The declaration of **x** in the Pascal file causes the compiler to produce an undefined reference to **x** in the symbol table, which is resolved by the linker.

- **Define a variable in C (uninitialized) and allude to in Pascal.** For example, the C file would contain:

  ```
  int x;
  ```

  and the Pascal source file would declare **x** as:

  ```
  VAR x: EXTERN INTEGER32;
  ```

  In this case, the uninitialized definition of x in the C module causes the C compiler to make a "weakly defined" entry in the symbol table. The declaration of **x** in the Pascal file causes the compiler to produce an undefined reference to **x** in the symbol table. The linker then places **x** in the **.bss** section, initialized to zero, and resolves the Pascal reference.

It is also possible to define the same variable in C and in Pascal, as long as only one or neither of the definitions contain initializers. If both definitions contain initializers, the linker will report an error.

In the following example, we define the global variable **xx** at the top of the C source file; the function **main()** prints the initial value of **xx** and then calls the C routine **add_three()** which adds 3 to **xx**; finally, **add_three()** calls the Pascal procedure **sub_two()** which subtracts 2 from **xx**.

The Pascal routine is:

```
MODULE global_var_p;

PROCEDURE sub_two;
    VAR
      xx : EXTERN INTEGER32;

    BEGIN
      xx := xx - 2;
      WRITELN('Value of xx after sub_two():',xx);
    END;
```

The C routines are:

```
/*  Program name is "global_var_c".  To execute this program,
 *  you must also obtain the Pascal program named
 *  "global_var_p". After compiling global_var_p and
 *  global_var_c, you must bind them together.
 */
#include <stdio.h>

int xx = 1;               /* Definition of xx */

int main( void )
{
    extern void add_three( void );

    printf( "Initial value of xx: %d\n", xx );
    add_three();
}

void add_three( void )
{
    extern void sub_two( void );

    xx += 3;
    printf( "Value of xx after add_three(): %d\n", xx );
    sub_two();
}
```

The result of executing the program is:

```
Initial value of xx: 1
Value of xx after add_three(): 4
Value of XX after sub_two():         2
```

7.7.4.2 Creating Overlay Data Sections

Both C and Pascal have syntaxes that enable you to produce named overlay sections for global data. Since the binder ensures that overlay sections with the same name refer to the same memory locations, this mechanism enables you to share data across procedures.

In Pascal, you create an overlay section with the syntax:

```
VAR '(' section_name ')'
   declaration
   declaration
      .
        .
        .
```

For instance, the following statements define an overlay section called **example** with two variables.

```
VAR (example)
    x : INTEGER16;
    y : DOUBLE;
```

In C, there are two ways to create overlay sections. If you use the **/com/cc** compiler, you can create an overlay section simply by defining an external variable. All external variables are automatically stored in their own named sections. For instance, if compiled with **/com/cc**, the declarations shown below create three overlay sections called **first_sec**, **second_sec**, and **example**.

```
int first_sec=0;
float second_sec=1.0;
struct {
       short x;
       double y;
       } example;
main()
{
        .
        .
        .
```

Note that **example** contains two variables: **x** and **y**.

If you compile your program with **/bin/cc**, you need to use a special **#attribute[section])** syntax (described in Section 3.16.6) to create a named overlay section:

```
int first_sec #attribute[section(first_sec)] = 0;
float second_sec #attribute[section(second_sec)] = 1.0;
struct {
       short x;
       double y;
       } example #attribute[section(example)];
int main( void )
{
        .
        .
        .
```

Consider the example below. The Pascal program calculates the power of a number. The number, the exponent, and the resulting value are all located in an overlay section accessible to the calling C program.

The Pascal routine is:

```
MODULE section_example_p;

    VAR (sec1)                  { All Pascal names are sent to }
                                { the binder in lowercase.     }
        exponent : INTEGER32;
        value : INTEGER16;
        result : DOUBLE := 1.0;

    PROCEDURE power;
        VAR
          temp : SINGLE;

        BEGIN
        temp := exponent;
            WHILE (temp >=1) DO
              BEGIN
                 result := result*value;
                 temp := temp-1;
              END
        END;
```

The **/com/cc** version of the program is:

```
/*  Program name is "section_example_c".  To execute this
 *  program, you must also obtain the Pascal program named
 *  "section_example_p". After compiling section_example_p and
 *  section_example_c, you must bind them together.
 */
#include <stdio.h>

struct {
       int exp;
       float val;
       double res;
       } sec1;

int main( void )
{
    extern void power( void );

    sec1.val = 5.1;
    sec1.exp = 3;
    power();
    printf( "%f to the power of %d is: %f\n", sec1.val,
            sec1.exp, sec1.res );
}
```

The **/bin/cc** version of the program is:

```
/*  Program name is "section_example_c".  To execute this
 *  program, you must also obtain the Pascal program named
 *  "section_example_p". After compiling section_example_p and
 *  section_example_c, you must bind them together.
 */
#include <stdio.h>

struct {
       int exp;
       float val;
       double res;
       } sec1 #attribute[section(sec1)];

int main( void )
{
    extern void power( void );

    sec1.val = 5.1;
    sec1.exp = 3;
    power();
    printf( "%f to the power of %d is: %f\n", sec1.val,
            sec1.exp, sec1.res );
}
```

The result is:

```
5.100000 to the power of 3 is: 132.650993
```

Note that the names of the variables in the overlay section can be different in the two routines. Their sizes and types, however, should be the same.

7.7.5 Data Sharing Between FORTRAN and C

In FORTRAN, variables are declared external by placing them in a common block. A common block declaration creates an overlay data section. To communicate with a C program, the C program must create an overlay section with the same name. If you compile with **/com/cc**, you can create an overlay section simply by defining an external variable. If you compile with **/bin/cc**, you must use the special **#attribute[section]** syntax, as described in Section 3.16.6. For example:

The FORTRAN program is

```
COMMON /XVAR/ X
INTEGER*4 X
        .
        .
        .
```

The **/com/cc** declaration is:

```
int xvar;
```

The **/bin/cc** declaration is:

```
int xvar #attribute[section(xvar)];
```

Note that the C declaration corresponds to the name of the common block, *not* to the name of the variable in the common block.

If the FORTRAN common block contains more than one external variable, the C source file should define an external structure with the same name as the common block. The fields of the structure should correspond to the variables in the common block. For example, consider the following FORTRAN and C declarations.

Here are the declarations in the FORTRAN source file:

```
COMMON /CNAME/IFIELD,RFIELD
INTEGER*4 IFIELD
REAL RFIELD
      .
      .
      .
```

Here is **/com/cc** version of the declaration:

```
struct {
    int ifield;
    float rfield;
       } cname;
```

Here is **/bin/cc** version of the declaration:

```
struct {
    int ifield;
    float rfield;
       } cname #attribute[section(cname)];
```

Note that the variable is declared as **cname** and not **CNAME** in the C programs. This is because all FORTRAN global names are exported to the linker in lowercase.

The example below illustrates this data-sharing mechanism. The C routine calls a FORTRAN subroutine that evaluates the natural log of a number. The number and the log of the number are global variables that can be accessed by both routines.

The FORTRAN routine is:

```
SUBROUTINE GET_LOG
REAL*4 NUM, LOG_OF_NUM
COMMON /GLOBAL_VARS/ NUM, LOG_OF_NUM

LOG_OF_NUM = LOG(NUM)

END
```

The **/com/cc** version of the program is:

```
/*  Program name is get_log_c".  To execute this program,
 *  you must also obtain the FORTRAN program named "get_log_f".
 *  After compiling get_log_c and get_log_f, you must bind
 *  them together.
 */
#include <stdio.h>

struct S {
       float cnum;
       float clog_of_num;
       } global_vars= { 1.0, 0.0 };

int main( void )
{
  extern void get_log( void );

  printf( "Number\t\t\tNatural Log of Number\n\n" );
  while (global_vars.cnum++ < 10)
  {
    get_log();
    printf( "%f\t\t\t%f\n", global_vars.cnum,
             global_vars.clog_of_num );
  }
}
```

The **/bin/cc** version of the program is:

```
/*  Program name is get_log_c".  To execute this program,
 *  you must also obtain the FORTRAN program named "get_log_f".
 *  After compiling get_log_c and get_log_f, you must bind
 *  them together.
 */
#include <stdio.h>

struct S {
       float cnum;
       float clog_of_num;
       } global_vars #attribute[section(global_vars)] =
                                                { 1.0, 0.0 };

int main( void )
{
  extern void get_log( void );

  printf( "Number\t\t\tNatural Log of Number\n\n" );
  while (global_vars.cnum++ < 10)
  {
    get_log();
    printf( "%f\t\t\t%f\n", global_vars.cnum,
             global_vars.clog_of_num );
  }
}
```

If we compile, bind, and execute, the program produces the following results:

```
Number          Log of Number

2.000000        0.693147
3.000000        1.098612
4.000000        1.386294
5.000000        1.609438
6.000000        1.791759
7.000000        1.945910
8.000000        2.079442
9.000000        2.197225
10.000000       2.302585
```

7.8 System Service Routines

System routines provide a variety of services, including direct manipulation of the display, error handling, interprocess communication, and general input and output. These routines are described in the *Domain/OS Call Reference* manual and the *Programming With Domain/OS Calls* manual.

The system routines follow the standard calling conventions described earlier in this chapter. You should treat them like Pascal routines when passing arguments and variables.

7.8.1 Insert Files

There are a number of **header files** distributed with the operating system and language software. A header file defines constants and type definitions used by the system service routines, as well as declarations of the system service routines themselves.

Each system component has an associated header file. For example, there is a header file for serial I/O, for touchpad manipulation, and for error reporting. All header files are distributed in the directory **/usr/include**. You can include a header file by specifying the full pathname enclosed in double quotes are by enclosing the filename in angle brackets:

#include "/usr/include/*component_name***.h"**

or

#include <*component_name***.h>**

where *component_name* is one of the header files listed in the *Domain/OS Call Reference* manual. Note that all header filenames end with a **.h** suffix.

The example below shows the files needed for a C program that uses the the system I/O routines and system error-handling routines:

```
#include <base.h>
#include <streams.h>
#include <error.h>
```

Always include the header file **<base.h>** first, since some of the other system header files rely on the definitions in this file.

7.8.2 Returned Status Code

Most system routines return a status code as a value of the system's **STATUS_$T** type. The status code indicates whether the routine completed successfully. The value of the status code is **STATUS_$OK** (defined as zero in **base.h**) for successful completion, positive for error-level failure, and negative to indicate a warning-level error. You should check the value of the status code after each system call to find out if errors occurred.

Every nonzero status code is associated with descriptive error text. To analyze the status code and retrieve the text, use the error handling routines described in the *Programming with Domain/OS Calls* manual.

7.8.3 Linking and Execution

The system service routines are included in preinstalled, shared libraries. References to identifiers in these libraries are resolved at execution time. Therefore, you do not need to specify any additional files when compiling or binding a program that calls system service routines. For more information about the linker, refer to *Domain/OS Programming Environment Reference*.

Chapter 8

Input and Output

Input and output are not built-in features of the C language. Instead, C comes with a standard run-time library that covers I/O functions and other operations. In addition to the standard run-time library, there is the UNIX run-time library, which enables you to perform I/O at a lower level, and the Domain/OS system library, which enables you to perform I/O at the lowest level.

Altogether, there are three types of I/O, organized hierarchically. Each higher-level function maps onto one or more lower-level functions, as shown in Figure 8-1.

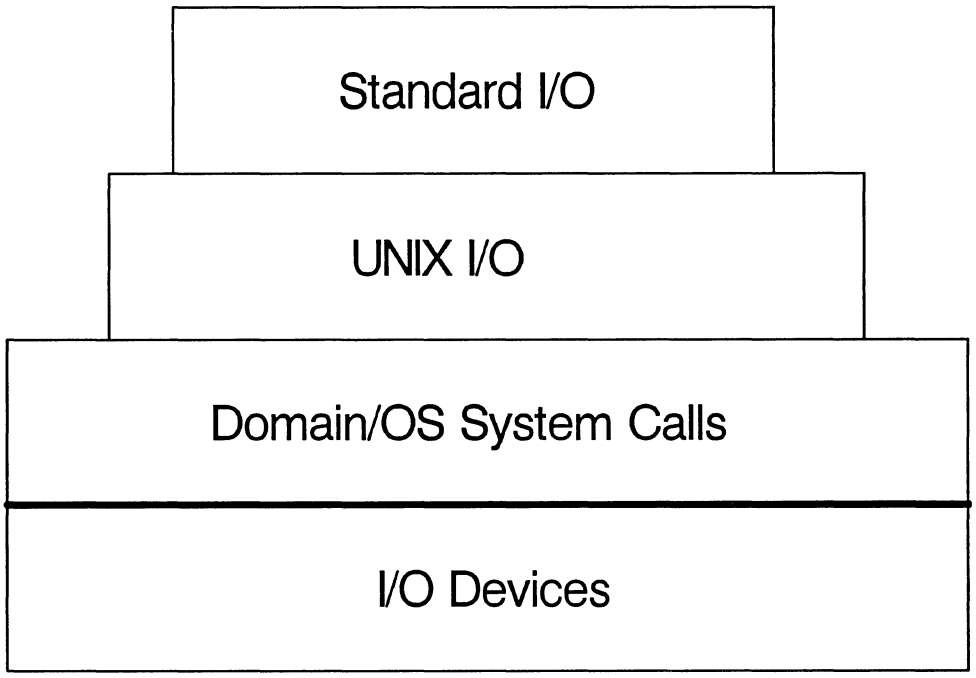


Figure 8-1. Hierarchy of I/O Libraries

Ultimately, all I/O is performed through Domain/OS system calls. The lower levels give you more flexibility, but they are more difficult to use and are not portable. All of the functions described in this chapter are available in all Domain/OS environments. Briefly, the three types of I/O are:

Standard I/O Functions

The standard C I/O library (clib) enables you to open and close files, and to read and write data in a variety of formats. These functions provide automatic buffering by default, but you can override this mechanism. In addition to file I/O functions, the standard library also includes several functions for performing I/O to default input and output devices. The standard I/O functions are the most portable. They are implemented in most C libraries regardless of the operating system.

UNIX I/O Functions

For users writing UNIX applications, these functions enable you to access files and devices via UNIX-compatible system calls. These functions offer many of the same capabilities as the standard I/O functions, but without buffering. In addition, the UNIX calls give you more control in assigning protection attributes to files.

Domain/OS System Calls

At the lowest level, you can access the Domain/OS operating system directly. These calls are more complex than the other two groups and they do not provide any portability. On the other hand, they offer some features that are not available with the other functions. You should use these calls only if portability is not an issue. In particular, you should use the Domain/OS system calls to access mailboxes, perform GPIO operations on peripheral devices, and access files that have a system-defined structure.

This chapter primarily describes performing I/O operations using the standard I/O library. For specific information about the standard I/O functions and the UNIX I/O function see the *SysV Programmer's Reference* and the *BSD Programmer's Reference* manuals. For information about Domain system calls, refer to the *Programming with Domain/OS Calls* manual.

8.1 General Remarks

The next few sections provide an overview of many of the I/O concepts that are common to both the standard buffered I/O library and the UNIX unbuffered library.

8.1.1 File Types

The Domain operating system supports many types of files, including the following:

- Headerless ASCII files
- Fixed-length record files
- Variable-length record files
- User-written type-manager files (extensible streams)
- No defined-record structure files

The Domain/OS system calls enable you to create and access any of these types. With the standard I/O library and UNIX functions, however, you can access only ASCII files. These are files that consist a string of ASCII characters. You can create your own records within such a file by entering a delimiting character, but there is no predefined record structure. Also, you can read and write bytes in numeric rather than string formats, but it is your responsibility to keep track of how data is represented.

8.1.2 Streams and File Descriptors

C makes no distinction between devices such as a terminal or tape drive and logical files located on a disk. In all cases, I/O is performed through **streams** that are associated with the files or devices. A stream consists of an ordered series of bytes. You can think of it as a 1-dimensional array of characters, as shown in Figure 8-2. Reading or writing to a file or device involves reading data from the stream or writing data onto the stream.

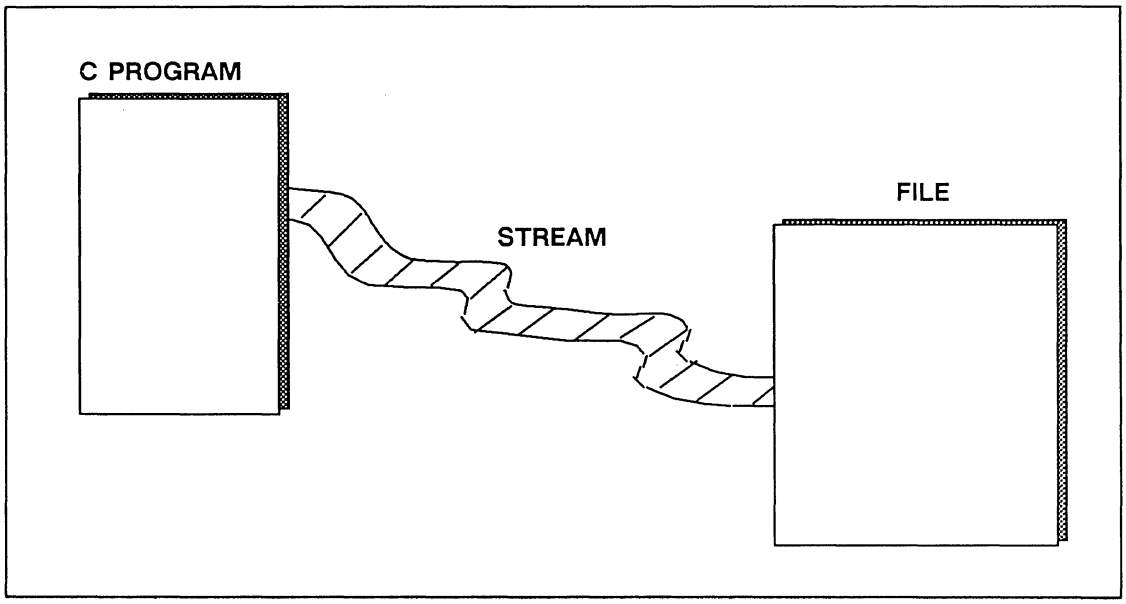


Figure 8-2. C Programs Access Data on Files Through Streams

To perform I/O operations, you must associate a stream with a file or device. For the buffered I/O operations (the ones in the standard I/O library), you do this by declaring a pointer to a structure type called **FILE**. The **FILE** structure, which is defined in the **stdio.h** header file, contains several fields to hold such information as the file's name, its access mode, and a pointer to the next character in the stream.

The **FILE** structures provide the operating system with bookkeeping information, but your only means of access to the stream is the pointer to the **FILE** structure (called a **file pointer**). The file pointer, which you must declare in your program, holds the stream identifier returned by the **fopen()** function. You use the file pointer to read from, write to, or close the stream. A program may have more than one stream open simultaneously, although each implementation imposes a limit on the number of concurrent streams. The limit for Domain/OS systems is 31.

For Unix unbuffered functions, you must also associate a stream with a file, but instead of identifying the file by a pointer to the stream, you identify it with a **file descriptor**. A file descriptor is a unique integer that identifies a particular stream. It is a component of the **FILE** structure. You can obtain a file descriptor with the **open()** function.

Even if you open a file with a standard I/O function, it is possible to extract the file descriptor and access the file through UNIX functions. Conversely, you can open a file with UNIX functions and then access it with standard I/O functions. You should not however, mix UNIX read and write operations with standard I/O read and write operations.

8.2 The Standard I/O Library

The standard, buffered I/O library contains nearly 30 functions for accessing files and devices. We have divided the functions into two groups:

1) Those that access standard streams.

2) Those that access user-defined files and devices.

Before describing the specific functions, however, we discuss the buffering mechanism.

8.2.1 Buffering

Compared to memory, secondary storage devices such as disk drives and tape drives are extremely slow. For most programs that involve I/O, the time taken to access these devices overshadows the time the CPU takes to perform operations. It is extremely important, therefore, to reduce the number of physical read and write operations as much as possible. Buffering is the simplest way to do this.

A **buffer** is an area where data is temporarily stored before being sent to its ultimate destination. Buffering provides more efficient data transfer because it enables the operating system to minimize accesses to I/O devices.

All operating systems use buffers to read from and write to I/O devices. That is, the operating system only accesses I/O devices in fixed-size chunks, called **blocks**. Typically, a block is 512 or 1024 bytes. In Domain/OS systems, blocks are 1024 bytes long by default. This means that even if you want to read only one character from a file, the operating system reads the entire block in which the character is located. For a single read operation, this isn't very efficient, but suppose you want to read 1000 characters from a file. If I/O were unbuffered, the system would perform 1000 disk seek and read operations. With buffered I/O, on the other hand, the system reads an entire block into memory and then fetches each character from memory when necessary. This saves 999 I/O operations.

The C run-time library contains an additional layer of buffering, which comes in two forms: **line buffering** and **block buffering**.

In line buffering, the system stores characters until a newline character is encountered, or until the buffer is filled, and then sends the entire line to the operating system to be processed. This is what happens, for example, when you read data from the terminal. The data is saved in a buffer until you enter a newline character. At that point, the entire line is sent to the program.

In block buffering, the system stores characters until a block is filled, and then passes the entire block to the operating system. Note that these are not the same blocks used by the operating system. To distinguish between the two levels of buffering, we use the term **user-level blocks** to refer to blocks used by the standard I/O library, and **kernel-level blocks** for blocks used by the operating system.

By default, all I/O streams that point to a file are block buffered. Streams that point to your terminal (**stdin** and **stdout**) are line-buffered.

The buffered I/O library package includes a **buffer manager** that keeps buffers in memory as long as possible. So if you access the same portion of a stream more than once, there is a good chance that the system can avoid accessing the I/O device multiple times. Note, however, that this can create problems if the file is being shared by more than one process. For inter-process synchronization, you need to use UNIX unbuffered functions or Domain/OS system calls.

In both line buffering and block buffering, you can explicitly direct the system to flush the buffer at any time (with the **fflush**() function), sending whatever data is in the buffer to its destination.

Although line buffering and block buffering are more efficient than processing each character individually, they are unsatisfactory if you want each character to be processed as soon as it is input or output. For example, you may want to process characters as they are typed rather than waiting for a newline to be entered. C allows you to tune the buffering mechanism by changing the default size of the buffer. You can set the size to zero to turn buffering off entirely. Alternatively, you can use the UNIX unbuffered functions or Domain/OS system calls.

There are several functions in the standard library that allow you to change the buffering parameters of a stream:

void setbuf(FILE *stream, char *buf)
Assigns a specific buffer to a stream rather than using the default buffer. If you pass a null pointer as the buffer, then the stream is unbuffered.

void setbuffer(FILE *stream, char *buf, int size)
(BSD library only) Same as **setbuf()**, but allows you to set the size of the buffer.

void setlinebuffer(FILE *stream)
(BSD library only) Changes **stdin** or **stdout** from block-buffered to line-buffered or unbuffered.

void setvbuf(FILE *stream, char *buf, int type, int size)
(SysV library only) Assigns a specific buffer to a stream. You may specify block-buffering, line-buffering, or no buffering. If you specify block-buffering, you may also specify the size of the block.

In most instances, buffering is invisible. The standard I/O functions make sure that all data is processed as if it were being handled immediately even though it is not. So long as you do not mix buffered calls with unbuffered calls, you should have no problem.

8.2.2 The <stdio.h> Header File

To use any of the standard I/O functions, you should include the **stdio.h** header file. This file contains:

- Prototype declarations for all the I/O functions.
- Declaration of the **FILE** structure.
- Several useful macro constants, including **stdin**, **stdout**, **stderr**, **EOF**, and **NULL**.

EOF is the value returned by many functions when the system reaches the end-of-file marker. **NULL** is the name for a null pointer.

8.2.3 Macros and Functions

A number of the standard I/O functions are implemented as macros rather than functions. Specifically, the macros are:

- **getc()**
- **getchar()**
- **putc()**
- **putchar()**
- **ferror()**
- **clearerr()**
- **feof()**
- **fileno()**

Because they are macros, you should not include side effect operators in the arguments when you invoke them. For example,

```
putc(c, *fp++)
```

causes erroneous results. For **getc()** and **putc()**, you can get around this problem by using **fgetc()** and **fputc()**, which perform the same operation, but are implemented as true functions.

8.2.4 Error Handling

All standard I/O functions return either **NULL** or **EOF** for errors. Both names are defined in **<stdio.h>**, **NULL** as zero and **EOF** as -1. Some functions also return **EOF** when an end-of-file condition is encountered. There are also two flags in the **FILE** structure that indicate whether an error or end-of-file has occurred for the stream. Because **EOF** is returned for both errors and end-of-files, it is often difficult to tell which of these conditions has occurred. Moreover, some functions, such as **getw()**, may return -1 as a valid return value.

To find out for sure whether an end-of-file has occurred, you can call **feof()**, which checks the end-of-file flag and returns 1 if an end-of-file has occurred. Similarly, the **ferror()** function checks the error flag. Neither of these functions, however, resets the flags. To reset the flags, use the **clearerr()** function. If either flag is set, the system will prevent you from performing further operations on the stream.

To summarize, the error-handling routines for standard I/O functions are:

void clearerr(FILE *stream)
Resets the error and end-of-file indicators for the specified stream.

int feof(FILE *stream)
Checks whether an end-of-file was encountered during a previous read operation.

int ferror(FILE *stream)
Returns an integer error code (the value of **errno**) if an error occurred while reading from or writing to a stream.

The following function checks the error and end-of-file flags for a specified stream and returns one of four values based on the results. The **clearerr()** function sets both flags equal to zero.

```
/*  If neither flag is set, stat will equal zero.
 *  If error is set, but not eof, stat equals 1.
 *  If eof is set, but not error, stat equals 2.
 *  If both flags are set, stat equals 3.
 */

#include <stdio.h>
#define EOF_FLAG 1
#define ERR_FLAG 2

char stream_stat( FILE *fp )
{
  char stat = 0;

  if (ferror( fp ))
    stat |= ERR_FLAG;
  if (feof( fp ))
    stat |= EOF_FLAG;
  clearerr( fp );
  return stat;
}
```

8.2.5 File Position Indicators

One of the fields in each **FILE** structure is a **file position indicator** that points to the byte where the next character will be read from or written to. As you read from and write to the file, the operating system adjusts the file position indicator to point to the next byte. Although you can't directly access the file position indicator (at least not in a portable fashion), you can fetch and change its value through library functions (**fseek()** and **ftell()**, thus enabling you to access a stream in non-serial order.

Do not confuse the file pointer with the file position indicator. The file pointer identifies an open stream connected to a file or device. The file position indicator refers to a specific byte position within a stream.

8.2.6 I/O to Standard Devices

There are three streams that are automatically open: **stdin, stdout,** and **stderr**. All three point to your pad by default. The streams **stdin** and **stdout** are both line-buffered. The

stderr stream, which is where error messages are output, is not buffered. At the command level, you can redirect the input and output by using the redirection commands or the pipe facility. To redirect the standard streams within programs, use the **freopen()** function.

The following is a list of all routines that perform input and output to **stdin**, **stdout**, and **stderr**.

int getchar(void) Reads the next character from the standard input stream. **getchar()** is identical to **getc(stdin)**.

char *gets(char *string)
Reads characters from *stdin* until a newline or end-of-file is encountered.

int printf(char *format, ...)
Outputs one or more values according to user-defined formatting rules.

int putchar(char c)
Outputs a single character to the standard output stream. **putchar()** is identical to **putc(stdout)**.

int puts(char *string)
Outputs a string of characters to *stdout*. It appends a newline character to the string.

int scanf(char *format, ...)
Reads one or more values from *stdin*, interpreting each according to user-defined formatting rules.

The *BSD Programmer's Reference* and the *SysV Programmer's Reference* manuals describe each of these functions in detail. The following example, which reads user input, and then writes output, uses several of these routines.

```
/* Program name is "standard_io_example". */
#include <stdio.h>
#define RETURN 10  /* ASCII value of linefeed character */

int main( void )
{
  int age, i = 0;
  static char name[30], profession[30], age_prompt[]= "Age: ";
  static char prof_prompt[] = "Profession: ";

   printf( "Name: " );
   gets( name );
   puts( age_prompt );
   scanf( "%d", &age );
   getchar();       /* Flush linefeed character from buffer. */
   printf( "%s", prof_prompt );
   while(((profession[i++]=getchar()) != RETURN) && (i < 30))
     ;
   profession[i] = '\0';
}
```

A typical execution of the program, with user input, is:

```
Name: John Doe
Age:
37
Profession: Tech Writer
```

The **gets()** function reads characters from **stdin** until a linefeed character is encountered. Although it reads the linefeed character, it replaces it with a null character when it stores the string in memory. The **puts()** function automatically outputs a linefeed following the string. The **scanf()** function takes an address of a variable as its argument. If you use the **%s** format, **scanf()** automatically appends a null character to the input string. **scanf()** does not read the linefeed character at the end of the input. As a result, the first character in the input buffer following a **scanf()** is often a linefeed character. You can discard this character by invoking **getchar()** once, as we did.

Unlike **puts()**, **printf()** does not output a linefeed after each string. The **getchar()** function reads successive characters from **stdin.** If an error or end-of-file occurs, it returns **EOF**. In our program, we call **getchar()** until it reads a linefeed character (ASCII value 10). We then append a null character to make it a true string.

8.2.7 I/O to Files

For each of the functions in the previous section, there is a corresponding function that is exactly the same except that it takes one additional argument, a pointer to a file. There are also additional functions for opening and closing files, listed below (they are listed alphabetically by function name).

int fclose(FILE *stream)
: Closes a stream.

FILE *fdopen(int filedes, char *type)
: Associates a stream with a file descriptor. This enables you to open a file with UNIX functions and then access it with standard I/O functions.

int fflush(FILE *stream)
: Flushes a buffer by writing out everything that has been buffered for the specified stream. The stream remains open.

int fgetc(FILE *stream)
: Same as **getc()**, but it is implemented as a function rather than a macro.

char *fgets(char *s, int n, FILE *stream)

Reads a string from a specified input stream. Unlike **gets()**, **fgets()** enables you to specify a maximum number of characters to read and includes the terminating newline in the string.

int fileno(FILE *stream)

Returns the file descriptor associated with a specified stream. This enables you to open a file with standard I/O functions, and then access it with UNIX functions.

FILE *fopen(char *filename, char *type)

Opens and possibly creates a file, and associates a stream with it. **fopen()** takes two arguments: a pathname identifying the file, and a mode specification that determines what types of operations may be performed on the file. See Section 8.2.8 for more information about this function.

int fprintf(FILE *stream, char *format, ...)

Exactly like **printf()**, except that output is to a specified file.

int fputc(int c, FILE *stream)

Writes a character to a stream. This is the same as **putc()**, but it is implemented as a function rather than a macro.

int fputs(char *s, FILE *stream)

Writes a string to a stream. This is like **puts()**, except that it does not append a newline to the stream.

int fread(void *ptr, unsigned size, unsigned nitems, FILE *stream)

Reads a block of binary data from a stream. The arguments specify the size of the block and where it should be stored.

FILE *freopen(FILE *stream)

Closes a specified stream, and then reopens it for a new file. This is useful for recycling a stream, particularly **stdin**, **stdout**, and **stderr**.

int fscanf(FILE *stream, char *format, ...)

Same as **scanf()**, except that data is read from a specified file.

int fseek(FILE *stream, long offset, int ptrname)

Positions a stream marker. This function enables you to perform random access on a file.

long ftell(FILE *stream)

Returns the position of a stream marker.

int fwrite(void *ptr, unsigned size, unsigned nitems, FILE *stream)
Writes a block of binary data from a specified buffer to a specified stream.

int getc(FILE *stream)
Reads a character from a specified stream.

int getw(FILE *stream)
Reads the next word (four bytes) from a specified stream.

int putc(char c, FILE *stream)
Writes a character to a specified stream.

int putw(int w, FILE *stream)
Writes a word (four bytes) to a specified stream.

void rewind(FILE *stream)
Sets the file position indicator to the beginning of the file for a specified stream.

int ungetc(int c, FILE *stream)
Pushes a character onto a stream. The next call to **getc()** returns this character.

8.2.8 Opening and Closing a File

Before you can read from or write to a file, you must open it with the **fopen()** function. **fopen()** takes two arguments—the first is the file name and the second is the access mode. The text stream modes are shown in Table 8–1. Table 8–2 summarizes the properties of the **fopen()** modes.

When you open a file with one of the + modes, you may read and write to the file. However you cannot write and then read without an intervening **fseek()** or **rewind()** call. Likewise, you may not read and then write without an intervening **fseek()** or **rewind()** call, unless the write operation encounters an end-of-file.

If you use the append mode (**a**), it is impossible to overwrite existing data in the file. Whenever you write to the file, the data is appended at the end regardless of the stream marker's current position.

Table 8-1. fopen() Text Modes

Mode	Description
"r"	Open an existing text file for reading. The system initializes the file position indicator to point to the beginning of the file.
"w"	Create a new text file for writing. If the file already exists, the system will truncate it to zero length, thereby destroying the file's previous contents. The file position indicator is initially set to the beginning of the file.
"a"	Open an existing text file in append mode. You can write only at the end-of-file position. Even if you explicitly move the file position indicator, the system will reassign the inidicator to point to the end of the file prior to any write operation.
"r+"	Open an existing text file for reading and writing. The file position indicator is initially set to the beginning of the file.
"w+"	Create a new text file for reading and writing. If the file already exists, the system will truncate it to zero length, thereby destroy the file's previous contents.
"a+"	Open an existing file or create a new one in append mode. You can read data anywhere in the file, but you can only write data at the end-of-file marker.

The **fopen()** function returns a file pointer that you can use to access the file later in the program. The following function opens a text file called **test** with read access.

```
#include <stdio.h>

FILE *open_test( void );  /* Returns a pointer to a FILE */
{                         /* struct */
  FILE *fp;

  fp = fopen( "test", "r" );
  if (fp == NULL)
    fprintf( stderr, "Error opening file test\n" );
  return fp;
}
```

Note how the file pointer **fp** is declared as a pointer to **FILE**. The **fopen()** function returns a null pointer (**NULL**) if an error occurs. If successful, **fopen()** returns a

non-zero file pointer. The **fprintf()** function is exactly like **printf()**, except that it takes an extra argument indicating which stream the output should be sent to. In this case, we send the message to the standard I/O stream **stderr**. By default, this stream usually points to your terminal.

Table 8-2. File and Stream Properties of fopen() Modes

	Mode					
Property	**r**	**w**	**a**	**r+**	**w+**	**a+**
File must exist before open	*			*		
Truncates file to zero length		*			*	
Can read from stream	*			*	*	*
Can write to stream		*	*	*	*	*
Can write to stream only at end			*			*

We have written the **open_test()** function more verbosely than is usual. Typically, the error test is combined with the file pointer assignment:

```
if ((fp = fopen( "test","r" )) == NULL)
  fprintf( stderr, "Error opening file test\n" );
```

The **open_test()** function is a little too specific to be useful since it can only open one file, called **test**, and only with read-only access. A more useful function, shown below, can open any file with any mode.

```
#include <stdio.h>

FILE *open_file( char *file_name, char *access_mode )
{
  FILE *fp;
  if ((fp = fopen( file_name, access_mode )) == NULL)
    fprintf( stderr, "Error opening file %s with access mode\
%s\n", file_name, access_mode );
  return fp;
}
```

Our **open_file()** function is essentially the same as **fopen()**, except that it prints an error message if the file cannot be opened.

To open `test` from **main()**, you could write:

```
#include <stdio.h>

main()
{
  extern FILE *open_file();

  if ((open_file("test", "r")) == NULL)
    exit(1);
      .
      .
}
```

Note that the **stdio.h** header file is included in both routines. You can include it in any number of different source files without causing conflicts.

8.2.8.1 Closing a File

To close a file, you need to use the **fclose()** function:

```
fclose( fp );
```

Closing a file frees up the **FILE** structure that **fp** points to so that the operating system can use the structure for a different file. It also flushes any buffers associated with the stream. Domain/OS has a limit on the number of streams that can be open at once (128), so it's a good idea to close files when you're done with them. In any event, the system automatically closes all open streams when the program terminates normally. Domain/OS will close open files even when a program aborts abnormally, but it is more efficient for you to close the files yourself.

Bug Alert: Opening a File

In the statement,

```
if ((fp = fopen( "test","r" )) == NULL)
  fprintf( stderr, "Error opening file test\n" );
```

the parentheses around,

```
fp = fopen( "test", "r" )
```

are necessary because == has higher precedence than =. Without the parentheses, **fp** gets assigned zero or one, depending on whether the result of **fopen()** is a null pointer or a valid pointer. This is a common programming mistake.

8.2.9 Reading and Writing Data

Once you have opened a file, you use the file pointer to perform read and write operations. The standard I/O library supports three degrees of I/O **granularity.** That is, you can perform I/O operations on three different sizes of objects. The three degrees of granularity are as follows:

- One character at a time
- One line at a time
- One block at a time

Each of these methods has some pros and cons. In the following sections, we show three ways to write a simple function that copies the contents of one file to another. Each uses a different degree of granularity.

One rule that applies to all levels of I/O is that you cannot read from a stream and then write to it without an intervening call to **fseek()**, **rewind()**, or **fflush()**. The same rule holds for switching from write mode to read mode. These three functions are the only I/O functions that flush the buffers without disconnecting the stream.

8.2.9.1 One Character at a Time

There are four functions that read and write one character to a stream:

getc()	A macro that reads one character from a stream.
fgetc()	Same as **getc()**, but implemented as a function.
putc()	A macro that writes one character to a stream.
fputc()	Same as **putc()**, but implemented as a function.

Note that **getc()** and **putc()** are usually implemented as macros whereas **fgetc()** and **fputc()** are guaranteed to be functions. Because they are implemented as macros, **putc()** and **getc()** usually run much faster. In fact, on Apollo computers, they are almost twice as fast as **fgetc()** and **fputc()**. Because they are macros, however, they are susceptible to side effect problems (see Section 8.2.3). For example, the following is a dangerous call that may not work as expected:

```
putc( 'x', fp[j++] );
```

If an argument contains side effect operators, you should use **fgetc()** or **fputc()**, which are guaranteed to be implemented as functions.

The following example uses **getc()** and **putc()** to copy one file to another.

```
#include <stdio.h>
#define FAIL 0
#define SUCCESS 1

int copyfile( char *infile, char *outfile )
{
  FILE *fp1, *fp2;

  if ((fp1 = fopen( infile, "r" )) == NULL)
    return FAIL;
  if ((fp2=fopen( outfile, "w" )) == NULL)
  {
    fclose( fp1 );
    return FAIL;
  }

  while (!feof( fp1 ))
    putc( getc( fp1 ), fp2 );

  fclose( fp1 );
  fclose( fp2 );
  return SUCCESS;
}
```

The **getc()** function gets the next character from the specified stream and then moves the file position indicator one position. Successive calls to **getc()** read each character in a stream. The **feof()** function returns a nonzero value if the stream's end-of-file flag is set.

8.2.9.2 One Line at a Time

Another way to write this function is to read and write lines instead of characters. There are two line-oriented I/O functions—**fgets()** and **fputs()**. The prototype for **fgets()** is:

char *fgets(char *s, int n, FILE *stream* **);**

The three arguments have the following meanings:

s	A pointer to the first element of an array to which characters are written.
n	An integer representing the maximum number of characters to read.
stream	The stream from which to read.

fgets() reads characters until it reaches a newline, an end-of-file, or the maximum number of characters specified. **fgets()** automatically inserts a null character after the last character written to the array. This is why, in the following **copyfile()** function, we specify the maximum to be one less than the array size. **fgets()** returns **NULL** when it reaches the end-of-file. Otherwise, it returns the first argument. The **fputs()** function writes the array identified by the first argument to the stream identified by the second argument. The prototype for **fputs()** is:

char *fputs(char *s, int n, FILE *stream* **);**

The three arguments have the following meanings:

s	A pointer to the first element of an array from which characters are read.
n	An integer representing the maximum number of characters to write.
stream	The stream to which characters are written.

One point worth mentioning is the difference between **fgets()** and **gets()** (the function that reads lines from **stdin**). Both functions append a null character after the last character written. However, **gets()** does not write the terminating newline character to the input array. **fgets()** does include the terminating newline character. Also, **fgets()** allows you to specify a maximum number of characters to read, whereas **gets()** reads characters indefinitely until it encounters a newline or end-of-file. There is a similar difference between **fputs()** and **puts()**. **puts()** appends a newline to the end of each string it writes, but **fputs()** does not.

The following function illustrates how you might implement **copyfile()** using the line-oriented functions.

```
#include <stdio.h>

#define FAIL 0
#define SUCCESS 1
#define LINESIZE 100

int copyfile( char *infile, char *outfile )
{
  FILE *fp1, *fp2;
  char line[LINESIZE];

  if ((fp1 = fopen( infile, "r" )) == NULL)
    return FAIL;
  if ((fp2 = fopen( outfile, "w" )) == NULL)
  {
    fclose( fp1 );
    return FAIL;
  }
  while (fgets( line, LINESIZE-1, fp1 ) != NULL)
    fputs( line, fp2 );
  fclose( fp1 );
  fclose( fp2 );
  return SUCCESS;
}
```

You might think that the **copyfile()** version that reads and writes lines would be faster than the version that reads and writes characters because it requires fewer function calls. Actually, though, the version using **getc()** and **putc()** is significantly faster. This is because Domain/OS systems implement **fgets()** and **fputs()** using **fputc()** and **fgetc()**. Since these are functions rather than macros, they tend to run more slowly.

8.2.9.3 One Block at a Time

In addition to character and line granularity, you can also access data in lumps called **blocks**. Note that these are *user-level* blocks, not *kernel-level* blocks. You can think of a block as an array. When you read or write a block, you need to specify the number of elements in the block and the size of each element. The two block I/O functions are **fread()** and **fwrite()**. The prototype for **fread()** is

```
int fread( void *ptr, int size, int nmemb, FILE *stream );
```

The arguments represent the following data:

ptr	A pointer to an array in which to store the data.
size	The size of each element in the array.
nmemb	The number of elements to read.
stream	The file pointer.

fread() returns the number of elements actually read. This should be the same as the third argument unless an error occurs or an end-of-file condition is encountered.

The **fwrite()** function is the mirror-image of **fread()**. It takes the same arguments, but instead of reading elements from the stream to the array, it writes elements from the array to the stream.

The following function shows how you might implement **copyfile()** using the block I/O functions. Note that we test for an end-of-file condition by comparing the actual number of elements read (the value returned from **fread()**) with the number specified in the argument list. If they are different, it means that either an end-of-file or an error condition occurred. We use the **ferror()** function to find out which of the two possible events happened. If an error occurred, we print an error message and return an error code. Otherwise we return a success code. For the final **fwrite()** function we use the value of **num_read** as the number of elements to write, since it is less than **BLOCKSIZE**.

```
#include <stdio.h>
#define FAIL 0
#define SUCCESS 1
#define BLOCKSIZE 512
typedef char DATA;

int copyfile( char *infile, char *outfile )
{
  FILE *fp1,*fp2;
  DATA block[BLOCKSIZE];
  int num_read;


  if ((fp1 = fopen( infile, "r" )) == NULL)
  {
    printf( "Error opening file %s for input.\n", infile );
    return FAIL;
  }

  if ((fp2 = fopen( outfile, "w" )) == NULL)
  {
    printf( "Error opening file %s for output.\n", outfile );
    fclose( fp1 );
    return FAIL;
  }

  while ((num_read = fread( block, sizeof(DATA),
          BLOCKSIZE, fp1 )) == BLOCKSIZE)
    fwrite( block, sizeof(DATA), num_read, fp2 );

  fwrite( block, sizeof(DATA), num_read, fp2 );
  fclose( fp1 );
  fclose( fp2 );

  if (ferror( fp1 ))
  {
    printf( "Error reading file %s\n", infile );
    return FAIL;
  }
  return SUCCESS;
}
```

Like **fputs()** and **fgets()**, the block I/O functions are usually implemented using **fputc()** and **fgetc()** functions, so they are not as efficient as the macros **putc()** and **getc()**. Note also that these block sizes are independent of the blocks used for buffering. The buffer size, for instance, might be 1024 bytes. If the block size specified in a read operation is only 512 bytes, the operating system will still fetch 1024 bytes from the disk and store them in memory. Only the first 512 bytes, however, will be made available to the **fread()** function. On the next **fread()** call, the operating system will fetch the remaining 512 bytes from memory rather than performing another disk access. The block sizes in **fread()** and **fwrite()** functions, therefore, do not affect the number of device I/O operations performed.

8.2.10 Random Access

The previous examples accessed files sequentially, beginning with the first byte and accessing each successive byte in order. For a function such as **copyfile()**, this is reasonable since you need to read and write each byte anyway. In this case, it's just as fast to access them sequentially as any other way.

For many applications, however, you need to access particular bytes in the middle of the file. In these cases, it is more efficient to use C's two random access functions—**fseek()** and **ftell()**.

The **fseek()** function moves the file position indicator to a specified position in a stream. The prototype for **fseek()** is:

int fseek(FILE ***stream*, **long int** *offset*, **int** *whence* **);**

The three arguments are:

stream	A file pointer.
offset	An offset measured in characters (can be positive or negative).
whence	The starting position from which to count the offset.

There are three choices for the *whence* argument, all of which are designated by names defined in **stdio.h**:

SEEK_SET	The beginning of the file.
SEEK_CUR	The current position of the file position indicator.
SEEK_END	The end-of-file position.

For example, the statement,

```
stat = fseek(fp, 10, SEEK_SET)
```

moves the file position indicator to character 10 of the stream. This will be the next character read or written. Note that streams, like arrays, start at the zero position, so character 10 is actually the 11th character in the stream.

The value returned by **fseek()** is zero if the request is legal. If the request is illegal, **fseek()** returns a nonzero value. This can happen for a variety of reasons. For example, the following is illegal if **fp** is opened for read-only access because it attempts to move the file position indicator beyond the end-of-file position:

```
stat = fseek(fp, 1, SEEK_END)
```

Obviously, if **SEEK_END** is used with read-only files, the offset value must be less than or equal to zero. Likewise, if **SEEK_SET** is used, the offset value must be greater than or equal to zero.

The **ftell()** function takes just one argument, which is a file pointer, and returns the current position of the file position indicator. **ftell()** is used primarily to return to a specified file position after performing one or more I/O operations. For example, in most text editor programs, there is a command that allows the user to search for a specified character string. If the search fails, the cursor (and file position indicator) should return to its position prior to the search. This might be implemented as follows:

```
cur_pos = ftell( fp );
if (search( string ) == FAIL)
  fseek(fp, cur_pos, SEEK_SET);
```

Note that the position returned by **ftell()** is measured from the beginning of the file.

The example in the next section illustrates random access, as well as some of the other I/O topics discussed in this chapter.

8.2.10.1 Printing a File in Sorted Order

Suppose you have a large data file composed of records. Let's assume that the file contains one thousand records, where each record is a **VITALSTAT** structure, as declared below in a file called **vitalstat.h**:

```
#define NAME_LEN 19
typedef char NAME[NAME_LEN];
typedef struct date {
  unsigned day :   5,
           month : 5,
           year : 11;
} DATE;

typedef struct vitalstat
{
   NAME vs_name;
   char vs_ssnum[11];
   DATE vs_date;
   char vs_jersey;
} VITALSTAT;
```

Suppose further that the records are arranged randomly, but you want to print them alphabetically by the **vs_name** field. First, you need to sort the records. We can do this by creating an index for each record.

The following function reads the key field (**vs_name**) of every record, and stores them in an array of structures that contain just two fields—the record id (index) and the key.

We assume that the data file has already been opened, so that the function is passed a file pointer. The include file **recs.h** contains the following:

```
#include "vitalstat.h"
#include <stdio.h>
#define MAX_REC_NUM 1000
typedef struct {
                 int index;
                 NAME key;
               } INDEX;
```

```
/*  Reads up to max_rec_num records from a file and stores the
 *  key field of each record in an index array.  Returns the
  * number of key fields stored.
 */

#include "recs.h"

int get_records( FILE *data_file, INDEX names_index,
                 int max_rec_num)
{
  int offset = 0, counter = 0;

  for (k = 0; !feof( data_file ) && counter < max_rec_num; k++)
  {
    fgets( names_index[k].key, NAME_LEN, data_file );
    offset += sizeof(VITALSTAT);
    if (fseek( data_file, offset, SEEK_SET ) &&
       (!feof( data_file )))
    {
       fprintf(stderr, "Problem accessing file\n");
       exit( 1 );
    }
    counter++;
  }
  return counter;
}
```

The function reads the first **NAME_LEN** characters of each record using **fgets()** and stores them in the array **names_index**, then moves the file position indicator to the beginning of the next record with **fseek()**. In this way, we avoid reading extraneous parts of the record. In reality, of course, the I/O buffering mechanism fetches blocks of 1024 characters, so the entire records are read anyway. Within each buffer, however, we need only access the first field in each record. This saves us memory-to-memory data copying time, even though we don't save any device-to-memory processing time. For large records, which span blocks, this approach could also save you device-to-memory processing time.

The next task is to sort the array of **NAMES_INDEX** structures. This function, which makes use of the library function **qsort()**, is shown below. The return value is a pointer to an ordered array of **NAMES_INDEX** structures.

```
/*  Sort an array of NAMES_INDEX structures by the
 *  name field.  There are index_count elements to be
 *  sorted.  Returns a pointer to the sorted array.
 */

#include "recs.h"

void sort_index( INDEX names_index, int index_count)
{
  int j;
  static int compare_func(); /*  Defined in this file. */
/*  Assign values to the index field of each structure.
 */
  for (j = 0; j < index_count; j++)
    names_index[j].index = j;

  qsort( names_index, index_count, sizeof(INDEX),
                                          compare_func );

  return names_index;
}


static int compare_func( NAMES_INDEX *p, NAMES_INDEX *q )
{
  return strcmp( p->name, q->name );
}
```

The next step is to print out the records in their sorted order. We definitely need to use **fseek()** for this function because we need to jump around the file. We can compute the starting point of each record by multiplying the index value with the size of the **VITALSTAT** structure. If each **VITALSTAT** structure is 40 characters long, for example, record 50 will start at character 2000. After positioning the file position indicator with **fseek()**, we use **fread()** to read each record. Finally, we print each record with a **printf()** call.

```
/*  Print the records in a file in the order
 *  indicated by the index array.
 */

#include recs.h

void print_indexed_records( FILE *data_file, INDEX index[],
                            int  index_count )
{
  VITALSTAT vs;
  int j;

  for (j = 0; j <= index_count; j++)
  {
    if (fseek( data_file,
           sizeof(VITALSTAT) * index[j].index,
           SEEK_SET ))
       exit( 1 );
    fread( &vs, 1, sizeof(VITALSTAT), data_file );
    printf( "%20s, %hd, %hd, %hd, %12s", vs.name, vs.bdate.day,
             vs.bdate.month, vs.bdate.year, vs.ssnum );
  }
}
```

To make this program complete, we need a **main()** function that calls these other functions. We have written **main()** so the filename can be passed as an argument.

```
#include "recs.h"

int main( int argc, *argv[] )
{
  extern int get_records();
  extern void sort_index();
  extern int print_indexed_records();

  FILE *data_file;
  static INDEX index[MAX_REC_NUM];

  int num_recs_read;

  if (argc != 2)
  {
    printf( "Error: must enter filename\n" );
    printf( "Filename: " );
    scanf( "%s", filename );
  }
  else
    filename = argv[1];

  if ((data_file = fopen( filename, "r" )) == NULL)
  {
    printf( "Error opening file %s.\n", filename );
    exit( 1 );
  }

  num_recs_read = get_index( data_file, index, MAX_REC_NUM );
  sort_index( index, num_recs_read );
  print_indexed_records( data_file, index, num_recs_read );
  exit( 0 );
}
```

8.3 UNIX Unbuffered I/O Functions

Although these functions are called "unbuffered," they do not bypass the disk buffering that occurs at the lowest levels of the operating system. These functions are called unbuffered because they do not use the additional layer of buffering employed by the standard I/O library.

Whereas the standard I/O functions access a stream through a stream pointer, UNIX unbuffered I/O functions operate through a **file descriptor**. A file descriptor is an integer that identifies a channel between a stream and a file or device. A unique file descriptor is returned whenever you open or create a file. Each process can support up to 20 file descriptors, numbered 0 through 19. By default the standard devices have the following file descriptors:

standard device	**file descriptor**
stdin	0
stdout	1
stderr	2

The basic UNIX I/O functions are shown in table 8–3.

Table 8–3. UNIX I/O Functions

Function	What It Does
close()	Closes a file. This function breaks the connection between a file descriptor and a file, allowing you to reuse the descriptor.
creat()	Creates a new file or re-creates (overwrites) an existing file. This function enables you to assign specific protection attributes to a file.
lseek()	Moves a stream marker. This function is similar to **fseek()**, but it uses a file descriptor instead of a stream pointer.
open()	Opens a file. This function is similar to **fopen()**, but it returns a file descriptor instead of a stream pointer.
read()	Reads a block. This function is similar to **fread()**, but blocks are unbuffered.
write()	Writes a block. This function is similar to **fwrite()**, but blocks are unbuffered.
unlink()	Deletes a file.

In addition to these functions, there are a number of functions that enable you to access directory files and change the protection attributes of data files, but these are beyond the range of this manual. For information about these functions, see the manuals *BSD Programmer's Reference* and the *SysV Programmer's Reference* manuals.

The following example is a file copy function using the UNIX I/O library.

```
/* Program name is "unix_copy". */
#include <fcntl.h>
#define BUFSIZE 100

unix_copy( char *infile, char *outfile )
{
   int fdin, fdout, nbuf;
   char buf[BUFSIZE];

   if ((fdin=open( infile, O_RDONLY )) == -1)
   {
     perror("Error");
     exit();
   }

   if ((fdout=open( outfile, O_WRONLY | O_CREAT, 066 )) == -1)

   {
     perror( "Error" );
     exit();
   }

   while ((nbuf = read( fdin, buf, sizeof(buf))) > 0)
        write( fdout, buf, nbuf );
        if (nbuf == -1)
           perror("Error");
  if (close( infile ) == -1 || close( outfile ) == -1)
    perror( "Error" );
}
```

This routine performs the same operation as the file copy functions listed previously using standard I/O calls. But in this function, we define our own buffering. Data is read in and written in 100-byte chunks.

8.3.1 UNIX I/O Error-Handling

Like the standard I/O functions, UNIX I/O functions return -1 or 0 when an error occurs, but they do not use the names **EOF** and **NULL**. Also, instead of setting flags in the **FILE** structure, they use a global variable called **errno**. This variable is assigned a positive integer value that represents a specific error message. Chapter 9 lists all the error codes and messages. After an error has occurred, you check to see which error it is by looking at the value of **errno** (**errno** is defined in **<errno.h>**, which you must include in the source file). There is also a function called **perror()** that prints out the message corresponding to **errno**'s current value. As with the standard I/O flags, you must explicitly reset **errno**.

Chapter 9

Diagnostic Messages

This chapter details the error, warning, and informational messages that the C compiler produces. An **error** indicates a problem severe enough to prevent the compiler from creating an executable object file. A **warning** is less severe than an error; a warning does not prevent the compiler from creating an executable object file. The warning message tells you about a potential ambiguity in your program for which the compiler believes it can generate the correct code. **Informational messages** are intended to inform you of potential problems in your program.

The C compiler always outputs error messages; warning messages can be suppressed by compiling with the **-nwarn** option. There are four levels of informational messages. You can select the level you want with the **-info** option. To suppress all informational messages, specify **-info 0** or **-ninfo** (this is the default).

When the compiler outputs a diagnostic message, it lists the following information:

- The error, warning, or informational message number. This is an integer symbolizing a message. In Section 9.2, we list all messages by number.
- The line number in the source code where the problem was detected. (Occasionally, the given line number is one or more lines after the line containing the error.)
- The line of source code where the problem was detected.
- The actual message.

The compiler includes in the message invalid symbols defined by the program, but it can identify only those symbols defined after preprocessor execution.

The Domain C compiler is designed to compile code as quickly as possible. This means that there are minimal error recovery mechanisms. Although the compiler does attempt to recover from errors, a single mistake can produce cascading errors. Therefore, the cardinal rule of error-fixing in C is:

Worry about the first reported error only!

For instance, if the compiler reports twenty errors, stare at the first one, for it may have indirectly triggerered the other nineteen. Now, it is entirely possible that some or all of the other nineteen errors may be real errors that you will have to take action on, but don't waste your time on them until you are sure that they are real errors. Fix the first one and then recompile.

9.1 Common C Programming Mistakes

We draw your attention to the most commonly made C programming mistakes:

- Forgetting a semicolon at the end of a statement.
- Putting a semicolon where it is not needed, for instance, at the end of a pre-processor directive, or after a function's argument list.
- Forgetting to balance braces; that is, you must have the same number of left braces { and right braces }.
- Confusing = with ==. (This confusion will cause a run-time error, not a compile time error.)
- Forgetting to use the ampersand (&) in front of an argument to the **scanf** function. (This will probably cause a run-time error, and possibly a compile time warning.)

9.2 Domain C Compiler Messages

Here is a list of the C compiler error, warning, and informational messages:

1 ERROR `Unterminated comment.`

You forgot to close a comment. Remember that you begin a comment with /* and close it with */.

2 ERROR `Improper numeric constant.`

For example, you entered a numeric constant of the form 0xreal. This implies that you are trying to specify a hexadecimal floating-point number. The number following 0x must be a hexadecimal integer.

3 ERROR `Unterminated character string.`

You started a string, but you did not finish it. Remember that you must enclose a string with double-quotes. A common trigger for this error is calling **printf()** and forgetting to end the string before you list the data arguments.

4 ERROR `Bad syntax` *TOKEN*.

The compiler encountered *TOKEN* when it was expecting to find something else.

5 ERROR `Illegal module name` *module_name*.

A module name must be a legal identifier. See Chapter 2 for identifier rules. (The most common mistake is to begin the module name with a digit instead of a letter.)

6 ERROR `Quoted string is too long; maximum size is 4095 characters.`

You should break the long string into several shorter strings.

7 ERROR `-DEF option has no name to define.`

You specified the compiler option **-def** with the format:

```
-def = value
```

rather than the correct form which is:

```
-def name = value
```

See Chapter 6 for details on **-def**.

8 WARNING `Old-fashioned assignment operator; taken as` *assignment_operation*.

This is only a warning. Some older C compilers let you use assignment operators in a format opposite to that of modern C compilers. For instance, some older C compilers let you use the assignment operator =+ instead of +=. You should use the modern format (i.e., +=).

9 ERROR `"void" is illegal for` *identifier* `in this context.`

For a complete discussion of **void**, see Chapter 3.

11 ERROR `Missing right parenthesis on declaration.`

In a declaration, the number of right parentheses must match the number of left parentheses.

12 ERROR `Storage class` *specifier* `is illegal in this context; default assumed.`

You used a storage class specifier inappropriately. For example, you cannot specify **auto** or **register** when declaring a global variable. As a second example, you cannot specify **static, auto,** or **extern** on a parameter declaration. See Chapter 3 for a complete discussion of storage classes.

13 WARNING `Old-fashioned initialization; missing "=".`

This is only a warning. Some older C compilers allow you to initialize variables with the format

data_type variable initial_value;

Domain C lets you use this format, but we suggest that you use the modern C format which is

data_type variable = initial_value;

14 ERROR `Unrecognizable item` *token*`; syntax error in declaration.`

There are many possible causes for this error. One common cause is that you put a semicolon after a function definition. (When you remove the semicolon many other errors will probably go away.) Sometimes this error occurs when the compiler is expecting to find an identifier but finds *token* instead. (See Chapter 5 for a discussion of function syntax.)

15 ERROR `When allocated, size of` *array_name* `was zero.`

Possibly, you omitted the array size when defining *array_name* but you forgot to initialize the array; for example, compare the following two definitions:

```
char str[];              /* wrong */
char str[] = "Hello"; /* right  */
```

Another possibility is that you declared an array improperly, and consequently, the compiler did not allocate any space for it. See Chapter 3 for details about array declaration.

18 ERROR Array dimension *token* is not an integer constant.

When you declare the number of elements in an array, the number must be a positive integer value. For example, compare the following two declarations:

```
int a[3];   /* right */
int a[3.2]; /* wrong */
```

See Chapter 3 for details about array declaration.

19 ERROR Array dimension *token* is either zero or negative.

When you declare the number of elements in an array, the number must be a positive integer value. See Chapter 3 for details about array declaration.

20 ERROR Too many enumerators for "enum" type; max is 1024.

See Chapter 3 for details on enumerated variables.

21 WARNING "long" or "short" in this context is meaningless and ignored.

long and **short** can only be applied as prefixes to the data types **int**, **unsigned int**, or **float**. They cannot be applied to any other data type.

22 WARNING "unsigned" in this context is meaningless and ignored.

The **unsigned** keyword can only be applied to integer data types. Floating-point data types cannot be made unsigned.

23 ERROR *Identifier* has not been declared.

If it seems like you did declare it, then just make sure that your spelling matches the spelling of the variable in the definition.

24 ERROR `Multiple declaration of` *identifier*`, previous declaration was on line` *number*`.`

Possibly, you used the same identifier as both a parameter and a local variable. For example, the following function will trigger this error:

```
f(arg)
int  arg;
{
   int arg;
    .
    .
    .
}
```

Another possibility is that you declared the same variable twice in the same block.

25 WARNING `Repeated item` *token* `is ignored.`

You probably repeated the same data type prefix (like **long**, **short**, or **unsigned**) twice in the same declaration.

26 ERROR `Illegal type of constant` *token* `for "enum" type.`

The compiler encountered *token* when it expected to encounter an integer value. See Chapter 3 for details about enumerated variables.

27 ERROR `Improper use of "void" type for function; ("int" type assumed).`

A function can return the type "void", but it cannot return an aggregate type that uses void as its base type.

28 WARNING `"enum" constant` *number* `exceeds 16 bits.`

Since enumerated constants are stored as **signed short ints** by default, any number over +32767 or under –32768 will cause an overflow problem. Use **long enum** to store larger constants. Chapter 3 explains enumerated constants.

30 ERROR `Parameter` *identifier* `was not listed in the function declaration.`

You have defined a function parameter named *identifier*, but you did not put *identifier* in the function heading. See Chapter 5 for details on function syntax.

31 ERROR Dynamic aggregate variable *identifier* cannot be initialized.

The only kind of local aggregate variable that can be initialized is a static one. For instance, compare the following dynamic aggregate variable declarations:

```
f()       /* a function declaration */
{
            int a[2] = {500, 400};   /* Illegal */
  auto      int a[2] = {500, 400};   /* Illegal */
  register  int a[2] = {500, 400};   /* Illegal */
  extern    int a[2] = {500, 400};   /* Illegal */
  static    int a[2] = {500, 400};   /* Legal */
    .
    .
    .
}
```

35 WARNING In function *function_name*, parameter *identifier* was listed but never declared; ("int" type assumed).

This is only a warning. Assuming that you wanted *identifier* to be an **int**, you can ignore this warning. However, it is bad programming style to accept the default data types in parameter declarations. See Chapter 5 for details about proper function syntax.

36 ERROR Multiple declaration of *identifier* in parameter_list.

You've declared the same parameter more than once in the parameter list. See Chapter 5 for details on proper function syntax.

37 ERROR Cannot assign "void" from *function_name*.

You cannot assign a void function to a non-void lvalue. For example, the following function call triggers this error:

```
int i;
i = (void) printf("Bon Jour\n");
```

38 ERROR Label *name* on line *number* is outside of the scope of the goto.

You specified label *name*, but *name* is not defined within the current function. (See the "goto" listing for details.)

39 ERROR Improper parameter declaration *token*.

All the arguments in the argument list must be identifiers, but *token* is not an identifier.

40 ERROR *Token* et cetera is not an lvalue.

An "lvalue" is any C entity that can appear on the left side of an assignment statement. Here is a partial list of some things that are not lvalues, but which programmers often mistake for lvalues:

- An entire array (though a single component of an array is an lvalue)
- A constant specified by a #define statement.

A possible trigger for this error is to define an *n*-dimensional array, but to access it with less than *n* components.

41 ERROR Unknown type token in a structure or union.

The compiler was expecting a valid C data type and encountered *token* instead. Perhaps you misspelled the data type, or perhaps you put the data type in uppercase, or perhaps you just plain forgot the data type. See Chapter 3 for details about C data types.

42 ERROR Function declaration *function_name* is illegal in a structure or union.

You cannot declare a function as a component of a structure or union. Note that the compiler sees a function declaration as any phrase of the following form:

IDENTIFIER()

Perhaps you were trying to declare an array and used parentheses instead of brackets.

43 ERROR "switch" expression type is not an integral type.

The C integral types are **int**, **char**, and **enum**. For details on **switch**, see the "switch" listing in Chapter 4.

44 ERROR *Value* is not of the correct type for the "switch" on line *number*.

The most common cause of this error is that you used a floating-point *value* in a **case** statement. (C will not convert the floating-point number to an integer value.) For details on **switch**, see the "switch" listing in Chapter 4.

45 WARNING "switch" expression type is unsigned, but constant *name* is negative.

The C compiler has detected a probable mistake in your programming logic since this constant will never be equal to the switch expression. For details on **switch**, see the "switch" listing in Chapter 4.

46 ERROR *Value* has already occurred as a "case" constant on line *number*.

You cannot specify the same *value* for a **case** statement more than once in the same **switch** statement. Note that the C compiler evaluates the **case** expression, so although you may have specified two different expressions, if they evaluate to the same *value*, then this error occurs. For details on **switch**, see the "switch" listing in Chapter 4.

47 ERROR *Token* is not a valid option specifier.

You put a pound sign (#) in column 1 and then followed it with some token other than an identifier or a number. For example, the following expression triggers this error:

```
# "Aloha"
```

48 ERROR Include file name is not a string; found *token*.

C expected to find a string pathname immediately following **#include**, but it found *token* instead. Remember that a string consists of characters enclosed in double-quotes or angle brackets. For example, compare the following #include statements:

```
#include    /sys/ins/test.ins.c     /* wrong */
#include  "/sys/ins/test.ins.c"     /* right */
#include  </sys/ins/test.ins.c>     /* right */
```

49 ERROR Nested includes are too deep (> 16).

A header file can itself contain header files, which themselves can contain header files which can contain..., but there can be no more than 16 levels of header files. Since exceeding this depth is rather unlikely, this error is more likely to be caused by an include file that includes the file that had included it. (It's rather like two facing mirrors producing infinite reflections.) For instance, if program **main.c** lists "inc.c" as an include file, and file **inc.c** lists **main.c** as an include file, this error will be triggered.

50 ERROR *Token* is not a recognized option, or it does not begin in column 1.

This error is triggered by one of the following two mistakes. First, perhaps you mistakenly started a preprocessor directive in a line other than the leftmost column. Second, you put the pound sign # in column 1, but you did not put a legal preprocessor directive immediately after it. See Section 4.3 for an overview of preprocessor directives.

51 ERROR Include file *pathname* is not available.

You have specified an include file with the **#include** preprocessor directive, but the compiler cannot find it. Possibly, *pathname* does not exist or you have misspelled it, or perhaps network problems prevent the compiler from seeing the pathname.

53 ERROR Multiple declaration of *identifier* in a structure or union.

You cannot use the same identifier more than once in the same structure or union declaration. For details on structure and union declarations, see Chapter 3.

55 ERROR Bad syntax in a struct/union/enum; *Token* found.

See Chapter 3 for details on declaring structure, union, and enumerated variables. Possible triggers for this error include:

* You mistakenly separated two enumerated constants with a semicolon instead of a comma.
* You forgot to put a closing brace after the last enumerated constant.
* You mistakenly used a right parenthesis) or bracket] instead of a brace }.

56 ERROR Multiple definition of *label*, previous definition was on line *number*.

You cannot define the same *label* more than once in the same block. To correct the error, simply rename the second occurrence.

58 ERROR Bad syntax in a struct/union/enum; *token* found; assuming end of list.

An unneeded *token* has slipped into your declaration. Removing *token* should clear up the error. See Chapter 3 for details about declaring struct, union, and enum variables.

60 ERROR Improper use of *token*, only a variable or constant is valid here.

One possibility is that you mistakenly used a label name as the argument to a **case** statement. For example, the following program fragment causes this error:

```
abc:
    switch(i)
    {
      case abc: break;
    }
```

61 ERROR *Variable_name* is not an array.

Probably, you've used *variable_name* as if it were an array, but it is not. Note that C interprets any expression of the form

```
IDENTIFIER[]
```

as an attempt to access an array. A second possibility is that you defined a 1-dimensional array, but you tried to access it as a 2-dimensional array.

62 ERROR *Variable_name* is not a pointer variable.

You tried to use *variable_name* in a way that only a pointer variable can be used. For instance, maybe you tried to dereference *variable_name*, but you can only dereference a pointer variable. See the "pointer operations" listing in Chapter 4 for details.

63 ERROR *Variable_name* is not a structure or union.

You used *variable_name* in a manner that is appropriate for a structure or union variable only. Note that C interprets expressions of the form

```
identifier.token  OR
identifier->token
```

as an attempt to access a structure or union.

64 ERROR *Identifier* is not a member of *struct_or_union_name*.

It appears to the compiler that you are trying to access a member of a structure or union, but *identifier* not a declared member of this structure or union. Perhaps you misspelled *identifier* or perhaps there is a mistake in your structure or union declaration. See the "structure and union operations" listing in Chapter 4 for information about using structures and unions in the body of a function, or see Chapter 3 for information on declaring structures and unions.

66 ERROR Bit field constant *number* is not an integer.

Bit fields must be integers. See Section 3.8.4 for details on bit fields.

67 ERROR Improper use of *identifier*, only a function reference is valid here.

You used an expression of the form *identifier(token)*, but *identifier* was not a function name. A common mistake is to use parentheses ()instead of brackets (for an array) or braces (for comments).

68 ERROR The types of *token1* and *token2* are not compatible with the *operator_name* operator.

See Chapter 4 for descriptions of all the operators. Common mistakes include:

- Using the modulo operator (%) for floating-point division.
- Using floating-point expressions as arguments to the bit-shift operators.

69 ERROR The type of *variable_name* is not compatible with the *operator_name* operator.

See Chapter 4 for descriptions of all the operators.

70 ERROR Incompatible operands [*operand1*, *operand2*] to the *operator_name* operator.

This error can be triggered in many ways. Possibly, you've misused the = operator. Another possibility is that you've called a function using a format like this

```
answer = function();
```

but the data type the function will return cannot be converted to answer's data type. For example, if the function returns **void**, then answer must be **void** also. If you've misused an operator, see Chapter 4. But, if you've had a problem calling a function, see Chapter 5.

71 ERROR Subscript [*subscript*] to array *array_name* is not of the correct type.

The implicit or explicit data type of *subscript* must be compatible with the **int** type. For instance, you'll get this error if *subscript* is a pointer variable, but you won't get this error if *subscript* is an integer or enumerated value.

72 WARNING No path to statement *statement*.

This is only a warning, but it could very well mean that there is a mistake in your coding. The warning tells you that there is no way that the program will ever reach *statement*. This warning is usually caused by a **goto** statement or by a **return** statement (if it is unconditionally called and if it is not the last line of the function).

73 ERROR No declaration for type *type*.

A superfluous comma is the culprit here; notice the right and wrong ways to use the comma operator inside a declaration:

```
int  i, ;   /* wrong, causes error 73 */
int  i,j;   /* right */
int  ,i ;   /* wrong, causes error 73 */
```

74 ERROR Function *function_name* may not be defined inside another function; (*identifier* begins the definition).

Unlike some other structured languages, C does not support nested functions. Since you probably already know that you cannot nest functions, you probably made some other mistake. Did you forgot to end the previous function with a closing brace? Perhaps you mistakenly placed a pair of parentheses right after an identifier name. (This means "function definition" to the C compiler regardless of what you wanted it to mean.)

75 WARNING sizeof *identifier* is zero.

You have specified an expression whose storage allocation is zero bytes. Perhaps you mistakenly declared an array without an explicit size, and then forgot to supply an initial value that would allow the compiler to set its size.

76 ERROR Illegal cast type for *variable*.

You cannot cast *variable* to the stated data type. See the "casting operations" listing in Chapter 4.

77 ERROR Cannot initialize external variable *variable_name*.

This error highlights one of the subtler distinctions in C—that between allusion and definition. The storage class specifier **extern** indicates that you are alluding to a variable. Note that you can initialize a variable when you define it, but you cannot initialize a variable when you allude to it.

78 WARNING Incompatible pointer and integer operands [*operand1*, *operand2*] to the *operator_name* operator.

See the "pointer operations" listing of Chapter 4. For example, consider some right and wrong ways to mix pointers and integers:

```
VARIABLE = POINTER + INTEGER;              /* wrong */
POINTER_VARIABLE = POINTER + INTEGER;      /* right */
VARIABLE = *(POINTER + INTEGER);           /* right */
```

79 ERROR Illegal type of constant *token* for variable *variable_name*.

The compiler was expecting a constant of a particular data type. You supplied a constant, but it was of the wrong data type. For instance, code like the following triggers this error:

```
struct  {int  a;}  x = {3.14};  /* wrong */
struct  {int  a;}  x = {3};     /* right */
```

80 WARNING Illegal pointer combination: incompatible types.

C is flexible about converting data types; however, C is not so flexible that it allows you to mix pointer variables that point to two different types. For example, a pointer to an **int** cannot be assigned to a pointer to a **char**.

84 ERROR Named bit field identifier cannot have a size of 0.

A named bit field must have an integer value greater than or equal to 1. See Chapter 3 for details on structures and unions.

86 ERROR Unrecognized statement *keyword*.

You've used *keyword* (probably **break**) in an illegal context. For instance, you cannot use **break** outside of a **for**, **while**, or **do/while** loop or outside of a **switch** statement.

87 ERROR "goto" label expected; *token* found.

If you specify a token followed by a colon, C assumes that you are specifying a label. Although most computer languages accept numbers as labels, C only accepts identifiers as labels. (Remember, a number is not a legal identifier.) See Chapter 2 for a definition of identifiers.

89 WARNING Non-standard usage: partial member reference *field_name* resolved.

This is only a warning. Ignore the warning if you do not plan to port the program to another system. If you are trying to write portable code, then you will have to specify *field_name* in the standard way. (See the "structure and union operations" listing of Chapter 4 for details.)

90 WARNING Ambiguous reference; more than one member named *identifier*.

See the "structure and union operations" listing in Chapter 4 for details on this error message.

91 ERROR Illegal type for bit field *identifier*.

The only kind of data type that can be packed down into a bit field is an **int** or **unsigned int**. See Section 3.8.4 for details on bit fields.

92 ERROR Address operator is illegal for bit field *identifier*.

It is okay to take the address of a member of a structure or union. However, you cannot take the address of a bit field (even if the bit field starts on a byte boundary). For example, the following code triggers this error:

```
struct x {unsigned a : 2} y;

z = &(y.a);
```

93 ERROR Input line too long; it has been truncated.

You have exceeded the line limit of 1024 characters.

94 WARNING Negative shift constant *value* may give undefined results.

The compiler is warning you that a negative shift might not give the expected results. Note that C supports both a left shift operator << and a right shift operator >>, so instead of trying to use a negative shift value, perhaps you should just use the other shift operator with a positive shift value. See the "bit operators" listing in Chapter 4 for details.

95 ERROR Too many include files.

There is no fixed limit on the number of include files. In fact, even a program with a small number of **#include**s can cause this error if the include files themselves contain other include files.

96 ERROR Constant value *token* cannot be evaluated at compile time.

The compiler was expecting a constant that could be evaluated at compile time. Certain constants cannot be evaluated at compile time. For example, any constant that relies on an address cannot be evaluated until run time.

97 ERROR Label *label* is never defined.

You made one of three mistakes. First, perhaps you just plain forgot to define the label (see the "goto" listing in Chapter 4 for details on labels). Second, perhaps the spelling of the label does not match the spelling in the **goto** statement. Third, perhaps the label is defined in one function and the **goto** statement is in another function. (They must be in the same function.)

98 ERROR Line exceeds maximum length of *number* by *number* characters.

This line is too long; divide it into multiple lines.

99 ERROR Left brace ({) expected; *token* found.

There are many possible causes for this error. In particular, you should check to see that you are not missing a { immediately after the parameter declarations. Another possibility is that at the line prior to the line where the error was reported, there was a faulty function declaration.

100 ERROR Right brace (}) expected; *token* found.

You started a block with the left brace {, but the compiler did not encounter its matching right brace. Sometimes, this error occurs when you do a lot of nesting and forget to close a function or a loop. Another possibility is that you forgot to terminate a comment or a string.

101 ERROR Statement terminator expected; *token* found.

Probably, you forgot a semicolon at the end of a statement. Another possibility is that a letter somehow crept into a number; for example, maybe a line contained the numeric constant 1.2f3 instead of 1.2e3. A third possibility is that you forgot to enclose a compound statement within a pair of braces.

102 ERROR Improper argument list; *token* found.

You tried to call a C library routine, but your list of arguments does not look right. If the mistake occurred in a **printf()** call, make sure that you put the comma in the right places, for example:

```
printf("%d\n", count);    /* right */
printf("%d\n," count);    /* wrong */
```

103 ERROR *Keyword* expr must begin with "("; *token* found.

The keywords **if, switch, while,** and **for** must be followed by a parenthesized expression. If something besides a comment comes between one of these keywords and a left parenthesis, the compiler issues this error.

104 ERROR *Keyword* expr must end with ")"; *token* found.

The keywords **if, switch, while,** and **for** must be followed by a parenthesized expression. Apparently, you started the parenthesized expression, but you forgot to finish it with a right parenthesis.

105 ERROR Colon (:) expected in "case" or "default"; *case/default* found.

This error may confuse you because it will probably be reported at the line beneath where the error actually occurred. For instance, if the error was reported at line 20, look at line 19. The problem can be remedied by putting a colon after the **case** statement.

106 ERROR "case" or "default" expected in "switch"; *token* found.

See the "switch" listing in Chapter 4.

107 ERROR More than one "default" case given for a "switch".

See the "switch" listing in Chapter 4.

108 ERROR Illegal return expression (*expression* et cetera) for "void" function.

The function heading specifies that the function will not return any value to the caller. Therefore, you cannot specify any expression with **return**. For instance, compare a legal and an illegal return for a **void** function:

```
void f()
{
  .
  .
  .
  return;                  /* legal */
  return (expression);     /* illegal */
}
```

109 ERROR Cannot initialize null array *identifier*.

This error is a byproduct of some other error. The other error prevented the compiler from allocating any space for the array. An array with no space is a null array, and you cannot initialize a null array. Fix the other error and you'll cure this one too.

110 ERROR "while" expected in "do" statement; *token* found.

In a **do/while** loop, you must place the keyword **while** immediately after the closing } of the loop.

111 ERROR ";" expected in "for" statement; *token* found.

The parenthesized list that immediately follows the keyword **for** must contain exactly two semicolons.

112 ERROR String initializer too long for *array_name*; truncated to fit.

You declared *array_name* to hold *n* components, but you are initializing *array_name* with more than *n* values. It may surprise you to find that the following **char** array declaration triggers this error:

```
char alpha[5] = {"abcde"}
```

Although it seems like a snug fit between the five-element array and the five-**char** initialization string, in actuality, the string "abcde" takes up six components. The sixth component is the terminating null character that Domain C automatically supplies for you. If you want to be sure that you've defined the perfect array size, just omit the array size as explained in Chapter 3.

113 WARNING Structure or union member *member_name* has size of zero.

This warning will be associated with an error that explains what went wrong when you declared *member_name*. If you fix the associated error, this warning should go away. For details about structure and union declarations, see Chapter 3.

115 WARNING Function *function_name* is declared as an argument.

This is only a warning. The compiler was expecting parameter declarations here, but got another function declaration instead. Remember that the compiler interprets any expression of the form

```
IDENTIFIER()
```

as a function declaration. So, when the compiler issues this warning you should ask yourself, "Did I put the parentheses in the right place?"

116 ERROR Improper expression; *token* found.

Many situations could have caused this error. In particular, check for a missing comma or semicolon in a variable declaration. Conversely, check for an extra semicolon or comma.

117 ERROR Identifier expected; *token* found.

See Chapter 2 for a definition of identifier. The most common mistake is in trying to start an identifier with a digit rather than a letter. Another possible trigger for this error is that you used a keyword where an identifier was expected. See Chapter 2 for a list of Domain C keywords. Another possibility is that you were supposed to supply one or more identifiers inside a set of parentheses, but you supplied the parentheses without the identifiers.

118 ERROR Member name expected; *token* found.

Your source code contained an expression of the format

```
structure_or_union_variable.
```

This format is illegal. The correct format is

```
structure_or_union_variable.member
```

120 ERROR Illegal operation on pointer to function *function_name*.

You cannot do mathematical operations on a pointer to a function.

121 ERROR Function *function_name* returns more than 32K bytes.

You are trying to pass a very large amount of data back to the calling function. Probably, you are trying to pass back a structure that contains a very large array. Can you reduce the size of this array? If not, can you make the structure into a global variable that does not have to be returned to the caller?

122 WARNING Function *function_name* needs *number* bytes of stack, which approaches the maximum stack size of *number* bytes.

You are passing arguments to this function that take up a lot of space. The sort of argument that could trigger this error is a structure that contains a large array. This warning informs you that additional arguments may result in an overflow.

123 WARNING Function *function_name* needs *number* bytes of stack, which exceeds the maximum stack size of *number* bytes.

You are passing arguments to this function that take up too much space. The sort of argument that could trigger this error is a structure or union that contains a large array. If the compiler issues this warning, then the resulting object file is non-executable.

124 ERROR Cannot add two pointers: *pointer_exp1 + pointer_exp2*.

C permits you to add an integer to a pointer, but you can never add a pointer to another pointer. Perhaps you were trying to add two dereferenced pointers and you did not use the proper syntax. In that case, please see the "pointer operations" listing in Chapter 4. Incidentally, don't forget that an array name is a pointer constant.

125 ERROR Illegal type of *token* for addition to a pointer.

If you add a value to a pointer, the value must be an integer. Apparently, you have added a value that is not an integer.

126 ERROR Cannot subtract two pointers of different type: *pointer_var1 - pointer_var2*.

For example, compare two possible pointer subtractions:

```
int i,  *pi = &i;
int j,  *pj = &j;
char c, *pc = &c;
i = pi - pc;  /* wrong -- pi points to int, but pc
                        points to char */
i = pi - pj   /* right -- both pi and pj point to
                        ints */
```

127 ERROR Either *expression1* is not a pointer type, or *expression2* is not an integer type.

Pointers and subtraction do not often mix. C permits you to subtract an integer value from a pointer; however, you cannot

- Subtract a non-integer value from a pointer.
- Subtract a pointer from any value.

For example, compare some proper and improper methods of subtraction:

```
int  *px;

(px - 2)       /* right */
(px - 2.2)     /* wrong */
(2 - px)       /* wrong */
(2.2 - px)     /* wrong */
```

128 ERROR "main" function cannot return a type whose size is greater than 4 bytes.

By default, the main function returns an **int**. You are trying to pass back something that cannot be converted to an **int**.

129 WARNING Ignoring data initialization for "switch" variable declarations.

A compound switch statement contains a block. Since C permits you to define variables on the block level, you can define a variable within the **switch** statement. However, if you try to initialize this variable, C ignores the value. In other words, the variable will spring into existence when the program enters the block, but the variable will have a garbage value.

131 ERROR Cannot take the address of register variable *variable_name*.

Since a program ideally stores a register variable in a register, and since a CPU register has no address, you cannot find the address of a register variable. For more information on the register storage class specifier, see Chapter 3.

132 WARNING Multiple declaration of *variable_name* with data initialization, previous declaration was on line *number*.

You'll trigger this warning if you declare two or more variables of the same name and at least one of them contains an initialization value. If two or more variables have the same name but different initialization values, then the compiler will set the variable's value to the last initialization value. For example, given the following two initializations

```
float s = 2.2;
float s = 4.2;
```

the compiler will initialize s to 4.2.

133 ERROR Compiler failure, unexpected data init construct: *construct*.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

134 ERROR Compiler failure, Pascal-only error code.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

135 ERROR Floating point constant *number* conversion problem.

For some reason (probably overflow), the compiler could not convert *number* to the desired data type.

136 ERROR Compiler failure, register consistency.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

137 ERROR Compiler failure, no temp created.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

138 ERROR Compiler failure, improper forward label at *token*.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

139 ERROR Compiler failure, pseudo pc consistency.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

140 ERROR Compiler failure, unknown tree node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

141 ERROR Compiler failure, unknown top node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

142 ERROR Compiler failure, no temp space.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

143 ERROR Compiler failure, lost value of node.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

144 ERROR Compiler failure, registers locked.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

145 ERROR Compiler failure, no emit inst.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

146 ERROR Compiler failure, procedure too large.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

147 ERROR Compiler failure, inst disp too large.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

148 ERROR `Compiler failure, obj module too large.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

149 ERROR `Compiler failure, no free space.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

150 ERROR `Compiler failure, short branch optimization.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

151 ERROR `Compiler failure, data frame overflow.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

152 ERROR `External variable definition` *identifier* `conflicts with procedure or data section name.`

You cannot declare a global variable having the same name as a procedure or data section name.

154 ERROR `Compiler failure, too many nodes.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

159 WARNING `Variable` *variable_name* `was not initialized before this use.`

You are using *variable_name* on the right side of an assignment operator, but you have not assigned *variable_name* a value yet, so using it may cause bizarre results.

160 ERROR `Illegal bit field constant` *identifier*`; cannot be negative.`

Bit fields must be positive integers. See Section 3.8.4 for details about bit fields.

161 ERROR Unknown or incomplete structure/union type name.

You mistakenly tried to declare a recursive structure or union. For example, the following declaration causes this error because *name* was not yet a declared data type when you attempted to use it:

```
struct S   {int x;
              struct S c;};  /* wrong */
```

Note that you can declare a pointer to this structure or union. For example, the following declaration is okay:

```
struct S {int x;
            struct S *c;};  /* right */
```

162 ERROR Illegal option *identifier* for typedef.

You are using the **#attribute address** modifier in a typedef statement. In a typedef statement, **#attribute address** is illegal; however, **#attribute volatile** and **#attribute device** are legal. For details about the **#attribute** modifier, see Chapter 3. Incidentally, by fixing this error, you will probably fix a lot of other errors.

163 ERROR Left bracket ([) expected; *token* found.

The C compiler expected a left bracket immediately after the **#attribute** modifier, but found *token* instead. For details about the **#attribute** modifier, see Chapter 3.

164 ERROR Right bracket (]) expected; *token* found.

The C compiler expects a right bracket just after the **#attribute** argument. For details about the **#attribute** modifier and its arguments, see Chapter 3.

165 ERROR Left parenthesis "(" expected; *token* found.

Possibly, you were using the **#attribute address** modifier, but you forgot to put an address (enclosed within parentheses) right after **address**. For details on **#attribute** address, see Chapter 3. Another possibility is that you forgot the left parenthesis in a **#section** preprocessor directive.

166 ERROR Right parenthesis ")" expected; *token* found.

Possibly, you were using the **#attribute device** modifier with the **read** or **write** options, but you forgot to close the list of read and write options with a right parenthesis. For example, compare the following declarations:

```
int q  #attribute[device(read)] = 2;  /* right */
int q  #attribute[device(read] = 2;   /* wrong,
                                      missing ")" */
```

If you correct this error, a lot of other errors will probably vanish. For details on **#attribute device**, see Chapter 3.

Another possibility is that you were using the **#section** preprocessor directive, but forgot to put a comma between the two section names.

167 ERROR Number expected; *token* found.

You misused the **#line** preprocessor directive. Compare the right and wrong ways to use **#line** in the following examples:

```
#23                  /* right */
#23 "new_file.c"     /* right */
#line 23             /* right */
#line 23 "new_file.c /* right */
#line "new_file.c"   /* wrong, triggers error 167 */
```

For details about its correct use, see the "#line" listing in Chapter 4.

169 ERROR String expected; *token* found.

You forgot to enclose the file name in double quotes while using the **#line** preprocessor directive. Compare the right and wrong ways to use **#line** in the following examples:

```
#23                   /* right */
#23 "new_file.c"      /* right */
#line 23              /* right */
#line 23 "new_file.c" /* right */
#23 new_file.c      /* wrong, triggers error 169. */
#line new_file.c    /* wrong, triggers error 169. */
```

170 ERROR Dividing by zero in a compiletime constant expression.

Division by zero is illegal.

171 ERROR `Compiler failure, store elimination failure.`

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

172 ERROR `No static address for dynamic variable` *variable_name*.

C issues this error when you assign the address of a dynamic variable to a static pointer. For example, consider the following statements:

```
auto int x = 3;
static *px = &x;
```

Since **px** is a static pointer variable, it cannot hold the address of the dynamic variable **x**. To correct the problem, make both variables dynamic or make both variables fixed. For an explanation of dynamic and static, see Chapter 3.

173 WARNING `Comma expected but not found in data init list.`

You must separate the elements of a data initialization list with commas; for example:

```
int x[] = {2,3,5,7};     /* okay */
int x[] = {2 3 5 7};     /* wrong */
```

174 ERROR `Empty structure or union.`

C prohibits you from declaring a structure or union without members. By fixing this error, you may indirectly also fix many other errors. For details on structures and unions, see Chapter 3.

175 ERROR `Unknown type name in "sizeof".`

The **sizeof** operator is evaluated at compile time, not run time. Therefore, if **sizeof**'s operand is a partially constructed type, then this error is triggered. For example, the following use of **sizeof** triggers this error because data type **x** is not fully constructed at the point when **sizeof** is called:

```
struct x {unsigned int   q : sizeof(struct x);};
```

176 ERROR `Too many nested pointer references for debug tables.`

You declared a structure or union having a member which is itself a structure or union, and one of the members of this structure or union is itself a structure or union, and so on, and so on, down 256 or more levels.

177 WARNING `8 or 9 found in an octal number.`

This is only a warning, but if you get it, your program will probably produce bizarre run-time results. As in the rules of conventional math, the digits 8 and 9 are forbidden in a base 8 number. Note that in C, an octal number is any integer that begins with the digit 0. Did you mistakenly put a leading 0 in your decimal number?

178 ERROR `Null dimension in a sub-array declaration.`

A multidimensional array cannot have any null dimensions other than the first dimension. For example, consider the following array declarations:

```
int  x[3][5];    /* right */
int  x[3][5][];  /* wrong */
int  x[];        /* right */
int  x[][3][5];  /* right */
```

179 ERROR `Invalid systype` *string*.

See the "#systype" or "if" listings in Chapter 4 for a list of valid systypes.

180 ERROR `Compiler failure, limit exceeded;` *identifier*.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

182 ERROR `Cannot give more than one "systype".`

You can put no more than one **#systype** preprocessor directive in a file. For details on **#systype**, see the "#systype" listing in Chapter 4.

183 ERROR `Cannot take "systype" once other tokens are seen.`

The only place that a **#systype** preprocessor directive can occur is as the first or the second token in the file. If it is the second token, then the only token that can precede it is the **#module** preprocessor directive. For details on **#systype**, see the "#systype" listing in Chapter 4.

184 ERROR `Comma expected,` *token* `found.`

Probably, you forgot a comma in a **#module** preprocessor directive. For example, compare the right and wrong ways to use **#module**:

```
#module  math,  x$, y$ /* right */
#module  math,  x$  y$ /* wrong, missing a comma */
```

185 ERROR Found "end-of-line" before end of definition.

You made a mistake in a preprocessor directive. Possibly, you forgot to close parentheses or quotes. See Chapter 4 for descriptions of all the preprocessor directives.

186 ERROR Redundant #module control line found; ignored.

A source file contains more than one **#module** preprocessor directive, but C allows one (at most) per file.

187 ERROR Procedure section name conflicts with a previously defined data section name or identifier.

Suppose you created a procedure section named "x" with the **#section** preprocessor directive. If, later in the same file you use **x** as a data section name, you will trigger this error. Also, if you define **x** as a global variable, C will issue this error because a global variable named **x** is stored in a data section named **x**. Probable cause for this error—you accidentally reversed the section names. See the "#section" listing in Chapter 4 for details.

188 ERROR Data section name conflicts with a previously defined procedure section name or identifier.

Suppose you created a data section named **x** with the **#section** preprocessor directive. If later in the same file you use **x** as a procedure section name, you will trigger this error. Also, if you define **x** as a global variable, C will also issue this error because a global variable named **x** is stored in a data section named **x**. Probable cause for this error—you accidentally reversed the section names. See the "#section" listing in Chapter 4 for details.

189 ERROR Extraneous data at end of control line; ignored.

A **#section** preprocessor directive should end with a right parenthesis). However, you have mistakenly put some more code after the right parenthesis. This extra code may cause many errors. See the "#section" listing in Chapter 4 for details.

190 ERROR Compiler failure, invalid use of multiple sections and non-local goto to label *identifier*.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

191 ERROR Compiler failure, bad address constant.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

192 ERROR Compiler failure, invalid use of multiple sections and up-level referencing in routine *identifier*.

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

193 ERROR Illegal option *token* for parameter.

You cannot use **#attribute address** in a parameter declaration, though **#attribute volatile** and **#attribute device** are okay. For details on the **#attribute** modifier, see Chapter 3.

194 ERROR Data section name conflicts with a previously defined external variable.

One of your global variables matches the name of a data section. (See the "#module" listing of Chapter 4 for details on section names.)

195 ERROR Cannot take "#module" directive once other tokens are seen.

The **#module** preprocessor directive is optional, but when you use it, it must be the very first thing in the file.

196 ERROR "#section" directive may not appear within a function.

A C function can appear outside a function, but can never appear within a function. See the "#section" listing in Chapter 4 for details.

197 WARNING Identifier exceeds 32 characters, only *identifier* is recognized.

Your source code can contain names of up to 256 characters; however, for internal representation, Domain C truncates any name longer than 32 characters down to 32 characters. Thus, the compiler sees the following two identifiers as identical even though we see them as unique:

```
int   accounts_receivable_kansas_city_kansas;
int   accounts_receivable_kansas_city_missouri;
```

198 WARNING Type of *variable_name* is illegal for member *token*.

Possibly, you are misusing the arrow operator ->. The arrow operator dereferences a pointer to a structure or union, so the compiler issues this warning if *variable_name* is not a pointer to a structure or union. Another possibility is that you misspelled the name of the structure or union. See the "structure and union operations" listing in Chapter 4 for details.

199 ERROR Non-unique member *name* requires struct/union or struct/union pointer.

You can trigger this rather rare error by using the same member name more than once in different structure or union declarations. For example, the following attempt to reference member **a** is doomed because the compiler cannot figure out whether you mean **j.a** or **k.a**:

```
struct  {int a;} j;
struct  {float a;} k;

int  *i;
i->a = 10;
```

200 ERROR Illegal return type for function *function_name*; functions must return either an lvalue or void.

Functions cannot return arrays, but see Chapter 5 for a way around this restriction.

201 ERROR Internal error - *error_message*

The error is in the compiler, not in your code. Please contact your customer support representative or mail us an APR.

202 WARNING Value assigned to *variable_name* is never used; assignment eliminated by optimizer.

You made an assignment to a local, automatic variable, but never used that variable again. To make the program more efficient, the optimizer eliminated the assignment.

203 ERROR Illegal declaration of *variable_name*; cannot have an array of functions.

You have attempted to declare an array of functions as in:

```
int  f[]();
```

This is not allowed in C. To declare an array of pointers to functions, which is legal, you can write:

```
int (*f[])()
```

204 WARNING Wrong size for enum *variable_name*; original size type_name assumed.

This warning occurs when you declare an enum type with the **char**, **short**, or **long** qualifiers, and then use the type without the qualifier. For example:

```
short enum color { green, red, blue };
enum color hue;
```

The declaration of **hue** will generate this warning because it does not include the **short** specifier.

205 WARNING Enumeration type clash [*variable_name, variable_name*] to the *operator* operator.

Technically, it is legal to mix enums of one type with enums of another type, and to mix enums with integer types. However, Domain C reports a warning when it encounters one of these type clashes.

206 WARNING Address of array or function in this context is redundant and ignored.

This warning occurs when you precede a naked array name (i.e., one without a subscript) or a naked function name (i.e., one without the parentheses indicating invocation) with an ampersand. Naked array names and function names are implicitly converted to addresses, so the address-of operator is ignored.

207 ERROR Illegal type "void" for argument *parameter_name*.

You have attempted to pass an argument of type **void**. Recall that the **void** type used in a prototype means that the function accepts *no* arguments, not that it accepts an argument of type **void**.

208 ERROR Illegal use of "void" in a function prototype; void must be the only type specified.

The type **void** in a prototype means that the function takes no arguments, so it is invalid to specify **void** *and* additional parameter types.

209 ERROR Illegal use of "ellipsis" in a function prototype; no other elements may follow "...".

The ellispsis notation in a function prototype can only appear as the last parameter. It indicates that the function accepts an unspecified number of additional arguments.

210 ERROR Exceeded maximum number of allowable parameters (> 64).

Domain C does not support functions that take more than 64 arguments.

211 ERROR Type of formal parameter parameter_name conflicts with prototype declaration.

This error occurs when the types declared in a prototype declaration do not match the parameter types in the function definition. For example:

```
extern int foo( int );

int foo( char x ) {};
```

212 INFO 1 Old-style function declaration encountered; default prototype *function_name*(...) assumed.

This informational message indicates an old-style function definition, such as:

```
extern int f(), g();
```

213 INFO 1 No prototype in scope, default prototype *function_name*(...) assumed.

When the compiler encounters a function invocation for a function that has not been prototyped, it assumes that the function returns an **int** and takes an unspecified number of arguments.

214 WARNING Cannot dereference "pointer to void" .

You may not dereference a pointer that is declared to point to the **void** type.

215 ERROR Missing parameter name for argument *argument_name* of *function_name*.

You have forgotten to enter a parameter name in a prototype definition.

216 INFO 1 Although argument *argument_name* to *function_name* is assignment compatible, it does not match the declared argument type.

You have invoked a function with arguments that are assignment-compatible with the parameters declared in the prototype, but are not exactly the same type. For example:

```
extern void foo( short, double );
int a;
float b;
   .
   .
foo( a, b );
```

a will be converted to **short**, and *b* to **double**, but you will receive info messages telling you that they types of *a* and *b* are not the same as the parameter types declared in the prototype.

217 ERROR Invalid #options specifier, *token*.

The only valid **#options** specifiers are **a0_return** and **d0_return**.

218 ERROR Illegal declaration of *variable_name*; array of references not allowed.

You have attempted to create an array of reference variables, which is not allowed.

219 ERROR Uninitialized reference variable *variable_name*.

You have declared a reference variable, but failed to initialize it. You must initialize all reference variables.

220 ERROR Global or static reference variable *variable_name*; not implemented.

Currently, Domain C does not support global or static reference variables. Reference variables must be local and automatic, or be function parameters.

221 WARNING Incompatible combination of integer and pointer types.

This warning occurs when you attempt to assign an integer value to a pointer type, or vice versa. Assignments such as these are not portable.

222 ERROR Invalid runtype *token*.

You have compiled with the **-runtype** switch, but have specified a runtype that the compiler does not recognize. See Chapter 6 for a list of valid runtypes.

223 INFO 3 Unnaturally aligned load/store *variable_name* diminishes code quality.

You have referenced an object that is not naturally aligned.

224 ERROR Compiler failure, no case for object type.

Internal failure. Submit APR.

225 ERROR Argument to *attribute_specifier* attribute conflicts with value already specified for this type.

You have attempted to assign an attribute specifier to an object that has already been declared with a conflicting specifier.

226 ERROR Maximum specifiable alignment is *alignment_value*.

You may not specify alignment greater than 3 (octword boundaries).

227 ERROR Size of "@1" bits is invalid for specified type.

Pascal error. Not used by C compiler.

228 ERROR Structured types may not be UNALIGNED.

You have specified byte alignment for a structure or union. The minimum alignment for structures and unions is word alignment.

229 WARNING Specified *attribute_specifier* attribute conflicts with attributes of base type.

This error occurs when you specify an attribute for an object of a user-specified type, and the type definition specifies a conflicting attribute. For example:

```
typedef struct {
                  int a;
                  short b;
                } S #attribute[natural];

S s1 #attribute[align(1)];
```

230 ERROR *Attribute_specifier* attribute is inappropriate for target machine type.

Reserved for future use.

231 ERROR *Attribute_specifier* and *attribute_specifier* attributes may not both be specified.

You have specified two attributes that are mutually exclusive.

232 ERROR PHYSICAL attribute specified without an ADDRESS.

You have specified the **physical** attribute for a variable, but have failed to specify an **address**. You must specify an **address** attribute when you specify a **physical** attribute. See Chapter 2 for more information about these attributes.

233 ERROR *Attribute* is inappropriate in this context.

You have specified an inappropriate attribute. For example, specifying **volatile** for a function parameter will generate this error.

234 INFO 1 Actual alignment of *variable_name* (*alignment*) is less than natural alignment (*natural_alignment*).

This info message tells you that you have declared an object or type that is not naturally aligned.

235 INFO 1 Large bit field *bit_field_name* not on longword boundary.

The bit field named *bit_field_name* is not aligned on a longword boundary.

236 WARNING Invalid section attribute for static var - ignored.

You have used the **#section** specifier to indicate a named overlay section, but the section name you specified is not valid.

237 ERROR This section name conflicts with a previously defined global variable.

You have attempted to create a section with the same name as a previously defined global variable. This is not allowed. For example:

```
int global_var = 1;
int x #attribute[section(global_var)];

main()
{
  ...
}
```

238 ERROR Bit field constant *bit_field_name* too large.

You may not declare a bit field larger than 32 bits.

239 INFO 1 Actual alignment of array elements *array_name* is less than natural alignment "@2".

This message occurs when you declare an array of structures where the size of the structure is not evenly divisible by the size of the largest member. For example:

```
typedef struct {
                  int a;
                  short b;
                } S;

S ar_of_S[10];
```

240 WARNING Alignment of array elements is dependent on the current default alignment environment.

This warning signifies that the alignment of array elements depends on the current alignment setting.

241 WARNING Size of array element rounded up from *num* to *num* bits.

Reserved for future use.

Appendix A
ISO Latin–1 Table

Domain C uses the ISO DIS 8859/1 character set, commonly known as Latin–1, for character data representation. The Latin–1 set also includes all ASCII characters in their standard positions. Table B–1 shows the decimal, octal, and hexadecimal values for all ISO Latin–1 characters.

You can use Latin–1 characters in comments or character strings, but are limited to using ASCII letters A–Z and a–z (decimal positions 65–90 and 97–122, respectively), digits, underscores (_), and dollar signs ($) in identifiers. This adheres to existing C standards.

Table A–1. ISO Latin–1 Codes

oct	dec	hex	character		oct	dec	hex	character
0	0	0	NUL	^@	40	32	20	space
1	1	1	SOH	^A	41	33	21	!
2	2	2	STX	^B	42	34	22	"
3	3	3	ETX	^C	43	35	23	#
4	4	4	EOT	^D	44	36	24	$
5	5	5	ENQ	^E	45	37	25	%
6	6	6	ACK	^F	46	38	26	&
7	7	7	BEL	^G	47	39	27	'
10	8	8	BS	^H	50	40	28	(
11	9	9	TAB	^I	51	41	29	)
12	10	A	LF	^J	52	42	2A	*
13	11	B	VT	^K	53	43	2B	+
14	12	C	FF	^L	54	44	2C	,
15	13	D	CR	^M	55	45	2D	-
16	14	E	SO	^N	56	46	2E	.
17	15	F	SI	^O	57	47	2F	/
20	16	10	DLE	^P	60	48	30	0
21	17	11	DC1	^Q	61	49	31	1
22	18	12	DC2	^R	62	50	32	2
23	19	13	DC3	^S	63	51	33	3
24	20	14	DC4	^T	64	52	34	4
25	21	15	NAK	^U	65	53	35	5
26	22	16	SYN	^V	66	54	36	6
27	23	17	ETB	^W	67	55	37	7
30	24	18	CAN	^X	70	56	38	8
31	25	19	EM	^Y	71	57	39	9
32	26	1A	SUB	^Z	72	58	3A	:
33	27	1B	ESC	^[	73	59	3B	;
34	28	1C	FS	^|	74	60	3C	<
35	29	1D	GS	^]	75	61	3D	=
36	30	1E	RS	^^	76	62	3E	>
37	31	1F	US	^_	77	63	3F	?

(Continued)

Table A-1. ISO Latin-1 Codes (Cont.)

oct	dec	hex	character	oct	dec	hex	character
100	64	40	@	140	96	60	'
101	65	41	A	141	97	61	a
102	66	42	B	142	98	62	b
103	67	43	C	143	99	63	c
104	68	44	D	144	100	64	d
105	69	45	E	145	101	65	e
106	70	46	F	146	102	66	f
107	71	47	G	147	103	67	g
110	72	48	H	150	104	68	h
111	73	49	I	151	105	69	i
112	74	4A	J	152	106	6A	j
113	75	4B	K	153	107	6B	k
114	76	4C	L	154	108	6C	l
115	77	4D	M	155	109	6D	m
116	78	4E	N	156	110	6E	n
117	79	4F	O	157	111	6F	o
120	80	50	P	160	112	70	p
121	81	51	Q	161	113	71	q
122	82	52	R	162	114	72	r
123	83	53	S	163	115	73	s
124	84	54	T	164	116	74	t
125	85	55	U	165	117	75	u
126	86	56	V	166	118	76	v
127	87	57	W	167	119	77	w
130	88	58	X	170	120	78	x
131	89	59	Y	171	121	79	y
132	90	5A	Z	172	122	7A	z
133	91	5B	[	173	123	7B	{
134	92	5C	\	174	124	7C	|
135	93	5D	]	175	125	7D	}
136	94	5E	^	176	126	7E	~
137	95	5F	_	177	127	7F	del

(Continued)

Table A-1. ISO Latin-1 Codes (Cont.)

oct	dec	hex	character	oct	dec	hex	character
204	132	84	IND	247	167	A7	§
205	133	85	NEL	250	168	A8	¨
206	134	86	SSA	251	169	A9	©
207	135	87	ESA	252	170	AA	ª
210	136	88	HTS	253	171	AB	«
211	137	89	HTJ	254	172	AC	¬
212	138	8A	VTS	255	173	AD	SHY
213	139	8B	PLD	256	174	AE	®
214	140	8C	PLU	257	175	AF	¯
215	141	8D	RI	260	176	B0	°
216	142	8E	SS2	261	177	B1	±
217	143	8F	SS3	262	178	B2	2
220	144	90	DCS	263	179	B3	3
221	145	91	PU1	264	180	B4	´
222	146	92	PU2	265	181	B5	µ
223	147	93	STS	266	182	B6	¶
224	148	94	CCH	267	183	B7	·
225	149	95	MW	270	184	B8	¸
226	150	96	SPA	271	185	B9	1
227	151	97	EPA	272	186	BA	º
233	155	9B	CSI	273	187	BB	»
234	156	9C	ST	274	188	BC	¼
235	157	9D	OSC	275	189	BD	½
236	158	9E	PM	276	190	BE	¾
237	159	9F	APC	277	191	BF	¿
240	160	A0	NBSP	300	192	C0	À
241	161	A1	¡	301	193	C1	Á
242	162	A2	¢	302	194	C2	Â
243	163	A3	£	303	195	C3	Ã
244	164	A4	¤	304	196	C4	Ä
245	165	A5	¥	305	197	C5	Å
246	166	A6	¦	306	198	C6	Æ

(Continued)

Table A–1. ISO Latin–1 Codes (Cont.)

oct	dec	hex	character	oct	dec	hex	character
307	199	C7	Ç	347	231	E7	ç
310	200	C8	È	350	232	E8	è
311	201	C9	É	351	233	E9	é
312	202	CA	Ê	352	234	EA	ê
313	203	CB	Ë	353	235	EB	ë
314	204	CC	Ì	354	236	EC	ì
315	205	CD	Í	355	237	ED	í
316	206	CE	Î	356	238	EE	î
317	207	CF	Ï	357	239	EF	ï
320	208	D0	Ð	360	240	F0	ð
321	209	D1	Ñ	361	241	F1	ñ
322	210	D2	Ò	362	242	F2	ò
323	211	D3	Ó	363	243	F3	ó
324	212	D4	Ô	364	244	F4	ô
325	213	D5	Õ	365	245	F5	õ
326	214	D6	Ö	366	246	F6	ö
327	215	D7	×	367	247	F7	÷
330	216	D8	Ø	370	248	F8	ø
331	217	D9	Ù	371	249	F9	ù
332	218	DA	Ú	372	250	FA	ú
333	219	DB	Û	373	251	FB	û
334	220	DC	Ü	374	252	FC	ü
335	221	DD	Ý	375	253	FD	ý
336	222	DE	Þ	376	254	FE	þ
337	223	DF	ß	377	255	FF	ÿ
340	224	E0	à				
341	225	E1	á				
342	226	E2	â				
343	227	E3	ã				
344	228	E4	ä				
345	229	E5	å				
346	230	E6	æ				

Appendix B

Domain C Extensions

This appendix lists all extensions to the *de facto* C standard defined in *The C Programming Language* by Kernighan and Ritchie. The extensions listed in Table B–1 are compatible with the proposed ANSI C standard or with the C++ programming language. The ones listed in Table B–2 are unique to Domain C.

Table B–1. ANSI C and C++ Extensions Supported by Domain C

Extension	ANSI	C++
function prototypes	✓	✓
reference variables		✓
void and (**void** *)	✓	✓
__FILE__ and **__LINE__** predefined symbols	✓	✓
__DATE__, **__TIME__** and **__STDC__** predefined symbols	✓	
structure and union assignment	✓	✓
union initialization	✓	✓
defined preprocessor operator	✓	✓
unsigned short, **unsigned long**, and **unsigned char** types	✓	✓
long constants	✓	✓
passing structures and unions as arguments	✓	✓

Table B–2. Domain Extensions to the C Language

Extension
sized **enums** (**char, short,** and **long**)
#attribute specifier
#option specifier
std_$call keyword
#section and **#module** preprocessor directives
#debug preprocessor directive
#eject preprocessor directive
#list and **#nolist** preprocessor directives
#systype preprocessor directive
systype predefined macro
_BFMT__COFF predefined name
long float data type
dollar sign ($) in identifiers
partial specification of struct and union members

Appendix C

BSD lint: A C Program Checker

C.1 Introduction

The **lint** utility examines C source code, detecting any bugs or obscurities. It enforces the type rules of C more strictly than the C compilers do. It may also be used to enforce many portability restrictions involved in moving programs between different machines and/or operating systems. Furthermore, it detects certain constructions which, although technically "legal," are nonetheless wasteful, error-prone, or otherwise best avoided. **lint** accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between **lint** and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible, in part, because the compilers don't do sophisticated type checking, especially between separately-compiled programs. **lint** takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This chapter discusses the use of **lint**, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

C.2 Summary of lint Options

The command currently has the form

% **lint** [*options*] *files... library-descriptors...*

The following options are available:

-a	Print messages about assignments of **long** objects to integers that are not **long**.
-b	Print messages about unreachable **break** statements.
-c	Complain about questionable casts.
-h	Perform heuristic checks.
-n	Don't do any library checking.
-p	Perform portability checks.
-s	Perform heuristic checks (same as **h**).
-u	Don't report unused or undefined externals.
-v	Don't report unused arguments.
-x	Report unused external declarations.

C.2.1 Usage

Suppose there are two C source files, **file1.c** and **file2.c**, that are ordinarily compiled and loaded together. Then the command,

```
$ lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The following command

```
$ lint -p file1.c file2.c
```

also produces these messages, as well as other messages that relate to the "portability" of the programs to other operating systems and machines. Replacing the **-p** by **-h** produces messages about constructions that, although legal, demonstrate poor programming style (according to **lint**). You may use both options

```
$ lint -hp file1.c file2.c
```

to get both types of messages.

Many of the facts that **lint** needs to establish may, in reality, be impossible to discover. For example, it may not be possible to know whether a given function in a program ever gets called without also knowing the input data. Deciding whether **exit** is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the **lint** algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, **lint** assumes it can be called.

lint tries to give only relevant information. Messages of the form "*xxx* might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, **lint** loses credibility, and its "error" messages merely clutter up the output, obscuring other, possibly more important messages.

C.2.2 Unused Variables and Functions

As sets of programs evolve, previously used variables and arguments to functions may become unused. It isn't uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally help you to discover bugs; if a function does a necessary job and is never called, something is probably wrong.

lint complains about variables and functions that are defined but not otherwise mentioned. An exception is variables that are declared through explicit **extern** statements but are never referenced; thus, the statement

```
extern float sin();
```

evokes no comment if **sin** is never used. Note that this agrees with the semantics of the Domain C compiler. In some cases, these unused external declarations might be of some interest; you can discover them by adding the **-x** option when you invoke **lint**.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option suppresses the printing of complaints about unused arguments. When **-v** is in effect, **lint** produces no messages about unused arguments except for those arguments that are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

In one particular case, information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some, but not all, files in a collection that is normally loaded together. Here, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. Use the **-u** option to suppress the spurious messages that might otherwise appear.

C.2.3 Set/Used Information

lint attempts to detect cases where a variable is used before it is assigned a value. This isn't easy to detect. Many algorithms take a good deal of time and space, and still produce "error" messages about perfectly valid programs. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur later, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. This genre of complaint has its roots in stylistic, rather than actual, error. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables used in the expression that first sets them.

The set/used information also permits recognition of those local variables that are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

C.2.4 Flow of Control

lint attempts to detect unreachable portions of the programs that it processes. It complains about unlabeled statements immediately following **goto, break, continue,** or **return** statements. It attempts to detect loops that can never be left at the bottom, detecting the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also complains about loops that can't be entered at the top. As is often true when **lint** makes false accusations, this condition may not be a bug, but a complaint about programming style.

lint has an important area of blindness in the flow of control algorithm: it can't detect functions that are called and never return. Thus, a call to **exit** may cause unreachable code that **lint** doesn't detect; the most serious effects of this are in the determination of returned function values (see Section C.2.5).

A **break** statement that can't be reached causes no message. Programs generated by **yacc** and **lex** may have hundreds of unreachable **break** statements. The **-O** option in the C compiler often eliminates the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing you can do about them, and the resulting messages would clutter up **lint**'s output. If you want to see these messages, invoke **lint** with the **-b** option.

C.2.5 Function Values

Sometimes functions return values that are never used; sometimes programs incorrectly use function "values" that have never been returned. **lint** addresses this problem in a number of ways. Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; **lint** gives the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. For example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if **a** tests false, **f** calls **g** and then returns with no defined return value; this triggers a complaint from **lint**. If **g**, like **exit**, never returns, the message is produced even though nothing is actually wrong. In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial fraction of the "noise" messages produced by **lint**.

On a global scale, **lint** detects cases where a function returns a value, but this value is sometimes or always unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., no testing for error conditions).

The dual problem of using a function value when the function does not return one is also detected. This is a serious problem that has been observed in "working" programs where, by chance, the desired function value was computed in the function return register.

C.2.6 Type Checking

lint enforces the type checking rules of C more strictly than compilers do. The additional checking goes on in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

Several operators have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the "—>" be a pointer to structure, the left operand of the "." be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, **lint** checks to see that enumeration variables or members are not mixed with other types or other enumerations. Another check ensures that the only operations applied are =, initialization, ==, !=, and function arguments and return.

C.2.7 Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider this assignment, where **p** is a character pointer:

```
p = 1 ;
```

lint has reason to complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. This assignment clearly signals the desired action. It seems harsh for **lint** to continue to complain about this. On the other hand, if this code is to be truly portable, such constructs should be examined carefully. The **-c** option controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

C.2.8 Nonportable Character Use

On most C implementations, characters take on only positive values. **lint** flags certain comparisons and assignments as illegal or nonportable. For example, the fragment

```
char c;
     ...
if( (c = getchar()) < 0 ) ....
```

works where the version of C allows a character to have a negative value, but fails on machines where characters always assume positive values. The real solution is to declare **c** an integer, since **getchar** is actually returning integer values. In any case, **lint** responds with "nonportable character comparison."

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because, on some machines, bitfields are considered signed quantities. While it may seem unintuitive to consider that a 2-bit field declared as type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

C.2.9 Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy in some implementations. This may happen in programs that have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** option.

C.2.10 Unorthodox Constructions

lint flags several perfectly legal, but somewhat unorthodox, constructions in the hope of promoting better code quality and clearer style, and even of pointing out bugs. The **-h** option enables these checks. For example, in the statement

```
*p++ ;
```

the asterisk (*) does nothing. This provokes the message "null effect" from **lint**. In the following program fragment,

```
unsigned x ;
if( x < 0 ) ...
```

the test never succeeds. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **lint** accuses you of making a "degenerate unsigned comparison" in these cases. If the code says

```
if( 1 != 0 ) ....
```

lint reports "constant in conditional context," since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs arising from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably don't do what was intended. The best solution is to place such expressions in parentheses, and **lint** encourages this by an appropriate message.

Finally, when the **-h** option is in force, **lint** complains about variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many to be bad style, often unnecessary, and frequently a bug.

C.2.11 Antiquated Syntax

lint attempts to discourage several forms of older syntax. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . .) could cause ambiguous expressions, such as

```
a =-1 ;
```

This expression could be interpreted as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

It is especially perplexing when such ambiguity arises as the result of a macro substitution. The newer and preferred operators (+=, -=, etc.) don't cause such confusion. To spur the abandonment of the older forms, **lint** complains about these older operators.

A similar issue arises with initialization. Older versions of C allowed

```
int x 1 ;
```

to initialize **x** to 1. This also caused syntactic difficulties. For example,

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { . . .
```

and the compiler must read some distance past **x** to be sure what the declaration really is. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equal sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

C.2.12 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. On machines where double–precision values may begin on any integer boundary, it is reasonable to assign integer pointers to double pointers. On other machines, double–precision values must begin on even word boundaries; thus, not all such assignments make sense. **lint** tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the **–p** or **–h** options are in effect.

C.2.13 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards,

function arguments are probably be best evaluated from right–to–left; on machines with a stack running forward, left–to–right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

So that the efficiency of C on a particular machine isn't unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++] ;
```

draws the complaint:

```
warning: i evaluation order undefined
```

C.3 Implementation Details

lint consists of two programs and a driver. The first program is a version of the Portable C Compiler (PCC). This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator (as the other compilers do), **lint** produces an intermediate file which consists of lines of ASCII text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring together all information collected about a given external name. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of **lint**.

C.3.1 Portability

This section describes some of the differences between C implementations, and discusses the **lint** features that encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The loader resolves these declarations and cause only a single word of storage to be set aside for *a*. Under some implementations, this isn't feasible, so each such declaration causes a word of storage to be set aside and called **a**. When loading or library editing takes place, this causes fatal conflicts that prevent the proper operation of the program. If **lint** is invoked with the **-p** option, it detects such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. Names known externally to UNIX software have seven significant characters, with the upper/lowercase distinction preserved. On other systems, the number of characters used and the preservation of case distinction may not be handled the same way. This leads to situations where programs that run fine under the UNIX system encounter loader problems on other systems. **lint -p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling. The UNIX system uses 8-bit ASCII. Other systems may use other character lengths or even other encoding schemes (e.g., EBCDIC). Moreover, character strings go from high to low bit positions ("left to right") on some systems, and low to high ("right to left") on the others. Thus, code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, are suspect. **lint** is of little help here, except to flag multi-character character constants.

Other problems are likely to arise in shifting or masking words. C supports a bit-field facility that can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. For example, consider the use of

```
x &= 0177700 ;
```

to clear the low order six bits of **x**. If the bit field feature cannot be used, the same effect can be obtained by writing the following, which works on many machines:

```
x &=  9~ 8 077 ;
```

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed unsigned. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware that has infiltrated itself into the C language. If there were a good way to discover the programs that would be affected, C could be changed; in any case, **lint** is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of

the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, **lint** has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

C.3.2 Suppressing Unwanted Output

Sometimes you want **lint** to refrain from citing various constructs that, while technically "wrong," are nevertheless there for a good reason. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of controlling **lint**'s output is often desirable.

The form that this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause several words to be recognized by **lint** when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, **lint** directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the **lint** directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to **lint**, it can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if you want to turn off strict type checking for the next expression, you can use the directive

```
/* NOSTRICT */
```

This causes the program to revert to the previous default after the next expression. The **-v** option can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by using this directive

```
/* VARARGS */
```

before the function definition. Sometimes, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments to be checked; thus, this causes the first two arguments to be checked, the others unchecked:

```
/* VARARGS2 */
```

Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file (see Section 6.3.3).

C.3.3 Library Declaration Files

lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. You can use the VARARGS and ARGSUSED directives to specify features of the library functions.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions defined on a library file, but not used on a source file, draw no complaints. **lint** doesn't simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** option is in effect, another file containing descriptions of the standard I/O library routines that are expected to be portable across various machines is checked. The **-n** option can be used to suppress all library checking.

Appendix D

SysV lint Utility

The **lint** program examines C language source programs, detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error-prone constructions, which nevertheless are legal. **lint** accepts multiple input files and library specifications and checks them for consistency.

D.1 Usage

The **lint** command has the form:

lint [*options*] *files* ... [*library-descriptors* ...]

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

-a	Suppress messages about assignments of long values to variables that are not long.
-b	Suppress messages about break statements that cannot be reached.
-c	Only check for intra-file bugs; leave external information in files suffixed with **.ln**.
-h	Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
-n	Do not check for compatibility with either the standard or the portable **lint** library.
-o *name*	Create a **lint** library from input files named **llib-***lname***.ln**.
-p	Attempt to check portability.
-u	Suppress messages about function and external variables used and not defined or defined and not used.
-v	Suppress messages about unused arguments in functions.
-x	Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as **-ab** or **-xha**. The names of files that contain C language programs should end with the suffix **.c**, which is mandatory for **lint** and the C compiler.

lint accepts certain arguments, such as:

```
-lm
```

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The **VARARGS** and **ARGSUSED** comments can be used to specify features of the library functions. Section D.2 describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **-p** option is used, another file is checked containing descriptions of the standard library routines that are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

D.2 lint Message Types

The following paragraphs describe the major categories of messages printed by **lint.**

D.2.1 Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages about variables and functions which are defined but not otherwise mentioned, unless the message is suppressed by means of the **-u** or **-x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks, depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the **-v** option is available to suppress the printing of these messages. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as **register** arguments. This can be considered an active (and preventable) waste of the register resources of the machine. Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the source code before the function. This has the effect of the **-v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked. When **lint** is applied to some but not all files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **-u** option suppresses the spurious messages.

D.2.2 Set/Used Information

lint attempts to detect cases where a variable is used before it is assigned a value. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use" since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

D.2.3 Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while**(1) and **for**(;;) as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, use the **-b** option.

lint has no way of detecting functions that are called and never return. Thus, a call to exit may cause unreachable code which **lint** does not detect. The most serious effects of

this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the **-b** option when dealing with such input.

D.2.4 Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return(  expr  );
```

and

```
return ;
```

statements is cause for alarm; **lint** will give the message

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a )
{
  if ( a ) return ( 3 );
   g ();
}
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced when in fact nothing is wrong. A comment

```
/*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature. On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type **void**. For example:

```
void fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions). The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

D.2.5 Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure selection operators
- between the definition and uses of functions
- in the use of enumerations

There are several operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a return statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*s can, of course, be intermixed with pointers to **x**'s.

The type checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the "." be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

D.2.6 Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled these intentions. Nevertheless, **lint** will continue to print messages about this.

D.2.7 Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare **c** as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a 2-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

D.2.8 Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs that have been incompletely converted to use typedefs. When a typedef variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **int**s, which are truncated. The **-a** option can be used to suppress messages about the assignment of **long**s to **int**s.

D.2.9 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages encourage better code quality, clearer style, and may even point out bugs. The **-h** option is used to suppress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message

```
null effect
```

from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **lint** will print the message

```
degenerate unsigned comparison
```

in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

and

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

D.2.10 Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization. The older forms of assignment operators (e.g., =+, =-, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past **x** in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equal sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

D.2.11 Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

```
possible pointer alignment problem
```

results from this situation.

D.2.12 Multiple Subexpressions and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

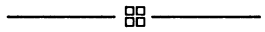
In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined. **lint** checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.



Appendix E

Using std_$call

This chapter describes how to use **std_$call** to invoke Pascal and FORTRAN routines from a C program. The **std_$call** convention is obsolete and will not be supported in future releases of the operating system. This documentation, therefore, is designed to enable you to maintain old programs. Do not use **std_$call** for new programs. Moreover, we strongly recommend that you remove **std_$call** from your existing programs as soon as possible. See Chapter 7 for information about current cross-language communication techniques.

E.1 Data Type Agreement of Arguments

When you call a C function in the absence of a prototype, the compiler automatically converts the data types of the parameters according to the rules shown in Table E-1. All of these conversions are suppressed, however, if you declare the function with a function prototype.

Table E-1. C Function Argument Conversions Without Prototype

Data Type of Argument	Data Type Actually Passed
char	int
short	int
unsigned char	unsigned int
unsigned short	unsigned int
float	double

E.2 Data Types of Constant Arguments

If you pass a constant to a Pascal or FORTRAN routine, you must make sure that the constant is the same size as the parameter declared in the Pascal or FORTRAN routine. The sections below describe the default sizes for constant expressions. For the sake of clarity, we recommend that you explicitly cast all constant expressions used as arguments to a **std_$call** routine even when the casting is not necessary. This does not produce any extraneous machine code.

> **NOTE:** When you pass a constant or constant expression, the value is stored in read-only memory. Therefore, you cannot attempt to change the parameter's value in the called routine. This applies to all FORTRAN parameters and any Pascal parameters declared with **VAR**, **OUT**, or **IN OUT**.

E.2.1 Integer Constants

Normally in C, all integral parameters are passed as 32-bit integers. For **std_$call** invocations, however, the default is 16 bits. That's because most of the Domain system calling sequences require 16-bit integers rather than 32-bit integers. The only times a constant expression is passed as 32 bits are when the value is too large to fit in 16 bits (i.e., if it is less than -32768 or greater than +32767), when it is explicitly cast to **int** or **long**, or when it has an "L" or "l" suffix. For instance, the following examples illustrate how different integer constants are passed to a **std_$call** routine.

Constant Expression	Type Passed
100	short
100*1000	long
-25	short
-25L	long
(long)25	long

E.2.2 Floating-Point Constants

All floating-point constants are represented as **double**. Therefore, they will agree in size with Pascal's **DOUBLE** and FORTRAN's **REAL*8** data types. To pass a 4-byte floating-point constant, you must cast it to **float**.

E.2.3 Character Constants

C treats character constants like integer constants. Since they are always within the range 0 through 128, they are passed as 16-bit values. To pass a character constant as a character, you must cast it to **char**.

E.2.4 String Constants

C passes string constants as arrays of type **char**.

E.3 Data Type Agreement of Function Declarations

Just as the parameters must agree in type, so must the function itself. For example, if a Pascal function returns an **INTEGER16** value, you must declare it in your C program as a **short** function. That is, if the Pascal declaration is

```
FUNCTION func1 (invar: DOUBLE) : INTEGER16;
```

then the C declaration should be:

```
std_$call short func1();
```

All C declarations of Pascal procedures and FORTRAN subroutines should use the **void** type since these routines do not return a value. For instance, the Pascal procedure defined by

```
PROCEDURE proc1(invar : DOUBLE);
```

should be declared as:

```
std_$call void proc1();
```

E.3.1 Functions Returning Pointers

In most cases, C treats pointers and integers interchangeably. For **std_$call** functions returning pointers, however, they are *not* the same. When Pascal returns the value of a function, it places it in one of two registers: a data register or an address register. Although C normally expects values to be returned in a data register, it conforms to the Pascal convention for **std_$call** functions. For instance, in the following example, C expects the value of **pass_point()** to be returned in an address register, and the value of **pass_data()** to be returned in a data register.

```
main()
{
    std_$call int *pass_point();
    std_$call int pass_data();
            .
            .
            .
```

Make sure that when you declare a **std_$call** function returning a pointer that you also declare it in Pascal as a function returning a pointer. Otherwise, the returned value will be put in one register while the C program is looking for it in a different register.

FORTRAN has no syntax for declaring a function that returns a pointer. All FORTRAN functions return their values in a data register. In C, you should never declare a **std_$call** FORTRAN function that returns a pointer.

E.3.2 Using std_$call

Pascal and FORTRAN usually pass arguments by reference; C generally passes arguments by value. To simplify cross-language communication, the Domain system uses a **standard calling convention.** In C, you signify that you are using the standard calling convention by declaring external Pascal and FORTRAN routines with the keyword **std_$call** before invoking them. This keyword tells the compiler that the C program will pass arguments according to the Domain system's standard calling convention.

The syntax for **std_$call** is:

std_$call *function-declaration*

For instance, all of the following are legal uses of **std_$call**:

```
std_$call void string_match();
std_$call int sum();
std_$call char *sp();
```

NOTE: Do not use the storage class **extern** in a **std_$call** declaration.

The **std_$call** declaration has the following effects on function calls:

- All arguments in the function call are passed by *reference* rather than by *value*. Essentially, the compiler adds an address-of operator (&) to every argument in the function call.
- All normal C argument conversions are suppressed.
- Integral constant expressions are passed as 16-bit values if they are in the range -32768 through +32767; otherwise, they are passed as 32-bit values.

The standard calling convention has some important consequences that are discussed in detail in this chapter. There are three general caveats that deserve special attention:

- **Passing Arrays**—When a FORTRAN or Pascal routine expects an array as an argument, *you must pass it an array reference: either an array name or a dereferenced pointer.* If you pass a pointer without dereferencing it, you will pass the address of the pointer. See Sections 7.5.3 and 7.6.4 for more information.

- **Passing Integer Constants**—Normally all integer arguments in C are expanded to 32 bits before they are passed. For **std_$call** functions, however, *constant integer expressions are passed as 16 bits if they can fit in 16 bits.* If they cannot fit in 16 bits, they are passed as 32 bits.

- **Passing Constants and Expressions**—*Do not pass a constant or an expression to a Pascal or FORTRAN routine that attempts to change the value of the incoming argument.* If you do, you will get an error or unpredictable results. The one exception to this rule occurs when you declare a Pascal parameter without a declaration keyword. In this case, Pascal generates a local copy of the argument that can be changed.

The program below shows two ways to call a Pascal routine named **pass_example()**, first by declaring it with **std_$call** and then by declaring it as a normal external routine and explicitly compensating for the different calling conventions.

```
static float x=1.0 , y=1.0;
void pass1()
{
    std_$call void pass_example();
    pass_example(x,y);   /* x and y are passed by reference */
                         /* because it is a std_$call function. */
}

void pass2()
{
    extern void pass_example();
    pass_example(&x,&y);  /* x and y are explicitly passed by
reference. */
}
```

In addition to the pass-by-reference and pass-by-value conventions, there are also complications created by the different data types supported by C, Pascal, and FORTRAN. The following sections describe the intricacies of data type agreement.

E.4 Pascal Examples

The following examples show how to pass various objects of different types and sizes to Pascal routines. In our examples, we always cast constant arguments even if the cast is unnecessary.

In Pascal, there are five ways to declare a formal parameter: **IN**, **OUT**, **IN OUT**, **VAR**, or without a keyword. In all five cases, the parameters are passed by reference, but the Pascal

keywords control what operations are legal within the Pascal routine and whether or not a local copy of the parameter is generated. The only time a local copy is generated is when the parameter is declared with no keyword. In this case, the Pascal routine can change the value of the argument without affecting the argument in the calling C program. Whenever one of the parameter keywords is used, however, the Pascal argument and the corresponding C argument have the *same address*.

E.4.1 Passing Integers

Suppose you need to send a 16-bit integer and a 32-bit integer to a Pascal routine that returns a 32-bit integer. In our example, the Pascal function squares the second argument, multiplies the result to the first argument, and returns the product.

```
MODULE pass_int_p;
FUNCTION pass_int(invar1 : INTEGER16 ;
                  invar2 : INTEGER32 ) : INTEGER32;
    BEGIN
        invar2 := sqr(invar2);
        pass_int := invar1*invar2;
    END;
```

The C program below shows a variety of ways to call this routine. Note especially that constants that can fit in 16 bits are passed as **shorts** unless you cast them. Non-constant expressions are passed as **int**s unless you cast them to **short**.

```
main()
{
    std_$call int pass_int();
    short arg1;              /* arg1 is 16 bits */
    int arg2,answer;         /* arg2 and answer are 32 bits */

/* Initialize variables. */
    arg1=10;
    arg2=20;

/* First we call pass_int with the correct-size arguments.
 * No casting is necessary.
 */
      answer = pass_int( arg1, arg2);
      printf("%d\t%d\t%d\n",arg1,arg2,answer);

/* If we want to send arg2 as the first argument and arg1 as
 * the second, both arguments must be cast.
 */
      answer = pass_int ( (short) arg2 , (long) arg1);
      printf("%d\t%d\t%d\n",arg2,arg1,answer);

/* Any integer expression containing a variable is converted to
 * long int.
 */
    answer = pass_int((short) (arg1+arg2) , (arg2-arg1));
    printf("%d\t%d\t%d\n",(arg1+arg2),(arg2-arg1),answer);

/* By default, integral constant expressions are passed as
 * shorts. Both constants are short because they are in the
 * range: -32767 - 32767.
 */
    answer = pass_int( (short) 10 , (long) (20*3));
    printf("%d\t%d\t%d\n",10,60,answer);

/* Append L to constant to make it long. */
    answer = pass_int( (short) 5 , 3L);
    printf("%d\t%d\t%d\n",5,3,answer);

/* Chars may be sent as integer values, but they are 16 bits.
*/
    answer = pass_int( (short) 'A' , (long) 'B');
    printf("%d\t%d\t%d\n",'A','B',answer);

}
```

If we execute the preceding program, the output is:

```
10      20      4000
20      10      2000
30      10      3000
10      60      36000
5       3       45
65      66      283140
```

E.4.2 Passing Floating-Point Numbers

The rules for passing floating-point numbers are similar to the rules for passing integers. All arguments must agree in type, either by declaration or by casting. By default, all floating-point expressions are passed as **double** values unless explicitly cast to **float**.

The example below shows a Pascal procedure that accepts two floating-point numbers; the first is four bytes and the second is eight bytes. The program assigns the second argument to be equal to the square root of the first argument. Note that this is not a function—the value is returned through the second argument, which is declared with **VAR**.

```
MODULE pass_float_p;

    PROCEDURE pass_float (single_var : SINGLE;
                          VAR double_var : DOUBLE);

    BEGIN
       double_var := sqrt(single_var);
    END;
```

The C program below shows various ways to call **pass_float()**. Because **double_var** is declared with **VAR** and has its value changed in the Pascal routine, it is illegal to pass it a constant or an expression.

```
#module pass_float_c
main()
{
    std_$call void pass_float();
    float arg1=0.0;
    double arg2=0.0;

/* First we pass variables of the correct size.  No casting is
 * needed.
 */

    printf("number\t\t\tsquare root\n\n");
    while (arg1<=2.0) {
       pass_float(arg1,arg2);
       printf("%f\t\t%f\n",arg1,arg2);
       arg1 += .25;
    }

/* Expressions that contain at least one floating-point
 * variable are converted to double before being passed.
 */

    pass_float((float)(arg1+3),arg2);
    printf("%f\t\t%f\n",arg1+3,arg2);

/* Floating-point constants must be cast or they will be passed
 * as doubles.
 */

    pass_float((float) 2e+3 , arg2);
    printf("%f\t\t%f\n",(float) 2e+3,arg2);

}
```

The output is:

```
number          square root

0.000000        0.000000
0.250000        0.500000
0.500000        0.707107
0.750000        0.866025
1.000000        1.000000
1.250000        1.118034
1.500000        1.224745
1.750000        1.322876
2.000000        1.414214
5.250000        2.291288
2000.000000     44.721359
```

E.4.3 Passing Character Data

When a Pascal routine expects a Pascal **CHAR** type, make sure that the argument you supply is passed as a reference to eight bits, not as a reference to 16 or 32 bits. You can do this either by passing a **C char** variable or by casting the argument to **char**. Be especially careful of **char** constants because for **std_$call** functions, all integral constant arguments, including character constants, are passed as 2-byte integers if possible.

Consider the following Pascal case-inversion routine that accepts a character as an argument and returns the same character with the opposite case.

```
MODULE pass_char_p;

    FUNCTION upper_lower (in_char : CHAR) : CHAR;
    BEGIN
        If ((ord(in_char) < 65) OR (ord(in_char) > 122) OR
            ((ord(in_char) >= 91) AND (ord(in_char) <= 96)))
                THEN upper_lower := in_char
        ELSE IF (ord(in_char) <= 97)
          THEN upper_lower := chr(ord(in_char) + 32)
        ELSE
          upper_lower := chr(ord(in_char) -32);
    END;
```

The C program below calls **upper_lower()** in a variety of ways.

```
#module pass_char_c
main()
{
      std_$call char upper_lower();
      char out_char, result;          /* 8-bit variables */
      short long_char, long_result;   /* 16-bit variables */

      out_char = 'A';
      long_char = 'B';
      printf("Original Char\t\tCase-Inverted\n\n");

/* We do not have to cast out_char because it is one byte. */

      result = upper_lower(out_char);
      printf("\t%c\t\t\t\t\t%c\n", out_char,result);

/* We must cast long_char because it is two bytes. */

      long_result = upper_lower((char) long_char)
      printf("\t%c\t\t\t\t\t%c\n",(char)long_char,long_result);

/* This is the right way to pass a character constant. */

      result = upper_lower((char) 'c');
      printf("\t%c\t\t\t\t\t%c\n",'c',result);

/* We can send integers if they can be represented in one byte.
*/

      result = upper_lower((char) 81);
      printf("\t%c\t\t\t\t\t%c\n",(char)81,result);

/* IF WE PASS A CONSTANT WITHOUT CASTING IT, IT WON'T WORK. */

      result = upper_lower('c');
      printf("\t%c\t\t\t\t\t%c\n",'c', result);

}
```

The result of this program execution is:

```
Original Char      Case-Inverted

    A                  a
    B                  b
    c                  C
    Q                  q
    c
```

Note that when we try to pass the constant 'c' without casting it, the result is an unprintable character.

E.4.4 Passing Character Arrays

Suppose you are calling a Pascal procedure that expects an array of **char** and the length of the string in the array. The Pascal program in our example takes two arguments: a string and the length of the string. It reverses the string and returns a pointer to the reversed string.

```
MODULE pass_string_p;
TYPE
   GENERIC_STRING = ARRAY [1..256] OF CHAR;
   STRING_POINT = ^GENERIC_STRING;

FUNCTION reverse_string (IN str : UNIV GENERIC_STRING;
                         IN len : INTEGER16) : STRING_POINT;
     VAR
         length : INTEGER16;
         temp : CHAR;
         temp_str : STATIC GENERIC_STRING;
     BEGIN
         length := len;
       WHILE length > len/2 DO
         BEGIN
           temp := str[len-length+1];
           temp_str[len-length+1] := str[length];
           temp_str[length] := temp;
           length := length-1;
         END;
       temp_str[len+1] := CHR(0);
       reverse_string := addr(temp_str);
     END;
```

The standard call declaration and some invocations appear below. Note that when Pascal expects an array argument, you must pass it either the name of the array or a dereferenced pointer to an array. For **std_$call** invocations, an array name and a pointer are *not* the same.

```
#module pass_string_c
main()
{
  std_$call char *reverse_string();
  char *sp="This is an example";

  short len=0;  /* C's short is equivalent to Pascal's
                 *INTEGER16 */

/* A "real" array of char!! */
  static char an_array[128]="This is the second example";

   len = strlen(sp);  /* strlen() returns a 32-bit length which
                       * is then converted to a short int. */

/* Notice that we must DEREFERENCE the pointer "sp", to make a
 * true 'array-type' expression.   Don't give "sp" by itself as
 * an argument, as you would in normal C; you'll only send the
 * ADDRESS of "sp"!
 */

   printf("%s\n",sp);
   sp = reverse_string(*sp,len);
   printf("%s\n\n",sp);


/*     reverse_string(sp, len);
WRONG!!!!!!!!!!!!!!!!!!!!! */

/* You could return the value from "strlen" directly, but then
 * you must cast it to short since the value returned is an
 * int. This next call returns the string back to the original.
 */

   sp = reverse_string(*sp, (short)strlen(sp));
   printf("%s\n\n",sp);

/* A real array of char is passed as an array reference
 * since that is what the Pascal procedure actually expects.
 */

   printf("%s\n",an_array);
   sp = reverse_string(an_array, (short)strlen(an_array));
   printf("%s\n",sp);
}
```

The output is:

```
This is an example
elpmaxe na si sihT

This is an example

This is the second example
elpmaxe dnoces eht si sihT
```

E.4.5 Passing Pointers

Passing pointers between C and Pascal programs is fairly straightforward. In both cases, pointers are 4-byte entities. The example below shows a simple linked-list application. The

C program creates the first element of the list and then calls the Pascal routine **append()** to add new elements to the list. The function **printlist()** is a C routine that prints the entire list. In addition to illustrating how to pass pointers, this example also shows the correspondence of Pascal *records* to C *structures*.

The Pascal program is:

```
MODULE pointer_example;

TYPE
    link = ^list;
    list =
        RECORD
           nex : link;
           data : char;
        END;

PROCEDURE append (firstrec : link;
                  val      : CHAR);
    VAR
      newdata : link;

    BEGIN
       new(newdata);          {allocate memory for new element.}

       WHILE firstrec^.nex <> NIL DO
          firstrec := firstrec^.nex;

       firstrec^.nex := newdata;
       newdata^.data := val;
       newdata^.nex := NIL;
    END;
```

The C program is shown below. Note that C's NULL pointer (defined in **<stdio.h>**) is equivalent to Pascal's NIL pointer.

```
#module pass_pointer_c
#include <stdio.h>
static struct list {
         struct list *next;
         char data;
       };

main()
{
    std_$call void append();
    extern void printlist();
    struct list first,*base;
    char ch='z';

    first.data = 'a';       /* assign value to first element of
                             * linked list
                             */
    first.next = NULL;    /* The first element is also the last
                           * so set pointer to NULL
                           */
    base = &first;      /* base points to the beginning of the
                         * list
                         */
    append(base,(char)'b');   /* Must cast a char constant. */
    append(base,ch);

    printlist(base);

}

/* printlist() prints the data in each member of the list. */

void printlist(base)
struct list *base;
{
    do {
       printf("%c\n",base->data);
       base= base->next;
       }  while (base != NULL);
}
```

After compiling and binding these routines, the output is:

```
a
b
z
```

E.4.6 Simulating the BOOLEAN Type

The Pascal **BOOLEAN** type is an 8-bit entity that evaluates to TRUE when its numeric value is -1 and to FALSE when its numeric value is 0. The **BOOLEAN** type can be simulated in C with the **char** data type. Suppose that you want to call the Pascal routine shown below. This routine takes a **BOOLEAN** argument and returns a **BOOLEAN** result (the opposite of the argument).

```
MODULE pass_boolean_p;

FUNCTION bool( bool_arg : BOOLEAN) : BOOLEAN;
    BEGIN
      writeln('Pascal value of argument:',bool_arg);
      bool_arg := NOT bool_arg;
      writeln('Pascal value returned:   ',bool_arg);
      bool := bool_arg;
    END;
```

The C program below shows several ways to invoke **bool()**.

```
#module pass_boolean_c
#define TRUE ((char)-1)    /* Cast to char and set all bits to
                            *  1.
                            */
#define FALSE ((char)0)    /* Cast to char and set all bits to
                            * 0.
                            */
main()
{
    std_$call char bool();
    int x;

    printf("Numeric value of argument: %d\n",TRUE);
    x = (bool(TRUE));
    printf("Numeric value returned: %d\n\n",x);

    printf("Numeric value of argument: %d\n",FALSE);
    x = (bool(FALSE));
    printf("Numeric value returned: %d\n\n",x);

}
```

The output after compiling, binding and executing is:

```
Numeric value of argument: -1
Pascal value of argument:           TRUE
Pascal value returned:             FALSE
Numeric value returned: 0

Numeric value of argument: 0
Pascal value of argument:          FALSE
Pascal value returned:              TRUE
Numeric value returned: -1
```

E.5 FORTRAN Examples

The following examples show how to pass various objects of different types and sizes to FORTRAN routines. Remember that FORTRAN does not make local copies of parameters. Therefore, if you change the value of a parameter in a FORTRAN routine, the corresponding argument in the C program is also changed. Do not pass constants or expressions as arguments if the FORTRAN routine attempts to change the argument value.

There are a two restrictions concerning the types of data that you can pass to, or return from, a FORTRAN routine:

- You cannot pass an *assumed-size* array from C to FORTRAN. In other words, the called FORTRAN routine *cannot* declare an array parameter with an asterisk, as in:

  ```
  SUBROUTINE assumed_size(ar)
  INTEGER*4 ar(*)
  ```

- You cannot return a character array of any size, including 1, from a FORTRAN function. For instance, a FORTRAN function declared as

  ```
  CHARACTER FUNCTION char_func()
  ```

  cannot be called from a C program.

E.5.1 Passing Integers

Suppose you need to send a 16-bit integer and a 32-bit integer to a FORTRAN routine that returns a 32-bit integer. In our example, the FORTRAN function returns the sum of the two arguments squared.

```
INTEGER*4 FUNCTION PASS_INT(invar1,invar2)
INTEGER*2 invar1
INTEGER*4 invar2

    PASS_INT = (invar1*invar1) + (invar2*invar2)
END
```

The C program below shows a variety of ways to call this routine. Note especially that constants that can fit in 16 bits are passed as **shorts** unless you cast them. Non-constant expressions are passed as **ints** unless you cast them to **short**.

```
#module pass_int_cf
main()
{
  std_$call int pass_int();
  short arg1;                  /* arg1 is 16 bits */
  int arg2,answer;             /* arg2 and answer are 32 bits */

/* Initialize variables. */
  arg1=10;
  arg2=20;

/* First we call pass_int with the correct-size arguments.
 * No casting is necessary.
 */
  answer = pass_int( arg1, arg2);
  printf("%d\t%d\t%d\n",arg1,arg2,answer);

/* If we want to send arg2 as the first argument and arg1 as
 * the second, both arguments must be cast.
 */
  answer = pass_int ((short) arg2 , (long) arg1);
  printf("%d\t%d\t%d\n",arg2,arg1,answer);

/* Any expression that contains a variable is converted to an
 * int.
 */
  answer = pass_int ((short)(arg1+5), (arg2+arg1));
  printf("%d\t%d\t%d\n",(arg1+5),(arg1+arg2),answer);

/* By default, integral constant expressions are passed as
 * shorts.
 */
/* Both constants are short because they are in the range:
 * -32767 - 32767.
 */
  answer = pass_int( (short) 10 , (long)(20*3);
  printf("%d\t%d\t%d\n",10,60,answer);

/* Append L to constant to make it long. */
  answer = pass_int( (short) 5 , 3L);
  printf("%d\t%d\t%d\n",5,3,answer);

/* Chars may be sent as integer values, but they are 16 bits.
 */
  answer = pass_int( (short) 'A' , (long) 'B');
  printf("%d\t%d\t%d\n",'A','B',answer);
}
```

The output is:

```
10      20      500
20      10      500
15      30      1125
10      60      3700
5       3       34
65      66      8581
```

E.5.2 Passing Floating-Point Numbers

The rules for passing floating-point numbers are similar to the rules for passing integers. All arguments must agree in type, either by declaration or by casting. By default, all float-

ing-point constants and expressions are passed as **double** values unless explicitly cast to **float**.

The example below shows a FORTRAN subroutine that accepts the values of the two sides of a right-angle triangle and returns the length of the hypotenuse. The first parameter is four bytes and the second is eight bytes. The result is eight bytes.

```
REAL*8 FUNCTION hypot(side1,side2)
REAL*4 side1
REAL*8 side2

hypot = SQRT((side1*side1) + (side2*side2))

END
```

The C program below shows various ways to call **hypot()**.

```
#module pass_float_cf
main()
{
  std_$call double hypot();
  float arg1=3.0;
  double arg2=4.0,result;

  printf("side1\t\t\side2\t\thypotenuse\n\n");

/* First we call it with the correct data types. */

  result = hypot(arg1, arg2);
  printf("%f\t%f\t%f\n",arg1,arg2,result);

/* If we reverse the order of the arguments, we must cast both.
 */

  result = hypot( (float) arg2, (double) arg1);
  printf("%f\t%f\t%f\n",arg2,arg1,result);

/* Any expression that contains a floating-point variable is
 * converted to double.
 */
  result = hypot((float) (arg1+arg2), (arg2+2));
  printf("%f\t%f\t%f\n",(arg1+arg2),(arg2+2),result);

/* When we pass constant expressions, the float argument must
 * be cast.
 */
  result = hypot( (float)7.5 , (double) 3.2);
  printf("%f\t%f\t%f\n",7.5,3.2,result);
}
```

The output is:

```
side1       side2       hypotenuse

3.000000    4.000000    5.000000
4.000000    3.000000    5.000000
7.000000    6.000000    9.219544
7.500000    3.200000    8.154140
```

E.5.3 Passing Character Data

When a FORTRAN routine expects a FORTRAN **CHARACTER** type, make sure that the argument you supply is passed as a reference to eight bits, not as a reference to 16 or 32 bits. You can do this either by passing a **char** variable or by casting the argument to **char**. Be especially careful of character constants, because for **std_$call** functions all integral constant arguments, including character constants, are passed as 2-byte integers if possible.

Note that you cannot return a character from a FORTRAN function. To return a character variable, create a subroutine and return the character value in a parameter.

Consider the following FORTRAN case-inversion routine that takes two character arguments. The routine inverts the case of the first argument and returns the result through the second argument.

The FORTRAN routine is:

```
SUBROUTINE UPPER_LOWER(in_char,inverted)
CHARACTER in_char,inverted

IF (ICHAR(in_char) .LE. 97) THEN
    inverted = CHAR(ICHAR(in_char) + 32)
ELSE
    inverted = CHAR(ICHAR(in_char) - 32)
END IF

END
```

The following C program calls **upper_lower()** in a variety of ways.

```
#module pass_charf_c
main()
{
  std_$call void upper_lower();
  char out_char,result;          /* 8-bit variables */
  short long_char;               /* 16-bit variable  */

  out_char = 'A';
  long_char = 'b';
  printf("Original Char\t\tCase-Inverted\n\n");

/* We do not have to cast out_char because it is 8 bits. */
  upper_lower(out_char,result);
  printf("\t%c\t\t\t\t\t%c\n", out_char,result);

/* The short int argument must be cast to char. */

  upper_lower((char) long_char,result);
  printf("\t%c\t\t\t\t\t%c\n",'b', result);

/* This is the right way to pass a character constant. */

  upper_lower((char) 'c',result);
  printf("\t%c\t\t\t\t\t%c\n",'c',result);

/* You can send integers if they can be represented in 8 bits.
*/

  upper_lower((char) 81,result);
  printf("\t%c\t\t\t\t\t%c\n",(char)81,result);

/* THIS DOESN'T WORK BECAUSE THE CONSTANT IS NOT CAST. */

  upper_lower('c',result);
  printf("\t%c\t\t\t\t\t%c\n",'c', result);

}
```

The result of program execution is:

```
Original Char     Case-Inverted

    A                   a
    b                   B
    c                   C
    Q                   q
    c
```

Note that when we try to pass the constant 'c' without casting it, the result is an unprintable character.

E.5.4 Passing Arrays

There are two points to remember when passing arrays from C to FORTRAN:

- FORTRAN and C access multidimensional arrays in a different order. In C, the rightmost subscript varies fastest while in FORTRAN the leftmost subscript varies fastest.

- When FORTRAN expects an array argument, you must pass it either the name of the array or a dereferenced pointer to an array. For **std_$call** invocations, an array name and a pointer are *not* the same.

The following example illustrates how to pass a character array from C to FORTRAN. Note that you can declare the array in FORTRAN as a character string or as an array of type **CHARACTER**. The two FORTRAN routines shown here return the last character of a string and the next-to-last character, respectively.

```
C Pass a string and get the last char.

        SUBROUTINE pass_char_array(ca, clen, outchar)
        CHARACTER ca(256)
        INTEGER*2 clen
        CHARACTER outchar

C Test for null string.

        IF (clen .LT. 1) THEN
            outchar = ' '
            RETURN
        ENDIF

        outchar = ca(clen)
        RETURN
        END

C Pass a string and get the next-to-last char.

        SUBROUTINE pass_char_string(ca, clen, outchar)
        CHARACTER*256 ca
        INTEGER*2 clen
        CHARACTER outchar

C Test for null string.

        IF (clen .LT. 1) THEN
            outchar = ' '
            return
        ENDIF

        outchar = ca(clen-1:clen-1)
        RETURN
        END
```

The following C program calls these FORTRAN routines.

```
#module pass_char_array
main()
{
    std_$call void pass_char_string();
    std_$call void pass_char_array();
    char result, *s1 = "This is the first string";
    static char s2[] = "This is the second string";
    short length;

/* First we pass a dereferenced pointer. */

    length = strlen(s1);
    pass_char_string(*s1,length,result);
    printf("The second to last character is %c\n",result);


/* Then we pass an array. */

    pass_char_array(s2, ((short)strlen(s2)),result);
    printf("The last character is %c\n",result);
}
```

The result is:

```
The second to last character is n
The last character is g
```

E.5.4.1 Passing Adjustable Arrays

The following example illustrates how to pass an **adjustable array** from C to FORTRAN. The C program passes two arguments: an array of integers and the size of the array. The FORTRAN routine uses the second argument to declare the size of the array. The routine then returns the average value of the array elements.

```
C  Pass an array of long int and return the average.

      INTEGER*4 FUNCTION pass_int_array(larray, array_len)
      INTEGER*4 array_len
      INTEGER*4 larray(array_len)
      INTEGER*4 i, tot

      tot = 0
      DO i = 1,array_len
        tot = tot + larray(i)
        print *,'larray(',i,') = ',larray(i)
      END DO

      pass_int_array = tot / array_len
      RETURN
      END
```

The C program is:

```
#module pass_int_array
main()
{
    std_$call int pass_int_array();
    static int average, array_size,pass_ar-
ray[]={325,478,982,331,21,56,79};

    array_size=sizeof(pass_array)/4;
    average = pass_int_array(pass_array,array_size);
    printf("The average is: %d\n",average);
}
```

The result is:

```
larray( 1) =  325
larray( 2) =  478
larray( 3) =  982
larray( 4) =  331
larray( 5) =  21
larray( 6) =  56
larray( 7) =  79
The average is: 324
```

E.5.4.2 Passing Multidimensional Arrays

When you pass a multidimensional array, it is important to remember that in C the rightmost subscript varies fastest while in FORTRAN the leftmost subscript varies fastest. The example below shows the consequences of this difference.

The FORTRAN routine is:

```
SUBROUTINE dyn_dim(arr, x, y)
INTEGER*4 x, y
INTEGER*4 arr(x, y)
INTEGER*2 i, j

WRITE(*,*)
WRITE(*,*) 'This is the FORTRAN array:'
DO i = 1, x
  DO j = 1, y
     WRITE(*,*) 'arr(',i,',',j,') = ',arr(i,j)
  END DO
END DO
END
```

The C program is:

```
#module multi_dim_array
main()
{
    std_$call void dyn_dim();
    static int arr[2][3]={1,2,3,4,5,6};
    short i,j;

    i=0; j=0;
    printf("This is the C array:\n");

    while (i<=1) {
        while (j<=2) {
            printf("arr(%d,%d) = %d\n",i,j,arr[i][j]);
            j++;
            }
        i++;
        j = 0;
        }
    dyn_dim(arr,(long)2, (long)3);
}
```

The result is:

```
This is the C array:
arr(0,0) = 1
arr(0,1) = 2
arr(0,2) = 3
arr(1,0) = 4
arr(1,1) = 5
arr(1,2) = 6

 This is the FORTRAN array:
 arr( 1, 1) =  1
 arr( 1, 2) =  3
 arr( 1, 3) =  5
 arr( 2, 1) =  2
 arr( 2, 2) =  4
 arr( 2, 3) =  6
```

E.5.5 Passing Pointers

As an extension to the ANSI standard, Domain FORTRAN enables a FORTRAN routine to dereference pointers passed from C or Pascal programs. For complete details, consult the *Domain FORTRAN User's Guide*.

In the following example, the C program passes the FORTRAN subroutine a pointer to a structure that contains four **short** integers. By using the the **POINTER** statement, the FORTRAN subroutine is able to modify the structure elements.

The FORTRAN subroutine is:

```
SUBROUTINE pass_point(p1)
INTEGER*4 p1
INTEGER*2 a,b,c,d
POINTER/p1/a,b,c,d

a=a+1
b=2**a
c=3**a
d=4**a

END
```

The C program is:

```
#module pass_point_c
struct S {
    short s1,s2,s3,s4;
    } struct_pass = {1,1,1,1};
main()
{
    std_$call void pass_point();
    struct S *p;

    p = &struct_pass;
    pass_point(p);
    printf("%d\n%d\n%d\n%d\n",struct_pass.s1,struct_pass.s2,
          struct_pass.s3,struct_pass.s4);

}
```

The result is:

```
2
4
9
16
```

E.5.6 Simulating the LOGICAL Types

The FORTRAN **LOGICAL** type is a 4-byte entity that evaluates to TRUE when its numeric value is -1 and to FALSE when its numeric value is 0. Although FORTRAN allocates four bytes, it uses only one of them (the high byte). Therefore, in C you can simulate the logical type with either a **char** type or an **int** type. For the best results, we recommend the following:

- Declare *in* arguments (those that are *not* changed in the FORTRAN code) as **char**s.
- Declare *out* arguments (those that are changed in the FORTRAN routine) as a **union** of a **char** and an **int**.
- Declare FORTRAN functions that return a LOGICAL value as type **char**.

The following example shows all three cases. The FORTRAN function accepts an *in* argument and an *out* argument and returns a LOGICAL value.

```
LOGICAL FUNCTION pass_logical(in_arg,out_arg)
LOGICAL in_arg, out_arg

PRINT *,'FORTRAN value of in_arg:',in_arg
PRINT *,'FORTRAN value of out_arg:',out_arg
out_arg = .NOT. out_arg
pass_logical = in_arg .EQV. out_arg
PRINT *,'FORTRAN value returned:', pass_logical

END
```

The C program below shows how to invoke **pass_logical**.

```
#module pass_logical_c
#define TRUE ((char)-1)  /* Cast to char and set all bits to 1.
                          */
#define FALSE ((char)0)  /* Cast to char and set all bits to 0.
                          */
main()
{
    std_$call char pass_logical();
    char arg1,result;
    union {
          char log;
          int filler;
          } arg2;

    arg1 = TRUE;
    arg2.log = TRUE;
    printf("C numeric value of arg1: %d\n",arg1);
    printf("C numeric value of arg2: %d\n",arg2.log);
    result = pass_logical(arg1,arg2);
    printf("C numeric value of arg2 after function call:
%d\n",arg2.log);
    printf("C numeric value returned: %d\n\n",result);

    printf("C numeric value of arg1: %d\n",arg1);
    printf("C numeric value of arg2: %d\n",arg2.log);
    result = pass_logical(arg1,arg2);
    printf("C numeric value of arg2 after function call:
%d\n",arg2.log);
    printf("C numeric value returned: %d\n\n",result);

}
```

The output after compiling, binding, and executing is:

```
C numeric value of arg1: -1
C numeric value of arg2: -1
 FORTRAN value of in_arg: T
 FORTRAN value of out_arg: T
 FORTRAN value returned: F
C numeric value of arg2 after function call: 0
C numeric value returned: 0

C numeric value of arg1: -1
C numeric value of arg2: 0
 FORTRAN value of in_arg: T
 FORTRAN value of out_arg: F
 FORTRAN value returned: T
C numeric value of arg2 after function call: -1
C numeric value returned: -1
```

E.5.7 Simulating the COMPLEX Type

The FORTRAN **COMPLEX** data type is stored as two 4-byte floating-point numbers, the first representing the real part and the second representing the imaginary part of a complex value. It is easy to simulate in C via a structure containing two floating-point members. In the following example, the FORTRAN function accepts a **COMPLEX** argument, and returns the square of the argument.

```
COMPLEX FUNCTION pass_complex(com_param)
COMPLEX com_param

pass_complex = com_param * com_param

END
```

The C program is:

```
#module pass_complex_c
main()
{
    struct complex {
        float real;
        float imag;
        };
    std_$call struct complex pass_complex();
    static struct complex result,arg = {2.5,3.5};

    printf("Complex Number\t\t\tSquare of Number\n\n");
    result = pass_complex(arg);
    printf("(%f,%f)\t\t(%f,%f)\n",arg.real,arg.imag,
            result.real, result.imag);
}
```

The result is:

```
Complex Number          Square of Number

(2.500000,3.500000)     (-6.000000,17.500000)
```

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain C Language Reference*
Order No.: 002093-A00
Date of Publication: July, 1988

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the Domain system?_____

What parts of the manual are especially useful for the job you are doing?_____

What additional information would you like the manual to include?_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)_____

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

LD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

OLD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Domain C Language Reference*
Order No.: 002093-A00
Date of Publication: July, 1988

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the Domain system?_____

What parts of the manual are especially useful for the job you are doing?_____

What additional information would you like the manual to include?_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)_____

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

.D

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

LD

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals fro your point of view.

Document Title: *Domain C Language Reference*
Order No.: 002093-A00
Date of Publication: July, 1988

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the Domain system?_____

What parts of the manual are especially useful for the job you are doing?_____

What additional information would you like the manual to include?_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure or table number wherever possible. Specify additional index entries.)_____

Your Name Date

Organization

Street Address

City State Zip

No postage necessary if mailed in the U.S.

cut or fold along dotted line

-D

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

LD


0002093-A001