Chapter 2
Introduction to the C Language

In Chapter 1, we traced the evolution of computer languages from the machine languages to high-level languages. As mentioned, **C** (the language used exclusively in this book) is a high-level language. Since you are going to spend considerable time working with the language, you should have some idea of its origins and evolution.

In this chapter we introduce the basics of the C language. You will write your first program, which is traditionally known in C as the "Hello World," or "Greeting" program. Along the way we will introduce you to the concepts of data types, constants, and variables. Finally, you will see two C library functions that read and write data. Since this chapter is just an introduction to C, most of these topics are covered only in sufficient detail to enable you to write your first program. They will be fully developed in future chapters.

Objectives

- ❏ To understand the structure of a C-language program
- ❏ To write your first C program
- ❏ To introduce the *include* preprocessor command
- ❏ To be able to create good identifiers for objects in a program
- ❏ To be able to list, describe, and use the C basic data types
- ❏ To be able to create and use variables and constants in a program
- ❏ To understand input and output concepts as they apply to C programs
- ❏ To be able to use simple input and output statements
- ❏ To understand the software engineering role in documentation, data naming, and data hiding

2.1 Background

C is a structured programming language. It is considered a high-level language because it allows the programmer to concentrate on the problem at hand and not worry about the machine that the program will be using. While many languages claim to be machine independent, C is one of the closest to achieving that goal. That is another reason why it is used by software developers whose applications have to run on many different hardware platforms.

C, like most modern languages, is derived from ALGOL, the first language to use a block structure. ALGOL never gained wide acceptance in the United States, but it was widely used in Europe.

ALGOL's introduction in the early 1960s paved the way for the development of structured programming concepts. Some of the first work was done by two computer scientists, Corrado Bohm and Guiseppe Jacopini, who published a paper in 1966 that defined the concept of structured programming. Another computer scientist, Edsger Dijkstra, popularized the concept. His letter to the editors of the *Communications of the ACM* (Association of Computing Machinery) brought the structured programming concept to the attention of the computer science community.

Several obscure languages preceded the development of C. In 1967, Martin Richards developed a language he called Basic Combined Programming Language, or BCPL. Ken Thompson followed in 1970 with a similar language he simply called B. B was used to develop the first version of UNIX, one of the popular network operating systems in use today. Finally, in 1972, Dennis Ritchie developed C, which took many concepts from ALGOL, BCPL, and B and added the concept of data types. This path, along with several others, is shown in Figure 2-1.

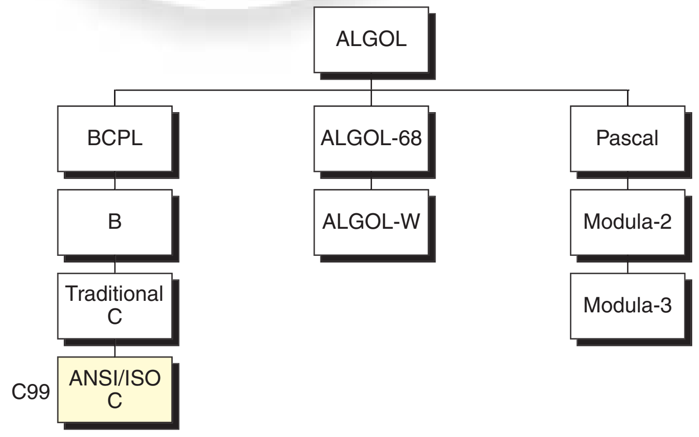


FIGURE 2-1 Taxonomy of the C Language

What is known as traditional C is this 1972 version of the language, as documented and popularized in a 1978 book by Brian W. Kernighan and Dennis Ritchie.[1] In 1983, the American National Standards Institute (ANSI) began the definition of a standard for C. It was approved in December 1989. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C is known as C89.

In 1995, minor changes were made to the standard. This version is known as C95. A much more significant update was made in 1999. The changes incorporated into the standard, now known as C99, are summarized in the following list.

1. Extensions to the character type to support non-English characters
2. A Boolean type
3. Extensions to the integer type
4. Inclusion of type definitions in the *for* statement.
5. Addition of imaginary and complex types
6. Incorporation of the C++ style line comment (//)

We use the Standard C in this book.

2.2 C Programs

It's time to write your first C program! This section will take you through all the basic parts of a C program so that you will be able to write it.

Structure of a C Program

Every C program is made of one or more preprocessor commands, a global declaration section, and one or more functions. The **global declaration section** comes at the beginning of the program. We will talk more about it later, but the basic idea of global declarations is that they are visible to all parts of the program.

The work of the program is carried out by its functions, blocks of code that accomplish a task within a program. One, and only one, of the functions must be named *main*. The *main* function is the starting point for the program. All functions in a program, including *main*, are divided into two sections: the declaration section and the statement section. The **declaration section** is at the beginning of the function. It describes the data that you will

1. Brian Kernighan and Dennis Ritchie, *The C Programming Language,* 2nd ed. (Englewood Cliffs, N.J.: Prentice Hall, 1989).

be using in the function. Declarations in a function are known as local declarations (as opposed to global declarations) because they are visible only to the function that contains them.

The **statement section** follows the declaration section. It contains the instructions to the computer that cause it to do something, such as add two numbers. In C, these instructions are written in the form of **statements**, which gives us the name for the section.

Figure 2-2 shows the parts of a simple C program. We have explained everything in this program but the preprocessor commands. They are special instructions to the preprocessor that tell it how to prepare the program for compilation. One of the most important of the preprocessor commands, and one that is used in virtually all programs, is **include**. The include command tells the preprocessor that we need information from selected libraries known as **header files**. In today's complex programming environments, it is almost impossible to write even the smallest of programs without at least one library function. In your first program, you will use one include command to tell C that you need the input/output library to write data to the monitor.

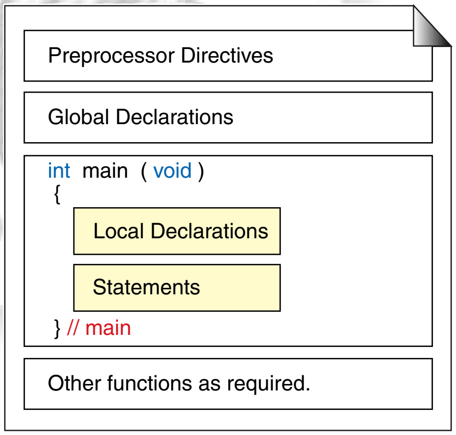
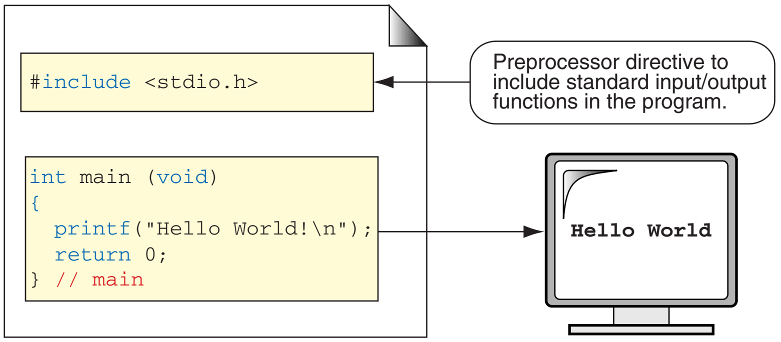


FIGURE 2-2 Structure of a C Program

Your First C Program

Your first C program will be very simple (see Figure 2-3). It will have only one preprocessor command, no global declarations, and no local definitions. Its purpose will be simply to print a greeting to the user. Therefore, its statement section will have only two statements: one that prints a greeting and one that stops the program.


Preprocessor directive to include standard input/output functions in the program.
Hello World

```
#include <stdio.h>

int main (void)
{
  printf("Hello World!\n");
  return 0;
} // main
```

FIGURE 2-3 The Greeting Program

Preprocessor Commands

The preprocessor commands come at the beginning of the program. All preprocessor commands start with a pound sign (#); this is just one of the rules of C known as its **syntax.** Preprocessor commands can start in any column, but they traditionally start in column 1.

The preprocessor command tells the compiler to include the standard input/output library file in the program. You need this library file to print a message to the terminal. Printing is one of the input/output processes identified in this library. The complete syntax for this command is shown below.

```
#include <stdio.h>
```

The syntax of this command must be exact. Since it is a preprocessor command, it starts with the pound sign. There can be no space between the pound sign and the keyword, *include*. Include means just what you would think it does. It tells the preprocessor that you want the library file in the pointed brackets (< >) *included* in your program. The name of the header file is *stdio.h*. This is an abbreviation for "standard input/output header file."

main

The executable part of your program begins with the function *main*, which is identified by the function header shown below. We explore the meaning of the function syntax in Chapter 4. For now, all you need to understand is that *int* says that the function will return an integer value to the operating system, that the function's name is *main*, and that it has no parameters (the parameter list is void). Note that there is no punctuation after the function header.

```
int main (void)
```

Within *main* there are two statements: one to print your message and one to terminate the program. The print statement uses a library function to do the actual writing to the monitor. To invoke or execute this print function, you *call* it. All function call statements consist of the name of the function, in this case *printf*, followed by a **parameter list** enclosed in parentheses. For your simple program, the parameter list simply contains what you want displayed, enclosed in two double quote marks (`"..."`). The `\n` at the end of the message tells the computer to advance to the next line in the output.

The last statement in your program, `return 0`, terminates the program and returns control to the operating system. One last thing: The function *main* starts with an open brace (`{`)and terminates with a close brace (`}`).

Comments

Although it is reasonable to expect that a good programmer should be able to read code, sometimes the meaning of a section of code is not entirely clear. This is especially true in C. Thus, it is helpful if the person who writes the code places some **comments** in the code to help the reader. Such comments are merely internal **program documentation**. The compiler ignores these comments when it translates the program into executable code. To identify a comment, C uses two different formats: block comments and line comments.

Block Comment

A **block comment** is used when the comment will span several lines. We call this comment format block comment. It uses opening and closing comment tokens. A **token** is one or more symbols understood by the compiler that help it interpret code. Each comment token is made of two characters that, taken together, form the token; there can be no space between them. The opening token is /* and the closing token is */. Everything between the opening and closing comment tokens is ignored by the compiler. The tokens can start in any column, and they do not have to be on the same line. The only requirement is that the opening token must precede the closing token. Figure 2-4 shows two examples of block comments.

```
/* This is a block comment that
   covers two lines.              */


/*
** It is a very common style to put the opening token
** on a line by itself, followed by the documentation
** and then the closing token on a separate line. Some
** programmers also like to put asterisks at the beginning
** of each line to clearly mark the comment.
*/
```

FIGURE 2-4 Examples of Block Comments

Line Comment

The second format, the **line comment**, uses two slashes (//) to identify a comment. This format does not require an end-of-comment token; the end of the line automatically ends the comment. Programmers generally use this format for short comments. The line-comment token can start anywhere on the line. Figure 2-5 contains two examples of line comments.

```
        // This is a whole line comment

        a = 5;          // This is a partial line comment
```

FIGURE 2-5 Examples of Line Comments

Although they can appear anywhere, comments cannot be nested. In other words, we cannot have comments inside comments. Once the compiler sees an opening block-comment token, it ignores everything it sees until it finds the closing token. Therefore, the opening token of the nested comment is not recognized, and the ending token that matches the first opening token is left standing on its own. This error is shown in Figure 2-6.

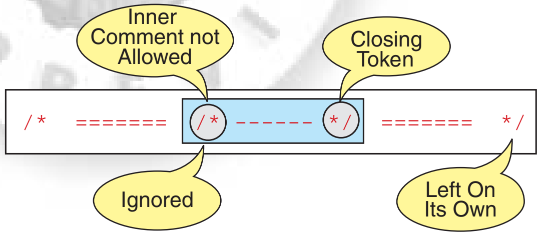


FIGURE 2-6 Nested Block Comments Are Invalid

The Greeting Program

Program 2-1 shows the greeting program just as we would write it. We have included some comments at the beginning that explain what the program is going to do. Each program we write begins with documentation explaining the purpose of the program. We have also shown comments to identify the declaration and statement sections of our program. The numbers on the left in Program 2-1 and the other programs in the text are for discussion reference. They are not part of the program.

PROGRAM 2-1 The Greeting Program

```
/* The greeting program. This program demonstrates
   some of the components of a simple C program.
      Written by:   your name here
      Date:         date program written
*/
#include <stdio.h>

int main (void)
{
// Local Declarations

// Statements

   printf("Hello World!\n");

   return 0;
} // main
```

2.3 Identifiers

One feature present in all computer languages is the **identifier**. Identifiers allow us to name data and other objects in the program. Each identified object in the computer is stored at a unique address. If we didn't have identifiers that we could use to symbolically represent data locations, we would have to know and use object's addresses. Instead, we simply give data identifiers and let the compiler keep track of where they are physically located.

Different programming languages use different syntactical rules to form identifiers. In C, the rules for identifiers are very simple. The only valid name symbols are the capital letters *A* through Z, the lowercase letters *a* through *z*, the digits 0 through 9, and the underscore. The first character of the identifier cannot be a digit.

Typically, application programs do not use the underscore for the first character either, because many of the identifiers in the C system libraries start with an underscore. In this way, we make sure that our names do not duplicate system names, which could become very confusing. The last rule is that the name we create cannot be **keywords**. Keywords, also known as **reserved words**, include syntactical words, such as *if* and *while*. For a list of the reserved words, see Appendix B.

Good identifier names are descriptive but short. To make them short, we often use abbreviations.[2] C allows names to be up to 63 characters long. If

2. One way to abbreviate an identifier is to remove any vowels in the middle of the word. For example, *student* could be abbreviated *stdnt*.

the names are longer than 63 characters, then only the first 63 are used. Table 2-1 summarizes the rules for identifiers.

1. First character must be alphabetic character or underscore.
2. Must consist only of alphabetic characters, digits, or underscores.
3. First 63 characters of an identifier are significant.
4. Cannot duplicate a keyword.

TABLE 2-1 Rules for Identifiers

You might be curious as to why the underscore is included among the possible characters that can be used for an identifier. It is there so that we can separate different parts of an identifier. To make identifiers descriptive, we often combine two or more words. When the names contain multiple words, the underscore makes it easier to read the name.

An identifier must start with a letter or underscore: it may not have a space or a hyphen.

Another way to separate the words in a name is to capitalize the first letter in each word. The traditional method of separation in C uses the underscore. A growing group of programmers, however, prefer to capitalize the first letter of each word. Table 2-2 contains examples of valid and invalid names.

C is a case-sensitive language.

Two more comments about identifiers. Note that some of the identifiers in Table 2-2 are capitalized. Typically, capitalized names are reserved for preprocessor-defined names. The second comment is that C is case sensitive. This means that even though two identifiers are spelled the same, if the case of each corresponding letter doesn't match, C thinks of them as different names. Under this rule, `num`, `Num`, and `NUM` are three different identifiers.

Valid Names		Invalid Name	
a	// Valid but poor style	$sum	// $ is illegal
student_name		2names	// First char digit
_aSystemName		sum-salary	// Contains hyphen
_Bool	// Boolean System id	stdnt Nmbr	// Contains spaces
INT_MIN	// System Defined Value	int	// Keyword

TABLE 2-2 Examples of Valid and Invalid Names

2.4 Types

A **type** defines a set of values and a set of operations that can be applied on those values. For example, a light switch can be compared to a computer type. It has a set of two values, on and off. Only two operations can be applied to a light switch: turn on and turn off.

The C language has defined a set of types that can be divided into four general categories: void, integral, floating-point, and derived, as shown in Figure 2-7.

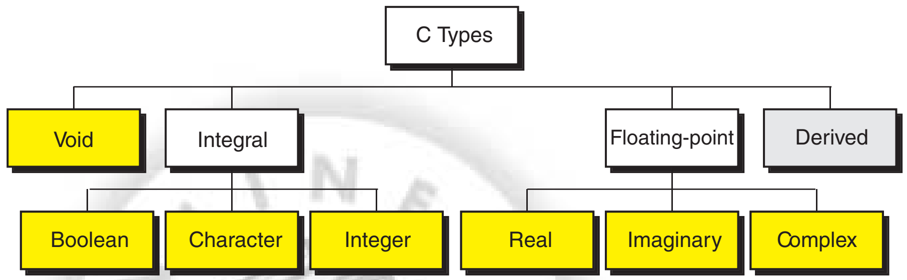


FIGURE 2-7 Data Types

In this chapter, we concentrate only on the first three types. The derived type will be discussed in future chapters.

Void Type

The void type, designated by the keyword *void*, has no values and no operations, Although having no values and operations might seem unusual, the *void* type is a very useful data type. For example, it is used to designate that a function has no parameters as we saw in the *main* function. It can also be used to define that a function has no return value as we see in Chapter 4. It can also be used to define a pointer to generic data as we will see in Chapter 9.

Integral Type

The C language has three **integral types**: Boolean, character, and integer. Integral types cannot contain a fraction part; they are whole numbers.

Boolean

With the release of C99, the C language incorporated a **Boolean** type. Named after the French mathematician/philosopher George Boole, a Boolean type can represent only two values: *true* or *false*. Prior to C99, C used integers to represent the Boolean values: a nonzero number (positive or negative) was used to represent true, and zero was used to represent false. For backward compatibility, integers can still be used to represent Boolean values; however, we recommend that new programs use the Boolean type. The Boolean type, which is referred to by the keyword *bool*, is stored in memory as 0 (*false*) or 1 (*true*).

Character

The third type is character. Although we think of characters as the letters of the alphabet, a computer has another definition. To a computer, a character is any value that can be represented in the computer's alphabet, or as it is better known, its **character set**. The C standard provides two character types: *char* and *wchar_t*.

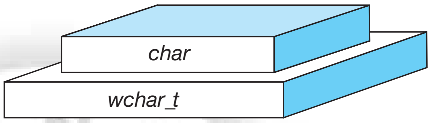


FIGURE 2-8 Character Types

Most computers use the American Standard Code for Information Interchange (**ASCII**—pronounced "ask-key") alphabet. You do not need to memorize this alphabet as you did when you learned your natural languages; however, you will learn many of the special values by using them. The ASCII code is included in Appendix A.

Most of the personal, mini-, and mainframe computers use 1 byte to store the *char* data types. A byte is 8 bits. With 8 bits, there are 256 different values in the *char* set. (Note in Appendix A that ASCII uses only half of these possible values.) Although the size of *char* is machine dependent and varies from computer to computer, normally it is 1 byte, or 8 bits.

If you examine the ASCII code carefully, you will notice that there is a pattern to its alphabet that corresponds to the English alphabet. The first 32 ASCII characters and the last ASCII character are control characters. They are used to control physical devices, such as monitors and printers, and in telecommunication systems. The rest are characters that we use to compose words and sentences.

All the lowercase letters are grouped together, as are all the uppercase letters and the digits. Many of the special characters, such as the shift characters on the top row of the keyboard, are grouped together, but some are found spread throughout the alphabet.

What makes the letter *a* different from the letter *x*? In English, it is the visual formation of the graphic associated with the letter. In the computer, it is the underlying value of the bit configuration for the letter. The letter *a* is binary `0110 0001`. The letter *x* is `0111 1000`. The decimal values of these two binary numbers are 97 and 120, respectively.

To support non-English languages and languages that don't use the Roman alphabet, the C99 standard created the wide character type (*wchar_t*). Without going in to all of the complexities, C supports two international standards, one for four-type characters and one for two-byte characters. Both of these standards support the traditional characters found in ASCII; that is, all extensions occur above the last ASCII character. The original ASCII characters are now known as the basic **Latin character set**. Generally speaking, the wide-character set is beyond the scope of an introductory programming text and is not covered in this text.

Integer

An integer type is a number without a fraction part. C supports four different sizes of the integer data type: *short int*, *int*, *long int*, and *long long int*. A *short int* can also be referred to as *short*, *long int* can be referred to as *long*, and *long long int* can be referred to as *long long*. C defines these data types so that they can be organized from the smallest to the largest, as shown in Figure 2-9. The type also defines the size of the field in which data can be stored. In C, this is true even though the size is machine dependent and varies from computer to computer.

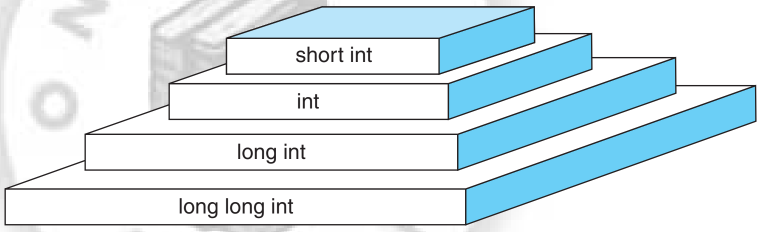


FIGURE 2-9 Integer Types

If we need to know the size of any data type, C provides an operator, *sizeof*, that will tell us the exact size in bytes. We will discuss this operator in detail in Chapter 3. Although the size is machine dependent, C requires that the following relationship always be true:

```
sizeof (short) ≤ sizeof (int) ≤ sizeof (long) ≤ sizeof (long long)
```

Each integer size can be a signed or an unsigned integer. If the integer is *signed*, then one bit must be used for a signed (0 is plus, 1 is minus). The *unsigned* integer can store a positive number that is twice as large as the signed integer of the same size.[3] Table 2-3 contains typical values for the integer types. Recognize, however, that the actual sizes are dependent on the physical hardware.

3. For a complete discussion, see Appendix D, "Numbering Systems."

Type	Byte Size	Minimum Value	Maximum Value
short int	2	–32,768	32,767
int	4	–2,147,483,648	2,147,483,647
long int	4	–2,147,483,648	2,147,483,647
long long int	8	–9,223,372,036,854,775,807	9,223,372,036,854,775,806

TABLE 2-3 Typical Integer Sizes and Values for Signed Integers

To provide flexibility across different hardware platforms, C has a library, *limits.h*, that contains size information about integers. For example, the minimum integer value for the computer is defined as `INT_MIN`, and the maximum value is defined as `INT_MAX`. See Appendix E, "Integer and Float Libraries" for a complete list of these named values.

Floating-Point Types

The C standard recognizes three **floating-point types**: real, imaginary, and complex. Like the limits library for integer values, there is a standard library, *float.h*, for the floating-point values (see Appendix E, "Integer and Float Libraries"). Unlike the integral type, real type values are always signed.

Real

The **real type** holds values that consist of an integral and a fractional part, such as 43.32. The C language supports three different sizes of real types: *float*, *double*, and *long double*. As was the case for the integer type, real numbers are defined so that they can be organized from smallest to largest. The relationship among the real types is seen in Figure 2-10.

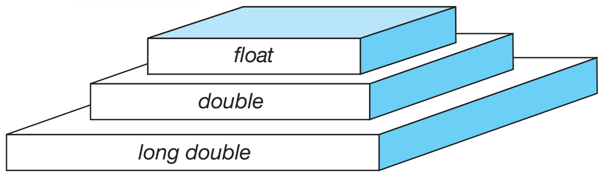


FIGURE 2-10 Floating-point Types

Regardless of machine size, C requires that the following relationship must be true:

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

Imaginary Type

C defines an **imaginary type**. An imaginary number is used extensively in mathematics and engineering. An imaginary number is a real number multiplied by the square root of –1 ($\sqrt{-1}$). The imaginary type, like the real type, can be of three different sizes: *float imaginary*, *double imaginary*, and *long double imaginary*.

Most C implementations do not support the imaginary type yet and the functions to handle them are not part of the standard. We mention them here because the imaginary type is one of the components of the complex type.

Complex

C defines a **complex type**, which is implemented by most compilers. A complex number is a combination of a real and an imaginary number. The complex type, like the real type, can be of three different sizes: *float complex*, *double complex*, and *long long complex*. The size needs to be the same in both the real and the imaginary part. We provide two program examples that use complex numbers at the end of this chapter.

Type Summary

A summary of the four standard data types is shown in Table 2-4.

Category	Type	C Implementation
Void	Void	*void*
Integral	Boolean	*bool*
	Character	*char, wchar_t*
	Integer	*short int, int, long int, long long int*
Floating-Point	Real	*float, double, long double*
	Imaginary	*float imaginary, double imaginary, long double imaginary*
	Complex	*float complex, double complex, long double complex*

TABLE 2-4 Type Summary

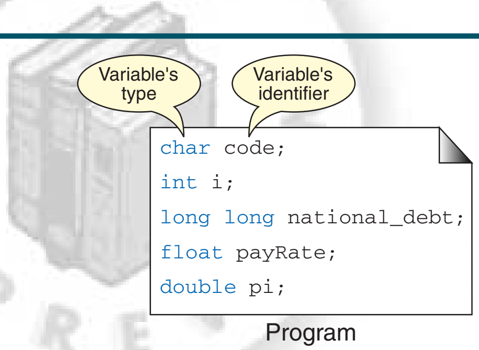
2.5 Variables

Variables are named memory locations that have a type, such as integer or character, which is inherited from their type. The type determines the values that a variable may contain the operations that may be used with its values.

Variable Declaration

Each variable in your program must be declared and defined. In C, a **declaration** is used to name an object, such as a variable. **Definitions** are used to create the object. With one exception, a variable is declared and defined at the same time. The exception, which we will see later, declares them first and then defines them at a later time. For variables, *definition* assumes that the declaration has been done or is being done at the same time. While this distinction is somewhat of an oversimplification, it works in most situations. We won't worry about the exception at this time.

When we create variables, the declaration gives them a symbolic name and the definition reserves memory for them. Once defined, variables are used to hold the data that are required by the program for its operation. Generally speaking, where the variable is located in memory is not a programmer's concern; it is a concern only of the compiler. From our perspective, all we are concerned with is being able to access the data through their symbolic names, their identifiers. The concept of variables in memory is illustrated in Figure 2-11.



```
char code;
int i;
long long national_debt;
float payRate;
double pi;
```

FIGURE 2-11 Variables

A variable's type can be any of the data types, such as *character, integer,* or *real*. The one exception to this rule is the type *void*; a variable cannot be type *void*.

To create a variable, we first specify the type, which automatically specifies it size (precision), and then its identifier, as shown below in the definition of a real variable named price of type float.

```
float price;
```

Table 2-5 shows some examples of variable declarations and definitions. As you study the variable identifiers, note the different styles used to make them readable. You should select a style and use it consistently. We prefer the use of an uppercase letter to identify the beginning of each word after the first one, although we do include examples using underscores.

```
bool   fact;
short  maxItems;              // Word separator: Capital
long   long national_debt;    // Word separator: underscore
float  payRate;               // Word separator: Capital
double tax;
float  complex voltage;
char   code, kind;            // Poor style—see text
int    a, b;                  // Poor style—see text
```

TABLE 2-5 Examples of Variable Declarations and Definitions

C allows multiple variables of the same type to be defined in one statement. The last two entries in Table 2-5 use this format. Even though many professional programmers use it, we consider it to be poor programming style. It is much easier to find and work with variables if they are defined on separate lines. This makes the compiler work a little harder, but the resulting code is no different. This is one situation in which ease of reading the program and programmer efficiency are more important than the convenience of coding multiple declarations on the same line.

Variable Initialization

We can initialize a variable at the same time that we declare it by including an initializer. When present, the **initializer** establishes the first value that the variable will contain. To initialize a variable when it is defined, the identifier is followed by the assignment operator[4] and then the initializer, which is the value the variable is to have when the function starts. This simple initialization format is shown below.

```
int count = 0;
```

Every time the function containing `count` is entered, `count` is set to zero. Now, what will be the result of the following initialization? Are both `count` and `sum` initialized or is only `sum` initialized?

```
int count, sum = 0;
```

The answer is that the initializer applies only to the variable defined immediately before it. Therefore, only `sum` is initialized! If you wanted both variables initialized, you would have to provide two initializers.

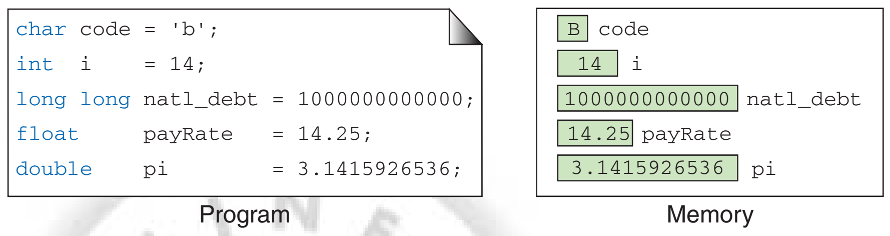
```
int count = 0, sum = 0;
```

4. The assignment operator is the equal sign (=).

Again, to avoid confusion and error, we prefer using only one variable definition to a line. The preferred code in this case would be

```
int count =  0;
int sum   =  0;
```

Figure 2-12 repeats Figure 2-11, initializing the values in each of the variables.



```
char code = 'b';
int  i     = 14;
long long natl_debt = 1000000000000;
float      payRate    = 14.25;
double     pi         = 3.1415926536;
```

FIGURE 2-12 Variable Initialization

It is important to remember that with a few exceptions that we will see later, variables are not initialized automatically. When variables are defined, they usually contain garbage (meaningless values left over from a previous use), so we need to initialize them or store data in them (using run-time statements) before accessing their values. Many compilers display a warning message when a variable is accessed before it is initialized.

When a variable is defined, it is not initialized. We must initialize any variable requiring prescribed data when the function starts.

One final point about initializing variables when they are defined: Although the practice is convenient and saves you a line of code, it also can lead to errors. It is better, therefore, to initialize the variable with an assignment statement at the proper place in the body of the code. This may take another statement, but the efficiency of the resulting program is exactly the same, and you will make fewer errors in your code.

EXAMPLE 2-1 Print Sum

At this point you might like to see what a more complex program looks like. As you read Program 2-2, note the blank lines to separate different groups of code. This is a good technique for making programs more readable. You should use blank lines in your programs the same way you use them to separate the paragraphs in a report.

PROGRAM 2-2 Print Sum of Three Numbers

```c
/* This program calculates and prints the sum of
   three numbers input by the user at the keyboard.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a;
   int b;
   int c;
   int sum;

// Statements
   printf("\nWelcome. This program adds\n");
   printf("three numbers. Enter three numbers\n");
   printf("in the form: nnn nnn nnn <return>\n");
   scanf("%d %d %d", &a, &b, &c);

   // Numbers are now in a, b, and c. Add them.
   sum = a + b + c;

   printf("The total is: %d\n\n", sum);

   printf("Thank you. Have a good day.\n");
   return 0;
}  //  main
```

```
Results:
Welcome. This program adds
three numbers. Enter three numbers
in the form: nnn nnn nnn <return>
11 22 33

The total is: 66

Thank you. Have a good day.
```

Program 2-2 Analysis Study the style of this program carefully. First, note how we start with a welcome message that tells the user exactly what needs to be entered. Similarly, at the end of the program, we print an ending message. It is a good style to print a start and end message.

This program contains three different processes. First it reads three numbers. The code to read the numbers includes the printed instructions and a read (*scanf*) statement. The second process adds the three numbers. While this process consists of only a comment and one statement, we separate it from the read process. This makes it easier for the reader to follow the program. Finally, we print the result. Again, the print process is separated from the calculate process by a blank line.

2.6 Constants

Constants are data values that cannot be changed during the execution of a program. Like variables, constants have a type. In this section, we discuss Boolean, character, integer, real, complex, and string constants.

Constant Representation

In this section, we show how we use symbols to represent constants. In the next section, we show how we can code these constants in our program.

Boolean Constants

A Boolean data type can take only two values. Therefore, we expect that we have only two symbols to represent a Boolean type. The values are *true* and *false*. As we mentioned before, a Boolean value can have only one of the two values: 0 (*false*) and 1 (*true*). We use the constant *true* or *false* in our program. To do so, however, requires that we include the Boolean library, *stdbool.h*.

Character Constants

Character constants are enclosed between two single quotes (apostrophes). In addition to the character, we can also use a backslash (\) before the character. The backslash is known as the **escape character**. It is used when the character we need to represent does not have any graphic associated with it—that is, when it cannot be printed or when it cannot be entered from the keyboard. The escape character says that what follows is not the normal character but something else. For example, `'\n'` represents the newline character (line feed). So, even though there may be multiple symbols in the character constant, they always represent only one character.

A character constant is enclosed in single quotes.

Wide-character constants are coded by prefixing the constant with an `L`, as shown in the following example.

```
L'x'
```

The character in the character constant comes from the character set supplied by the hardware manufacturer. Most computers use the ASCII character set, or as it is sometimes called, the ASCII alphabet. The ASCII character set is shown in Appendix A.

C has named the critical character values so that we can refer to them symbolically. Note that these control characters use the escape character followed by a symbolic character. They are shown in Table 2-6.

ASCII Character	Symbolic Name
null character	`'\0'`
alert (bell)	`'\a'`
backspace	`'\b'`
horizontal tab	`'\t'`
newline	`'\n'`
vertical tab	`'\v'`
form feed	`'\f'`
carriage return	`'\r'`
single quote	`'\''`
double quote	`'\"'`
backslash	`'\\'`

TABLE 2-6 Symbolic Names for Control Characters

Integer Constants

Although integers are always stored in their binary form, they are simply coded as we would use them in everyday life. Thus, the value 15 is simply coded as `15`.

If we code the number as a series of digits, its type is signed integer, or signed long integer if the number is large. We can override this default by specifying unsigned (`u` or `U`), and *long* (`l` or `L`) or *long long* (`ll` or `LL`), after the number. The codes may be combined and may be coded in any order. Note that there is no way to specify a *short int* constant. When we omit the suffix on a literal, it defaults to *int*. While both upper- and lowercase codes are allowed, we recommend that you always use uppercase to avoid confusion (especially with the lowercase letter *l*, which often looks like the number 1). Table 2-7 shows several examples of integer constants. The default types are typical for a personal computer.

Representation	Value	Type
+123	123	int
–378	–378	int
–32271L	–32,271	long int
76542LU	76,542	unsigned long int
12789845LL	12,789,845	long long int

TABLE 2-7 Examples of Integer Constants

Real Constants

The default form for real constants is *double*. If we want the resulting data type to be *float* or *long double*, we must use a code to specify the desired data type. As you might anticipate, `f` and `F` are used for *float* and `l` and `L` are used for *long double*. Again, do not use the lowercase *l* for *long double*; it is too easily confused with the number 1.

Table 2-8 shows several examples of real constants.

Representation	Value	Type
0.	0.0	double
.0	0.0	double
2.0	2.0	double
3.1416	3.1416	double
–2.0f	–2.0	float
3.1415926536L	3.1415926536	long double

TABLE 2-8 Examples of Real Constants

Complex Constants

Although we do not discuss the imaginary constants, we need to talk about complex constants that are widely used in engineering.

Complex constants are coded as two parts, the real part and the imaginary part, separated by a plus sign. The real part is coded using the real format rules. The imaginary part is coded as a real number times (`*`) the imaginary constant (`_Complex_I`). If the complex library (*complex.h*) is included, the imaginary constant can be abbreviated as `I`. Examples are shown in Table 2-9.

Representation	Value	Type
12.3 + 14.4 * I	$12.3 + 14.4 * (-1)^{1/2}$	double complex
14F + 16F * I	$14 + 16 * (-1)^{1/2}$	float complex
1.4736L+ 4.56756L * I	$1.4736 + 4.56756 * (-1)^{1/2}$	long double complex

TABLE 2-9 Examples of Complex Constants

The default form for complex constants is *double*. If we want the resulting data type to be *float* or *long double*, we must use a code to specify the desired data type. As you might anticipate, `f` and `F` are used for *float* and `l` and `L` are used for *long double*. Again, do not use the lowercase *l* for *long double*; it is too easily confused with the number 1.

The two components of a complex constant must be of the same precision, that is, if the real part is type **double**, *then the imaginary time must also be type* **double**.

Table 2-9 shows several examples of complex constants. Note that we use the abbreviated form for the imaginary part.

String Constants

A string constant is a sequence of zero or more characters enclosed in double quotes. You used a string in your first program without even knowing that it was a string! Look at Program 2-1 to see if you can identify the string.

Listed in Figure 2-13 are several strings, including the one from Program 2-1. The first example, an empty string, is simply two double quotes in succession. The second example, a string containing only the letter `h`, differs from a character constant in that it is enclosed in double quotes. When we study strings, we will see that there is also a big difference in how `h` is stored in memory as a character and as a string. The last example is a string that uses wide characters.

```
""                                      // A null string
"h"
"Hello World\n"
"HOW ARE YOU"
"Good Morning!"
L"This string contains wide characters."
```

FIGURE 2-13 Some Strings

It is important to understand the difference between the null character (see Table 2-6) and an empty string. The null character represents no value. As a character, it is 8 zero bits. An empty string, on the other hand, is a string containing nothing. Figure 2-14 shows the difference between these two constant types.

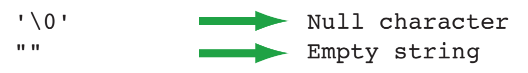


FIGURE 2-14 Null Characters and Null Strings

At this point, this is all you need to know about strings. We talk more about them and how they are stored in the computer when we study strings in Chapter 11.

Use single quotes for character constants. Use double quotes for string constants.

Coding Constants

In this section we discuss three different ways we code constants in our programs: literal constants, defined constants, and memory constants.

Literal Constants

A **literal** is an unnamed constant used to specify data. If we know that the data cannot be changed, then we can simply code the data value itself in a statement.

Literals are coded as part of a statement using the constant formats described in the previous section. For example, the literal 5 is used in the following statement.

```
a = b + 5;
```

Defined Constants

Another way to designate a constant is to use the preprocessor command define. Like all preprocessor commands, it is prefaced with the pound sign (#). The define commands are usually placed at the beginning of the program, although they are legal anywhere. Placing them at the beginning of the program makes them easy to find and change. A typical define command might be

```
#define SALES_TAX_RATE .0825
```

In the preceding example, for instance, the sales tax rate changes more often than we would like. By placing it and other similar constants at the beginning of the program, we can find and change them easily.

As the preprocessor reformats the program for the language translator, it replaces each defined name, `SALES_TAX_RATE` in the previous example with its value (`.0825`) wherever it is found in the source program. This action is just like the search-and-replace command found in a text editor. The preprocessor does not evaluate the code in any way—it just blindly makes the substitution. For a complete discussion of defined constants, see Appendix G, "Preprocessor Commands."

Memory Constants

The third way to use a constant is with memory constants. Memory constants use a C **type qualifier**, *const*, to indicate that the data cannot be changed. Its format is:

```
const type identifier = value;
```

We have seen how to define a variable, which does nothing more than give a type and size to a named object in memory. Now let us assume that we want to fix the contents of this memory location so that they cannot be changed. This is the same concept as a literal, only now we give it a name. The following code creates a memory constant, `cPi`. To help us remember that it is a constant, we preface the identifier name with *c*.

```
const float cPi = 3.14159;
```

Three points merit discussion: (1) The type qualifier comes first. (2) Then there must be an initializer. If we didn't have an initializer, then our named constant would be whatever happened to be in memory at `cPi`'s location when our program started. (3) Finally, since we have said that `cPi` is a constant, we cannot change it.

Program 2-3 demonstrates the three different ways to code `pi` as a constant.

PROGRAM 2-3 Memory Constants

```
/* This program demonstrates three ways to use
   constants.
      Written by:
      Date:
*/
#include <stdio.h>
#define PI 3.1415926536

```

continued

PROGRAM 2-3 Memory Constants *(continued)*

```c
int main (void)
{
// Local Declarations
   const double cPi = PI;

// Statements
   printf("Defined constant PI: %f\n", PI);
   printf("Memory constant cPi: %f\n", PI);
   printf("Literal constant:    %f\n", 3.1415926536);
   return 0;
}  // main
```

```
Results:
Defined constant PI:   3.141593
Memory constant cPi:   3.141593
Literal constant:      3.141593
```

2.7 Input/Output

Although our programs have implicitly shown how to print messages, we have not formally discussed how we use C facilities to input and output data. We devote two chapters, Chapters 7 and 13, to fully explain the C input/output facilities and how to use them. In this section, we describe simple input and output formatting.

Streams

In C, data is input to and output from a stream. A **stream** is a source of or destination for data. It is associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.

C uses two forms of streams: text and binary. A **text stream** consists of a sequence of characters divided into lines with each line terminated by a newline (\n). A **binary stream** consists of a sequence of data values such as integer, real, or complex using their memory representation. In this chapter, we briefly discuss only text streams. A more detailed discussion of text streams is found in Chapter 7, "Text Input/Output," and a detailed discussion of binary streams is found in Chapter 13, "Binary Input/Output."

A terminal can be associated only with a text stream because a keyboard can only send a stream of characters into a program and a monitor can only display a sequence of characters. A file, on the other hand can be associated with a text or binary stream. We can store data in a file and later retrieve them as a sequence of characters (text stream) or as a sequence of data values (binary streams).

In this chapter, we assume that the source of data is the keyboard and the destination of data is the monitor. In other words, the terminal devices we use produce or consume text streams. In C, the keyboard is known as **standard input** and the monitor is known as **standard output**.

A terminal keyboard and monitor can be associated only with a text stream. A keyboard is a source for a text stream; a monitor is a destination for a text stream.

Figure 2-15 shows the concept of streams and the two physical devices associated with input and output text streams.

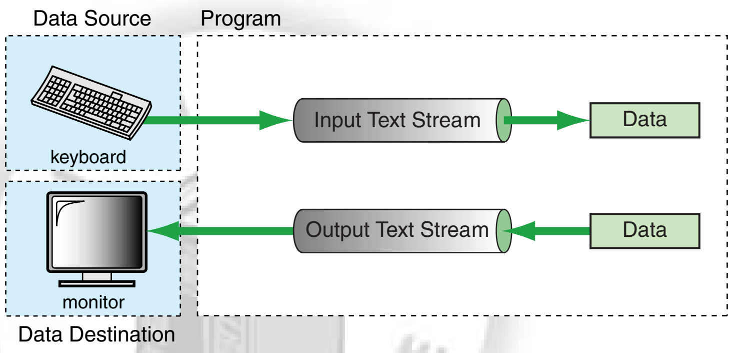


FIGURE 2-15 Stream Physical Devices

Formatting Input/Output

The previous section discussed the terminal as a text stream source and destination. We can only receive text streams from a terminal (keyboard) and send text streams to a terminal (monitor). However, these text streams often represent different data types, such as integer, real, and Boolean. The C language provides two formatting functions: *printf* for output formatting and *scanf* for input formatting. The *printf* function converts data stored in the program into a text stream for output to the monitor; the *scanf* function converts the text stream coming from the keyboard to data values and stores them in program variables. In other words, the *printf* and *scanf* functions are data to text stream and text stream to data converters.

Output Formatting: *printf*

The output formatting function is *printf*. The *printf* function takes a set of data values, converts them to a text stream using formatting instructions contained in a format control string, and sends the resulting text stream to the standard output (monitor). For example, an integer 234 stored in the program is converted to a text stream of three numeric ASCII characters ('2', '3', and '4') and then is sent to the monitor. What we see on the monitor is these three characters, not the integer 234. However, we interpret the three characters together as an integer value. Figure 2-16 shows the concept.

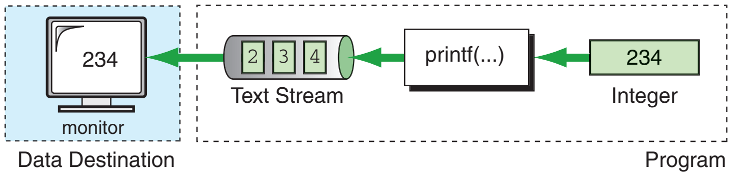


FIGURE 2-16 Output Formatting Concept

Basic Concept

The *printf* function uses an interesting design to convert data into text streams. We describe how the text stream should be formatted using a **format control string** containing zero or more **conversion specifications**. In addition to the conversion specifications, the control string may contain textual data and control characters to be displayed.

Each data value to be formatted into the text stream is described as a separate conversion specification in the control string. The specifications describe the data values' type, size, and specific format information, such as how wide the display width should be. The location of the conversion specification within the format control string determines its position within the text stream.

The control string and data values are passed to the print function (*printf*) as parameters, the control string as the first parameter and one parameter for each value to be printed. In other words, we supply the following information to the print function:

1. The format control string including any textual data to be inserted into the text stream.
2. A set of zero or more data values to be formatted.

Figure 2-17 is a conceptional representation of the format control string and two conversion specifications.

Figure 2-17(a) shows the format string and the data values as parameters for the print function. Within the control string we have specified quantity (`Qty:`) and total (`Tot:`) as textual data and two conversion specifications (`%d` and `%f`). The first specification requires an integer type value; the second requires a real type value. We discuss the conversion specifications in detail in the following section.

Figure 2-17(b) shows the formatting operation and the resulting text stream. The first data value is a literal integer; the second data value is the contents of a variable named `tot`. This part of Figure 2-17 shows how the *printf* function expands the control stream and inserts the data values and text characters.

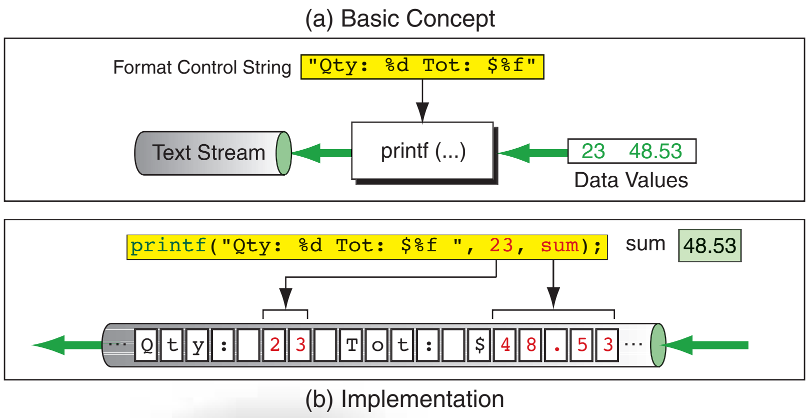


FIGURE 2-17 Output Stream Formatting Example

Format Control String Text

The control string may also contain text to be printed, such as instructions to the user, captions or other identifiers, and other text intended to make the output more readable. In fact, as we have already seen, the format string may contain nothing but text, in which case the text will be printed exactly as shown. We used this concept when we wrote the greeting program. In addition, we can also print control characters, such as tabs (`\t`), newlines (`\n`), and alerts (`\a`), by including them in the format string. Tabs are used to format the output into columns. Newlines terminate the current line and continue formatting on the next line. Alerts sound an audio signal to alert, usually to alert the user to a condition that needs attention. These control characters are seen in Table 2-6.

Conversion Specification

To insert data into the stream, we use a conversion specification that contains a start token (`%`), a **conversion code**, and up to four optional modifiers as shown in Figure 2-18. Only the field-specification token (`%`) and the conversion code are required.

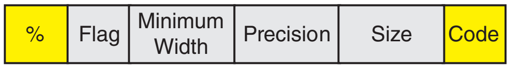


FIGURE 2-18 Conversion Specification

Approximately 30 different conversion codes are used to describe data types. For now, however, we are concerned with only three: character (`c`), integer (`d`), and floating point (`f`). These codes with some examples are shown in Table 2-10.

Type	Size[a]	Code	Example
`char`	None	`c`	`%c`
`short int`	`h`	`d`	`%hd`
`int`	None	`d`	`%d`
`long int`	None	`d`	`%ld`
`long long int`	`ll`	`d`	`%lld`
`float`	None	`f`	`%f`
`double`	None	`f`	`%f`
`long double`	`L`	`f`	`%Lf`

a. Size is discussed in the next section.

TABLE 2-10 Format Codes for Output

The **size modifier** is used to modify the type specified by the conversion code. There are four different sizes: `h`, `l` (el), `ll` (el el), and `L`. The `h` is used with the integer codes to indicate a short integer value;[5] the `l` is used to indicate a long integer value; the `ll` is used to indicate a long long integer value; and the `L` is used with floating-point numbers to indicate a long double value.

A **width modifier** may be used to specify the minimum number of positions in the output. (If the data require using more space than we allow, then *printf* overrides the width.) It is very useful to align output in columns, such as when we need to print a column of numbers. If we don't use a width modifier, each output value will take just enough room for the data.

If a floating-point number is being printed, then we may specify the number of decimal places to be printed with the **precision modifier**. The precision modifier has the format

```
.m
```

where `m` is the number of decimal digits. If no precision is specified, *printf* prints six decimal positions. These six decimal positions are often more than is necessary.

When both width and precision are used, the width must be large enough to contain the integral value of the number, the decimal point, and the number of digits in the decimal position. Thus, a conversion specification

5. The `h` code is a carry-over from assembler language where it meant half word.

of `%7.2f` is designed to print a maximum value of 9999.99. Some examples of width specifications and precision are shown below.

```
%2hd          // short integer—2 print positions
%4d           // integer—4 print positions
%8ld          // long int—8 (not 81) positions
%7.2f         // float—7 print positions: nnnn.dd
%10.3Lf       // long double—10 positions: nnnnnn.ddd
```

The **flag modifier** is used for four print modifications: justification, padding, sign, and numeric conversion variants. The first three are discussed here; the conversion variants are discussed in Chapter 7.

Justification controls the placement of a value when it is shorter than the specified width. Justification can be left or right. If there is no flag and the defined width is larger than required, the value is right justified. The default is right justification. To left justify a value, the flag is set to minus (`-`).

Padding defines the character that fills the unused space when the value is smaller than the print width. It can be a space, the default, or zero. If there is no flag defined for padding, the unused width is filled with spaces; if the flag is `0`, the unused width is filled with zeroes. Note that the zero flag is ignored if it is used with left justification because adding zeros after a number changes its value.

The **sign flag** defines the use or absence of a sign in a numeric value. We can specify one of three formats: default formatting, print signed values, or prefix positive values with a leading space. Default formatting inserts a sign only when the value is negative. Positive values are formatted without a sign. When the flag is set to a plus (+), signs are printed for both positive and negative values. If the flag is a space, then positive numbers are printed with a leading space and negative numbers with a minus sign.

Table 2-11 documents three of the more common flag options.

Flag Type	Flag Code	Formatting
Justification	None	right justified
	–	left justified
Padding	None	space padding
	0	zero padding
Sign	None	positive value: no sign negative value: –
	+	positive value: + negative value: –
	None	positive value: space negative value: –

TABLE 2-11 Flag Formatting Options

Output Examples

This section contains several output examples. We show the *printf* statement, followed by what would be printed. Cover up the solution and try to predict the results.

1. `printf("%d%c%f", 23, 'z', 4.1);`

```
23z4.100000
```

Note that because there are no spaces between the conversion specifications, the data are formatted without spaces between the values.

2. `printf("%d %c %f", 23, 'z', 4.1);`

```
23 z 4.100000
```

This is a repeat of Output Example 1 with spaces between the conversion specifications.

3.
```
int   num1  = 23;
char  zee   = 'z';
float num2  = 4.1;
printf("%d %c %f", num1, zee, num2);
```

```
23 z 4.100000
```

Again, the same example, this time using variables.

4.
```
printf("%d\t%c\t%5.1f\n", 23, 'Z', 14.2);
printf("%d\t%c\t%5.1f\n", 107, 'A', 53.6);
printf("%d\t%c\t%5.1f\n", 1754, 'F', 122.0);
printf("%d\t%c\t%5.1f\n", 3, 'P', 0.1);
```

```
23     Z     14.2
107    A     53.6
1754   F    122.0
3      P      0.1
```

In addition to the conversion specifications, note the tab character (`\t`) between the first and second, and second and third conversion specifications. Since the data are to be printed in separate lines, each format string ends with a newline (`\n`).

5. `printf("The number%dis my favorite number.", 23);`

```
The number23is my favorite number.
```

Since there are no spaces before and after the format code (`%d`), the number 23 is run together with the text before and after.

6. `printf("The number is %6d", 23);`

```
The number is     23
^^^^^^^^^^^^^^^^^^^^
```

If you count the spaces carefully, you will note that five spaces follow the word `is`. The first space comes from the space after `is` and before the `%` in the format string. The other four come from the width in the conversion specification.

7. `printf("The tax is %6.2f this year.", 233.12);`

```
The tax is 233.12 this year.
```

In this example, the width is six and the precision two. Since the number of digits printed totals five (three for the integral portion and two for the decimal portion), and the decimal point takes one print position, the full width is filled with data. The only spaces are the spaces before and after the conversion code in the format string.

8. `printf("The tax is %8.2f this year.", 233.12);`

```
The tax is   233.12 this year.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

9. `printf("The tax is %08.2f this year.", 233.12);`

```
The tax is 00233.12 this year.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This example uses the zero flag to print leading zeros. Note that the width is eight positions. Three of these positions are taken up by the precision of two digits and the decimal point. This leaves five positions for the integral portion of the number. Since there are only three digits (233), *printf* inserts two leading zeros.

10. `printf("\"%8c   %d\"", 'h', 23);`

```
"       h   23"
^^^^^^^^^^^^^^^
```

In this example, we want to print the data within quotes. Since quotes are used to identify the format string, we can't use them as print characters. To print them, therefore, we must use the escape character with the quote (`\"`), which tells *printf* that what follows is not the end of the string but a character to be printed, in this case, a quote mark.

11.
```
printf ("This line disappears.\r...A new line\n");
printf ("This is the bell character \a\n");
printf ("A null character\0kills the rest of the line\n");
printf ("\nThis is \'it\' in single quotes\n");
printf ("This is \"it\" in double quotes\n");
printf ("This is \\ the escape character itself\n");
```

```
        ...A new line
        This is the bell character
        A null character
        This is 'it' in single quotes
        This is "it" in double quotes
        This is \ the escape character itself
```

These examples use some of the control character names found in Table 2-6. Two of them give unexpected results. In Output Example 11, the return character (\r) repositions the output at the beginning of the current line without advancing the line. Therefore, all data that were placed in the output stream are erased.

The null character effectively kills the rest of the line. Had we not put a newline character (\n) at the beginning of the next line, it would have started immediately after `character`.

12. `New example with multiple flags.`
```
printf("|%-+8.2f| |%0+8.2f| |%-0+8.2f|", 1.2, 2.3, 3.4);
```

```
|+1.20   | | 0002.30| |+3.40   |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This example uses multiple flags. So that we can see the justification, each value is enclosed in vertical bars. The first value is printed left justified with the positive flag set. The second example uses zero fill with a space for the sign. Note that there is a leading space in the output. This represents the plus value. It is then followed by the leading zeros. The last example demonstrates that the zero fill is ignored when a numeric value is printed with left justification.

Common Output Errors

Each of the following examples has at least one error. Try to find each one before you look at the solutions. Your results may vary depending on your compiler and hardware.

1. `printf ("%d %d %d\n", 44, 55);`

```
        44 55 0
```

This example has three conversion specifications but only two values.

2. `printf ("%d %d\n", 44, 55, 66);`

```
        44 55
```

This example has two conversion specifications with three values. In this case, *printf* ignores the third value.

3.
```
float x = 123.45;
printf("The data are: %d\n", x);
```

```
The data are: 1079958732
```

This is a very common error in which the format specification (integer) does not match the data type (real).

Input Formatting: *scanf*

The standard input formatting function in C is *scanf* (scan formatting). This function takes a text stream from the keyboard, extracts and formats data from the stream according to a format control string, and then stores the data in specified program variables. For example, the stream of 5 characters ‘2’, ‘3’, ‘4’, ‘.’, and ‘2’ are extracted as the real 234.2. Figure 2-19 shows the concept.

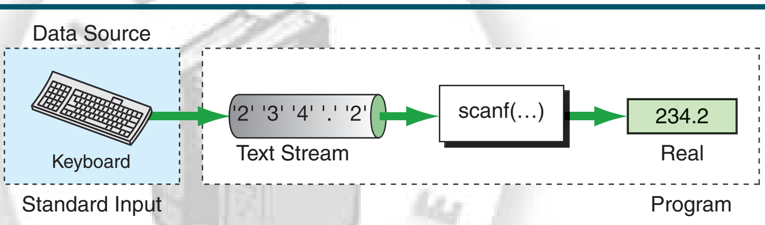


FIGURE 2-19 Formatting Text from an Input Stream

The *scanf* function is the reverse of the *printf* function.

1. A format control string describes the data to be extracted from the stream and reformatted.
2. Rather than data values as in the *printf* function, *scanf* requires the *variable* addresses were each piece of data are to be stored. Unlike the *printf* function, the destination of the data items cannot be literal values, they must store in the variables.
3. With the exception of the character specification, leading whitespaces are discarded.
4. Any non-conversion specification characters in the format string must be exactly matched by the next characters in the input stream.

   We must be careful about extra characters in the control stream. Extra characters in the control string can be divided into two categories: non-whitespace and whitespace.

a. Non-whitespace characters in the control string must exactly match characters entered by the user and are discarded by the *scanf* after they are read. If they don't match, then *scanf* goes into an error state and the program must be manually terminated.

We recommend that you don't use non-whitespace characters in the format string, at least until you learn how to recover from errors in Chapter 13. However, there are some uses for it. For example, if the users want to enter dates with slashes, such as 5/10/06, the slashes must either be read and discarded using the character format specification (see the discussion of the assignment suppression flag in the later section, "Conversion Specification") or coded as non-whitespace in the format specification. We prefer the option to read and discard them.

b. Whitespace characters in the format string are matched by zero or more whitespace characters in the input stream and discarded. There are two exceptions to this rule: the character conversion code and the scan set (see Chapter 11) do not discard whitespace. It is easy, however, to manually discard whitespace characters when we want to read a character. We simply code a space before the conversion specification or as a part of it as shown below. Either one works.

```
" %c" or "% c"
```

Remember that whenever we read data from the keyboard, there is a return character from a previous read. If we don't flush the whitespace characters when we read a character, therefore, we will get the whitespace from the previous read. To read a character, we should always code at least one whitespace character in the conversion specification; otherwise the whitespace remaining in the input stream is read as the input character. For example, to read three characters, we should code the following format string. Note the spaces before each conversion specification.

```
scanf(" %c %c %d", &c1, &c2, &c3);
```

Figure 2-20 demonstrates the input format string concept with a control string having two fields (%d and %f). The first one defines that a character will be inserted here; the second defines that a real will be inserted there. We will discuss these place holders, or format specifiers, later in the chapter.

Format Control String

Like the control string for *printf*, the control string for *scanf* is enclosed in a set of quotation marks and contains one or more conversion specifications that describe the data types and indicate any special formatting rules and/or characters.

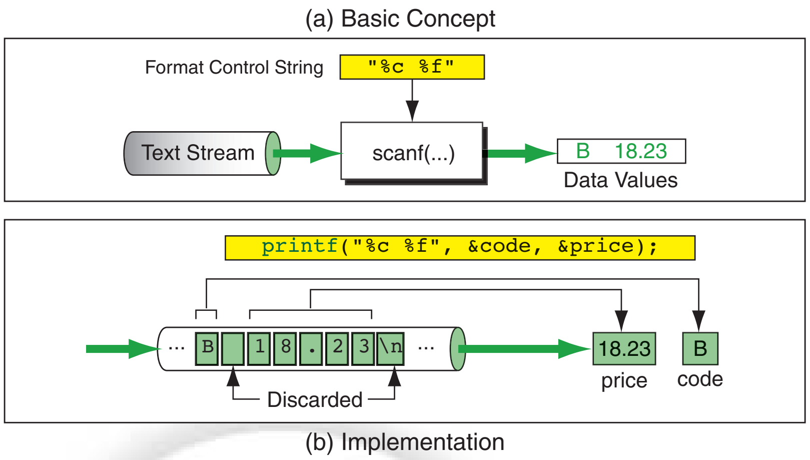


FIGURE 2-20 Input Stream Formatting Example

Conversion Specification

To format data from the input stream, we use a conversion specification that contains a start token (%), a **conversion code**, and up to three optional modifiers as shown in Figure 2-21. Only the field-specification token (%) and the conversion code are required.

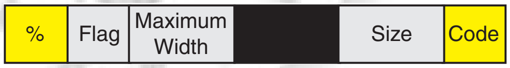


FIGURE 2-21 Conversion Specification

There are only three differences between the conversion codes for input formatting and output formatting. First, there is no precision in an input conversion specification. It is an error to include a precision; if *scanf* finds a precision it stops processing and the input stream is in the error state.

There is only one flag for input formatting, the assignment suppression flag (*). More commonly associated with text files (see Chapter 7), the assignment suppression flag tells *scanf* that the next input field is to be read but not stored. It is discarded. The following *scanf* statement reads an integer, a character, and a floating-point number from the input stream. The character is read and discarded. The other fields are read, formatted, and stored. Note that there is no matching address parameter for the data to be discarded.

```
scanf ("%d %*c %f", &x, &y);
```

The third difference is the width specification; with input formatting it is a *maximum*, not a minimum, width. The width modifier specifies the maximum number of characters that are to be read for one format code. When a width specification is included, therefore, *scanf* reads until the maximum number of characters have been processed or until *scanf* finds a whitespace character. If *scanf* finds a whitespace character before the maximum is reached, it stops.

Input Parameters

For every conversion specification there must be a matching variable in the address list. The address list contains the address of the matching variable. How do we specify an address? It's quite simple: Addresses are indicated by prefixing the variable name with an ampersand (`&`). In C, the ampersand is known as the **address operator**. Using the address operator, if the variable name is `price`, then the address is `&price`. Forgetting the ampersand is one of the most common errors for beginning C programmers, so you will have to concentrate on it when you use the *scanf* function.

scanf requires variable addresses in the address list.

Remember that the first conversion specification matches the first variable address, the second conversion specification matches the second variable address, and so on. This correspondence is very important. It is also very important that the variable's type match the conversion specification type. The C compiler does not verify that they match. If they don't, the input data will not be properly formatted when they are stored in the variable.

End of File and Errors

In addition to whitespace and width specifications, two other events stop the *scanf* function. If the user signals that there is no more input by keying **end of file** (EOF), then *scanf* terminates the input process. While there is no EOF on the keyboard, it can be simulated in most systems. For example, Windows uses the `<ctrl + z>` key combination to signal EOF. Unix and Apple Macintosh use `<ctrl + d>` for EOF. The C user's manual for your system should specify the key sequence for EOF.

Second, if *scanf* encounters an invalid character when it is trying to convert the input to the stored data type, it stops. The most common error is finding a nonnumeric character when it is trying to read a number. The valid characters are leading plus or minus, digits, and one decimal point. Any other combination, including any alphabetic characters, will cause an error. Although it is possible to detect this error and ask the user to re-input the data, we will not be able to cover the conventions for this logic until Chapter 7. Until then, be very careful when you enter data into your program.

Input Formatting Summary

Table 2-12 summarizes the rules for using *scanf*.

1. The conversion operation processes until:

   a. End of file is reached.
   b. The maximum number of characters has been processed.
   c. A whitespace character is found after a digit in a numeric specification.
   d. An error is detected.

2. There must be a conversion specification for each variable to be read.

3. There must be a variable address of the proper type for each conversion specification.

4. Any character in the format string other than whitespace or a conversion specification must be exactly matched by the user during input. If the input stream does not match the character specified, an error is signaled and *scanf* stops.

5. It is a fatal error to end the format string with a whitespace character. Your program will not run correctly if you do.

TABLE 2-12 *scanf* Rules

Input Examples

This section contains several examples. We list the data that will be input first. This allows you to cover up the function and try to formulate your own *scanf* statement.

1. `214 156 14Z`

```
        scanf("%d%d%d%c", &a, &b, &c, &d);
```

Note that a space between the `14` and the `Z` would create an error, because `%c` does not skip whitespace! To prevent this problem, put a space before the `%c` code as shown below. This will cause it to skip leading whitespace.

```
        scanf("%d%d%d %c", &a, &b, &c, &d);
```

2. `2314  15  2.14`

```
        scanf("%d %d %f",  &a, &b, &c);
```

Note the whitespace between the conversion specifications. These spaces are not necessary with numeric input, but it is a good idea to include them.

3. `14/26  25/66`

```
        scanf("%2d/%2d %2d/%2d",
               &num1, &den1, &num2, &den2);
```

Note the slashes (/) in the format string. Since they are not a part of the conversion specification, the user must enter them exactly as shown or *scanf* will stop reading.

4. `11-25-56`

```
        scanf ("%d-%d-%d", &a, &b, &c);
```

Again, we see some required user input, this time dashes between the month, day, and year. While this is a common date format, it can cause problems. A better solution would be to prompt the user separately for the month, the day, and the year.

Common Input Errors

Each of the following examples has at least one error. Try to find it before you look at the solution. Your results may vary depending on your compiler and hardware.

1.
```
int a = 0;
scanf ("%d",   a);
printf("%d\n", a);
```

```
234                            (Input)
0                              (Output)
```

This example has no address token on the variable (`&a`). If the program runs at all, the data are read into an unidentified area in memory. What is printed is the original contents of the variable, in this case `0`.

2.
```
float a = 2.1;
scanf  ("%5.2f", &a);
printf ("%5.2f",  a);
```

```
74.35                          (Input)
2.10                           (Output)
```

This example has no precision in the input conversion specification. When *scanf* finds a precision, it stops processing and returns to the function that called it. The input variable is unchanged.

3.
```
int a;
int b;
scanf  ("%d%d%d", &a , &b);
printf ("%d %d\n", a ,  b);
```

```
5 10                            (input)
5 10                            (output)
```

This example has three conversion specifications but only two addresses. Therefore, *scanf* reads the first two values and quits because no third address if found.

4.
```
int a = 1;
int b = 2;
int c = 3;
scanf  ("%d%d",    &a , &b, &c);
printf ("%d %d\n", a ,  b,  c);
```

```
5 10 15                         (input)
5 10 3                          (output)
```

This example has only two conversion specifications, but it has three addresses. Therefore, *scanf* reads the first two values and ignores the third address. The value 15 is still in the input stream waiting to be read.

2.8 Programming Examples

In this section, we show some programming example to emphasize the ideas and concepts we have discussed about input/output.

EXAMPLE 2-2 Print "Nothing!"

Program 2-4 is a very simple program that prints "`Nothing!`"

PROGRAM 2-4 A Program That Prints "Nothing!"

```c
/* Prints the message "Nothing!".
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   printf("This program prints\n\n\t\"Nothing!\"");
   return 0;
}  // main
```

continued

PROGRAM 2-4 A Program That Prints "Nothing!" *(continued)*

```
Results:
This program prints

    "Nothing!"
```

EXAMPLE 2-3 Print Boolean Constants

Program 2-5 demonstrates printing Boolean values. As the program shows, however, while a Boolean literal contains either *true* or *false*, when it is printed, it is printed as 0 or 1. This is because there is no conversion code for Boolean. To print it, we must use the integer type, which prints its stored value, 0 or 1.

PROGRAM 2-5 Demonstrate Printing Boolean Constants

```
/* Demonstrate printing Boolean constants.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
// Local Declarations
   bool x = true;
   bool y = false;

// Statements
   printf ("The Boolean values are: %d %d\n", x, y);
   return 0;
}  // main
```

```
Results:
The Boolean values are: 1 0
```

EXAMPLE 2-4 Print Character Values

Program 2-6 demonstrates that all characters are stored in the computer as integers. We define some character variables and initialize them with values, and then we print them as integers. As you study the output, note that the ASCII values of the characters are printed. The program also shows the value of some nonprintable characters. All values can be verified by referring to Appendix A.

PROGRAM 2-6 Print Value of Selected Characters

```
/* Display the decimal value of selected characters,
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char A         = 'A';
   char a         = 'a';
   char B         = 'B';
   char b         = 'b';
   char Zed       = 'Z';
   char zed       = 'z';
   char zero      = '0';
   char eight     = '8';
   char NL        = '\n';        // newline
   char HT        = '\t';        // horizontal tab
   char VT        = '\v';        // vertical tab
   char SP        = ' ';         // blank or space
   char BEL       = '\a';        // alert (bell)
   char dblQuote  = '"';         // double quote
   char backSlash = '\\';        // backslash itself
   char oneQuote  = '\'';        // single quote itself

// Statements
   printf("ASCII for char 'A'  is: %d\n",  A);
   printf("ASCII for char 'a'  is: %d\n",  a);
   printf("ASCII for char 'B'  is: %d\n",  B);
   printf("ASCII for char 'b'  is: %d\n",  b);
   printf("ASCII for char 'Z'  is: %d\n",  Zed);
   printf("ASCII for char 'z'  is: %d\n",  zed);
   printf("ASCII for char '0'  is: %d\n",  zero);
   printf("ASCII for char '8'  is: %d\n",  eight);
   printf("ASCII for char '\\n' is: %d\n", NL);
   printf("ASCII for char '\\t' is: %d\n", HT);
   printf("ASCII for char '\\v' is: %d\n", VT);
   printf("ASCII for char ' '  is: %d\n",  SP);
   printf("ASCII for char '\\a' is: %d\n", BEL);
   printf("ASCII for char '\"'  is: %d\n", dblQuote);
   printf("ASCII for char '\\'  is: %d\n", backSlash);
   printf("ASCII for char '\''  is: %d\n", oneQuote);

```

continued

PROGRAM 2-6 Print Value of Selected Characters *(continued)*

```c
   return 0;
}  // main
```

```
Results:
ASCII for character 'A'  is: 65
ASCII for character 'a'  is: 97
ASCII for character 'B'  is: 66
ASCII for character 'b'  is: 98
ASCII for character 'Z'  is: 90
ASCII for character 'z'  is: 122
ASCII for character '0'  is: 48
ASCII for character '8'  is: 56
ASCII for character '\n' is: 10
ASCII for character '\t' is: 9
ASCII for character '\v' is: 11
ASCII for character ' '  is: 32
ASCII for character '\a' is: 7
ASCII for character '"'  is: 34
ASCII for character '\'  is: 92
ASCII for character '''  is: 39
```

EXAMPLE 2-5 Define Constants

Let's write a program that calculates the area and circumference of a circle using a preprocessor-defined constant for π. Although we haven't shown you how to make calculations in C, if you know algebra you will have no problem reading the code in Program 2-7.

PROGRAM 2-7 Calculate a Circle's Area and Circumference

```c
/* This program calculates the area and circumference
   of a circle using PI as a defined constant.
      Written by:
      Date:
*/
#include <stdio.h>
#define PI  3.1416

int main (void)
{
// Local Declarations
   float circ;
   float area;
   float radius;
```

continued

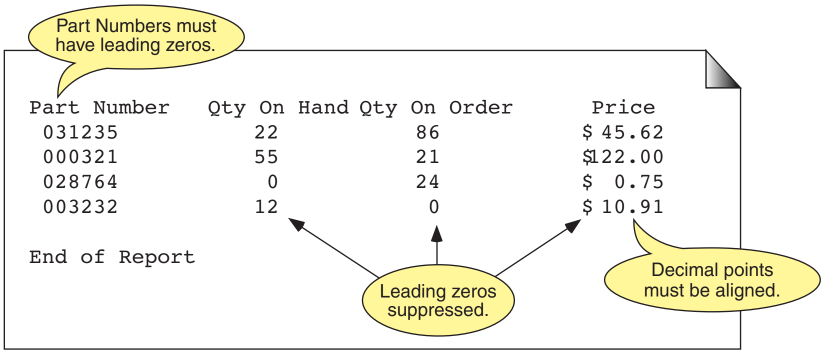
PROGRAM 2-7 Calculate a Circle's Area and Circumference *(continued)*

```c

// Statements
   printf("\nPlease enter the value of the radius: ");
   scanf("%f", &radius);

   circ  = 2  * PI     * radius;
   area  = PI * radius * radius;

   printf("\nRadius is :        %10.2f", radius);
   printf("\nCircumference is : %10.2f", circ);
   printf("\nArea is :          %10.2f", area);

   return 0;
}  // main
```

```
Results:
Please enter the value of the radius: 23

Radius is :               23.00
Circumference is :       144.51
Area is :               1661.91
```

EXAMPLE 2-6 Print a Report

You are assigned to a new project that is currently being designed. To give the customer an idea of what a proposed report might look like, the project leader has asked you to write a small program to print a sample. The specifications for the report are shown in Figure 2-22, and the code is shown in Program 2-8.



```
Part Number   Qty On Hand Qty On Order      Price
 031235           22           86          $ 45.62
 000321           55           21          $122.00
 028764            0           24          $  0.75
 003232           12            0          $ 10.91

End of Report
```

Part Numbers must have leading zeros.
Leading zeros suppressed.
Decimal points must be aligned.

FIGURE 2-22 Output Specifications for Inventory Report

The report contains four fields: a part number, which must be printed with leading zeros; the current quantity on hand; the current quantity on order; and the price of the item, printed to two decimal points. All data should be aligned in columns with captions indicating the type of data in each column. The report should be closed with an "End of Report" message.

PROGRAM 2-8 A Sample Inventory Report

```c
/* This program will print four lines of inventory data
   on an inventory report to give the user an idea of
   what a new report will look like. Since this is not
   a real report, no input is required. The data are
   all specified as constants
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Statements
   // Print captions
   printf("\tPart Number\tQty On Hand");
   printf("\tQty On Order\tPrice\n");

   // Print data
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           31235, 22, 86, 45.62);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           321, 55, 21, 122.);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           28764, 0, 24, .75);
   printf("\t %06d\t\t%7d\t\t%7d\t\t $%7.2f\n",
           3232, 12, 0, 10.91);

   // Print end message
   printf("\n\tEnd of Report\n");
   return 0;
}  // main
```

Program 2-8 Analysis There are a few things about Program 2-8 that you should note. First, it is fully documented. Professional programmers often ignore documentation on "one-time-only" programs, thinking they will throw them away, only to find that they end up using them over and over. It only takes a few minutes to document a program, and it is always time well spent. If nothing else, it helps clarify the program in your mind.

Next, look carefully at the formatting for the print statements. Spacing is controlled by a combination of tabs and format code widths. The double spacing for the end of report message is controlled by placing a newline command (`\n`) at the beginning of the message in Statement 29.

Finally, note that the program concludes with a return statement that informs the operating system that it concluded successfully. Attention to details, even in small programs, is the sign of a good programmer.

EXAMPLE 2-7 Printing The Attributes of a Complex Number

A complex number is made of two components: a real part and an imaginary part. In mathematics, it can be represented as a vector with two components. The real part is the projection of the vector on the horizontal axis (x) and the imaginary part is the projection of the vector on the vertical axis (y). In C, we use complex number and a predefined library function to print the real and imaginary values. We can also find the length of the vector, which is the absolute value of the complex number and the angle of the vector, which is the argument of the vector. These four attributes are shown in Figure 2-23.

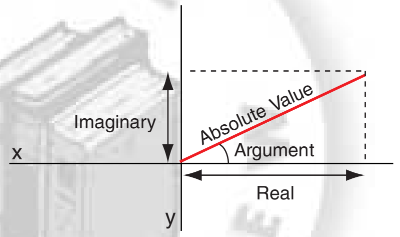


FIGURE 2-23 Complex Number Attributes

As the figure shows, the absolute value of the complex a + b * I can be found as $(a+b)^{1/2}$. The argument can be found as arctan (b/a). The conjugate of a complex number is another complex number defined as a – b * I.

Program 2-9 shows how we print the different attributes of a complex number using the predefined functions *creal*, *cimag*, *cabs*, and *carg*.

PROGRAM 2-9 Print Complex Number Attributes

```
/* Print attributes of a complex number.
      Written by:
      Date
*/
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
```

continued

PROGRAM 2-9 Print Complex Number Attributes *(continued)*

```
{
// Local Declarations
   double complex x = 4 + 4 * I;
   double complex xc;

// Statements
   xc = conj (x);
   printf("%f %f %f %f\n", creal(x), cimag(x),
                           cabs(x),  carg(x));

   printf("%f %f %f %f\n", creal(xc), cimag(xc),
                           cabs(xc),  carg(xc));
   return 0;
}  // main
```

```
Results:
4.000000 4.000000 5.656854 0.785398
4.000000 -4.000000 5.656854 -0.785398
```

EXAMPLE 2-8 Mathematics with Complex Numbers.

C allows us to add, subtract, multiply, and divide two complex numbers using the same operators (+, -, *, /) that we use for real numbers. Program 2-10 demonstrates the arithmetic use of operators with complex numbers.

PROGRAM 2-10 Complex Number Arithmetic

```
/* Demonstrate complex number arithmetic.
      Written by:
      Date:
*/
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main (void)
{
// Local Declarations
   double complex x = 3 + 4 * I;
   double complex y = 3 - 4 * I;
   double complex sum;
   double complex dif;
   double complex mul;
   double complex div;
```

continued

PROGRAM 2-10 Complex Number Arithmetic *(continued)*

```

// Statements
   sum = x + y;
   dif = x - y;
   mul = x * y;
   div = x / y;

   printf("%f %f %f %f\n", creal(sum), cimag(sum),
                           cabs(sum),  carg(sum));
   printf("%f %f %f %f\n", creal(dif), cimag(dif),
                           cabs(dif),  carg(dif));
   printf("%f %f %f %f\n", creal(mul), cimag(mul),
                           cabs(mul),  carg(mul));
   printf("%f %f %f %f\n", creal(div), cimag(div),
                           cabs(div),  carg(div));
   return 0;
}  // main
```

```
Results:
6.000000 0.000000 6.000000 0.000000
0.000000 8.000000 8.000000 1.570796
25.000000 0.000000 25.000000 0.000000
-0.280000 0.960000 1.000000 1.854590
```

2.9 Software Engineering

Although this chapter introduces only a few programming concepts, there is still much to be said from a software engineering point of view. We will discuss the concepts of program documentation, data naming, and data hiding.

Program Documentation

There are two levels of program documentation. The first is the general documentation at the start of the program. The second level is found within each function.

General Documentation

Program 2-11 shows what we recommend for program documentation. Each program should start with a general description of the program. Following the general description is the name of the author and the date the program was written. Following the date is the program's change history, which documents the reason and authority for all changes. For a production program whose use spans several years, the change history can become extensive.

PROGRAM 2-11 Sample of General Program Documentation

```
/* A sample of program documentation. Each program
   starts with a general description of the program.
   Often, this description can be taken from the
   requirements specification given to the programmer.
     Written by: original author
     Date:       Date first released to production
   Change History:
     <date> Included in this documentation is a short
            description of each change.
*/
```

Module Documentation

Whenever necessary, we include a brief comment for blocks of code. A block of code is much like a paragraph in a report. It contains one thought—that is, one set of statements that accomplish a specific task. Blocks of code in our program are separated by blank program lines, just as we skip blank lines between paragraphs in reports.

If the block of code is difficult, or if the logic is especially significant, then we give the reader a short—one- or two-line—description of the block's purpose and/or operation. We will provide many examples of this type of documentation throughout the text.

Sometimes a textbook suggests that each variable in a program be documented. We disagree with this approach. First, the proper location for variable documentation is in a data dictionary. A data dictionary is a system documentation tool that contains standard names, descriptions, and other information about data used in a system.

Second, good data names eliminate the need for variable comments. In fact, if you think you need to document the purpose of a variable, check your variable name. You will usually find that improving the name eliminates the need for the comment.

Data Names

Another principle of good structured programming is the use of **intelligent data names**. This means that the variable name itself should give the reader a good idea about what data it contains and maybe even an idea about how the data are used.

Although there are obvious advantages to keeping names short, the advantage is quickly lost if the names become so cryptic that they are unintelligible. We have seen programmers struggle for hours to find a bug, only to discover that the problem was the wrong variable was used. The time saved keying short, cryptic names is often lost ten- or a hundredfold in debugging time.

We have formulated several guidelines to help you construct good, intelligent data names.

1. The name should match the terminology of the user as closely as possible.

   Let's suppose that you are writing a program to calculate the area of a rectangle. Mathematicians often label the sides of a rectangle *a* and *b*, but their real names are length and width. Therefore, your program should call the sides of the rectangle `length` and `width`. These names are commonly used by anyone describing a rectangle.

2. When necessary for readability, and to separate similar variables from each other, combine terms to form a variable name.

   Suppose that you are working on a project to compute a payroll. There are many different types of taxes. Each of the different taxes should be clearly distinguished from the others by good data names. Table 2-13 shows both good and bad names for this programming situation. Most of the poor names are either too abbreviated to be meaningful (such as `ftr`) or are generic names (such as `rate`) that could apply to many different pieces of data.

Good Names		Poor Names			
ficaTaxRate	fica_tax_rate	rate	ftr	frate	fica
ficaWitholding	fica_witholding	fwh	ficaw	wh	
ficaWthldng	fica_wthldng	fcwthldng	wthldng		
ficaMax	ficaDlrMax	max	fmax		

TABLE 2-13 Examples of Good and Poor Data Names

Note the two different concepts for separating the words in a variable's name demonstrated in Table 2-13. In the first example, we capitalized the first letter of each word. In the second example, we separated the words with an underscore. Both are good techniques for making a compound name readable. If you use capitalization, keep in mind that `C` is case sensitive, so you must be careful to use the same cases for the name each time you use it.

3. Do not create variable names that are different by only one or two letters, especially if the differences are at the end of the word. Names that are too similar create confusion. On the other hand, a naming pattern makes it easier to recall the names. This is especially true when user terminology is being used. Thus, we see that the good names in Table 2-13 all start with `fica`.

4. Abbreviations, when used, should clearly indicate the word being abbreviated.

   Table 2-13 also contains several examples of good abbreviations. Whenever possible, use abbreviations created by the users. They will often have a glossary of abbreviations and acronyms that they use.

   Short words are usually not abbreviated. If they are short in the first place, they don't need to be made shorter.

5. Avoid the use of generic names.

   Generic names are programming or user jargon. For example, `count` and `sum` are both generic names. They tell you their purpose but don't give you any clue as to the type of data they are associated with. Better names would be `emplyCnt` and `ficaSum`. Programmers are especially fond of using generic names, but they tend to make the program confusing. Several of the poor names in Table 2-13 are generic.

6. Use memory constants or defined constants rather than literals for values that are hard to read or that might change from system to system.

   Some constants are nearly impossible to read. We pointed out the space earlier. If you need a space often, create a defined constant for it. Table 2-14 contains several examples of constants that are better when coded as defined constants.

#define SPACE ' '	#define BANG '!'
#define DBL_QTE '"'	#define QUOTE '\"'
#define COMMA ','	#define COLON ':'

TABLE 2-14 Examples of Defined Constants

Data Hiding

In "Structure of a C Program" in Section 2.2, we discussed the concept of global and local variables. We pointed out that anything placed before *main* was said to be in the global part of the program. With the exception of data that must be visible to other programs, no variables should be placed in this section.

One of the principles of structured programming states that the data structure should be hidden from view. The two terms you usually hear in connection with this concept are **data hiding** and **data encapsulation**. Both of these principles have as their objective protecting data from accidental destruction by parts of your program that don't require access to the data. In other words, if a part of your program doesn't require data to do its job, it shouldn't be able to see or modify the data. Until you learn to use functions in Chapter 4, however, you will not be able to provide this data-hiding capability.

Nevertheless, you should start your programming with good practices. And since our ultimate objective is good structured programming, we now formulate our first programming standard:

Programming Standard

No variables are to be placed in the global area of a program.

Any variables placed in the global area of your program—that is, before *main*—can be used and changed by every part of your program. This is undesirable and is in direct conflict with the structured programming principles of data hiding and data encapsulation.

2.10 Tips and Common Programming Errors

1. Well-structured programs use global (defined) constants but do not use global variables.
2. The function header for *main* should be complete. We recommend the following format:

```
int main (void)
```

   a. If you forget the parentheses after *main*, you will get a compile error.
   b. If you put a semicolon after the parentheses, you will get a compile error.
   c. If you misspell *main* you will not get a compile error, but you will get an error when you try to link the program. All programs must have a function named *main*.
3. If you forget to close the format string in the *scanf* or *printf* statement, you will get a compile error.
4. Using an incorrect conversion code for the data type being read or written is a run-time error. You can't read an integer with a *float* conversion code. Your program will compile with this error, but it won't run correctly.
5. Not separating read and write parameters with commas is a compile error.
6. Forgetting the comma after the format string in a read or write statement is a compile error.
7. Not terminating a block comment with a close token (`*/`) is a compile error.
8. Not including required libraries, such as *stdio.h*, at the beginning of your program is an error. Your program may compile, but the linker cannot find the required functions in the system library.
9. If you misspell the name of a function, you will get an error when you link the program. For example, if you misspell *scanf* or *printf*, your program will compile without errors, but you will get a linker error. Using the wrong case is a form of spelling error. For example, each of the following function names are different:

```
        scanf, Scanf, SCANF    printf, Printf, PRINTF
```

10. Forgetting the address operator (`&`) on a *scanf* parameter is a logic (run-time) error.
11. Do not use commas or other characters in the format string for a *scanf* statement. This will most likely lead to a run-time error when the user does not enter matching commas or characters. For example, the comma in the following statement will create a run-time problem if the user doesn't enter it exactly as coded.

```
scanf ("%d, %d", &a, &b);
```

12. Unless you specifically want to read a whitespace character, put a space before the character conversion specification in a *scanf* statement.
13. Using an address operator (`&`) with a variable in the *printf* statement is usually a run-time error.
14. Do not put a trailing whitespace at the end of a format string in *scanf*. This is a fatal run-time error.

2.11 Key Terms

address list
address operator
ASCII
binary stream
block comment
Boolean
call
character constant
character set
comment
complex type
constant
conversion code
conversion specification
data encapsulation
data hiding
declaration
definition
derived types
end of file (EOF)
escape character
flag modifier
floating-point type
format control string
function
global declaration section
header file
identifier
imaginary type
include
initializer
integral type
intelligent data name
justification
keyword
Latin character set
line comment
literal
logical data
memory constant
padding
parameter
parameter list
precision modifier
program documentation
real type
reserved word
sign flag
size modifier
standard input
standard output
statement
standard types
statement section
stream
string
string constant
syntax
text stream
token
type
type qualifier
variable
width modifier

2.12 Summary

- In 1972, Dennis Ritchie designed C at Bell Laboratories.
- In 1989, the American National Standards Institute (ANSI) approved ANSI C; in 1990, the ISO standard was approved.
- The basic component of a C program is the function.
- Every C function is made of declarations, definitions, and one or more statements.
- One and only one of the functions in a C program must be called *main*.

- To make a program more readable, use comments. A comment is a sequence of characters ignored by the compiler. C uses two types of comments: block and line. A block comment starts with the token `/*` and ends with the token `*/`. A line comment starts with the `//` token; the rest of the line is ignored.
- Identifiers are used in a language to name objects.
- C types include *void*, integral, floating point, and derived.
- A *void* type is used when C needs to define a lack of data.
- An integral type in C is further divided into Boolean, character, and integer.
  - A Boolean data type takes only two values: *true* and *false*. It is designated by the keyword *bool*.
  - A character data type uses values from the standard alphabet of the language, such as ASCII or Unicode. There are two character type sizes, *char* and *w_char*.
  - An integer data type is a number without a fraction. C uses four different integer sizes: *short int*, *int*, *long int*, and *long long int*.
- The floating-point type is further divided into real, imaginary, and complex.
  - A real number is a number with a fraction. It has three sizes: *float*, *double*, and *long double*.
  - The imaginary type represents the imaginary part of a complex number. It has three sizes, *float imaginary*, *double imaginary*, and *long double imaginary*.
  - The complex type contains a real and an imaginary part. C uses three complex sizes: *float complex*, *double complex*, and *long double complex*.
- A constant is data whose value cannot be changed.
- Constants can be coded in three different ways: as literals, as define commands, and as memory constants.
- Variables are named areas of memory used to hold data.
- Variables must be declared and defined before being used in C.
- To input data through the keyboard and to output data through the monitor, use the standard formatted input/output functions.
- *scanf* is a standard input function for inputting formatted data through the keyboard.
- *printf* is a standard output function for outputting formatted data to the monitor.
- As necessary, programs should contain comments that provide the reader with in-line documentation for blocks of code.
- Programs that use "intelligent" names are easier to read and understand.

2.13 Practice Sets

Review Questions

1. The purpose of a header file, such as *stdio.h*, is to store a program's source code.

   a. True
   b. False

2. Any valid printable ASCII character can be used in an identifier.

   a. True
   b. False

3. The C standard function that receives data from the keyboard is *printf*.

   a. True
   b. False

4. Which of the following statements about the structure of a C program is false?

   a. A C program starts with a global declaration section.
   b. Declaration sections contain instructions to the computer.
   c. Every program must have at least one function.
   d. One and only one function may be named *main*.
   e. Within each function there is a local declaration section.

5. Which of the following statements about block comments is false?

   a. Comments are internal documentation for programmers.
   b. Comments are used by the preprocessor to help format the program.
   c. Comments begin with a `/*` token.
   d. Comments cannot be nested.
   e. Comments end with a `*/` token.

6. Which of the following identifiers is not valid?

   a. `_option`
   b. `amount`
   c. `sales_amount`
   d. `salesAmount`
   e. `$salesAmount`

7. Which of the following is not a data type?

   a. *char*
   b. *float*
   c. *int*
   d. *logical*
   e. *void*

8. The code that establishes the original value for a variable is known as a(n):
   - a. assignment
   - b. constant
   - c. initializer
   - d. originator
   - e. value
9. Which of the following statements about a constant is true?
   - a. Character constants are coded using double quotes (`"`).
   - b. It is impossible to tell the computer that a constant should be a *float* or a *long double*.
   - c. Like variables, constants have a type and may be named.
   - d. Only integer values can be used in a constant.
   - e. The value of a constant may be changed during a program's execution.
10. The ______________ conversion specification is used to read or write a short integer.
    - a. `%c`
    - b. `%d`
    - c. `%f`
    - d. `%hd`
    - e. `%lf`
11. To print data left justified, you would use a ______________ in the conversion specification.
    - a. flag
    - b. precision
    - c. size
    - d. width
    - e. width and precision
12. The ______________ function reads data from the keyboard.
    - a. *displayf*
    - b. *printf*
    - c. *read*
    - d. *scanf*
    - e. *write*
13. One of the most common errors for new programmers is forgetting to use the address operator for variables in a *scanf* statement. What is the address operator?
    - a. The address modifier (`@`) in the conversion specification
    - b. The ampersand (`&`)
    - c. The caret (`^`)
    - d. The percent (`%`)
    - e. The pound sign (`#`)

Exercises

14. Which of the following is *not* a character constant in C?
    - a. `'C'`
    - b. `'bb'`
    - c. `"C"`
    - d. `'?'`
    - e. `' '`

15. Which of the following is *not* an integer constant in C?
    - a. `-320`
    - b. `+45`
    - c. `-31.80`
    - d. `1456`
    - e. `2,456`

16. Which of the following is *not* a floating-point constant in C?
    - a. `45.6`
    - b. `-14.05`
    - c. `'a'`
    - d. `pi`
    - e. `40`

17. What is the type of each of the following constants?
    - a. `15`
    - b. `-14.24`
    - c. `'b'`
    - d. `"1"`
    - e. `"16"`

18. Which of the following is *not* a valid identifier in C?
    - a. A3
    - b. `4A`
    - c. `if`
    - d. `IF`
    - e. `tax-rate`

19. What is the type of each of the following constants?
    - a. `"7"`
    - b. `3`
    - c. `"3.14159"`
    - d. `'2'`
    - e. `5.1`

20. What is the type of each of the following constants?
    - a. `"Hello"`
    - b. `15L`
    - c. `8.5L`
    - d. `8.5f`
    - e. `'\a'`

21. Which of the following identifiers are valid and which are invalid? Explain your answer.

a. `num`
b. `num2`
c. `2dNum`
d. `2d_num`
e. `num#2`

22. Which of the following identifiers are valid and which are invalid? Explain your answer.

a. `num-2`
b. `num 2`
c. `num_2`
d. `_num2`
e. `_num_2`

23. What is output from the following program fragment? To show your output, draw a grid of at least 8 lines with at least 15 characters per line.

```
// Local Declarations
int   x = 10;
char  w = 'Y';
float z = 5.1234;

// Statements
printf("\nFirst\nExample\n:");
printf("%5d\n, w is %c\n", x, w);
printf("\nz is %8.2f\n", z);
```

24. Find any errors in the following program.

```
// This program does nothing
int main
{
 return 0;
}
```

25. Find any errors in the following program.

```
#include (stdio.h)
int main (void)
{
    print ("Hello World");
    return 0;
{
```

26. Find any errors in the following program.

```
include <stdio>
int main (void)
{
 printf('We are to learn correct');
 printf('C language here');
 return 0;
} // main
```

27. Find any errors in the following program.

```
/* This is a program with some errors
   in it to be corrected.
*/
int main (void)
{
// Local Declarations
   integer        a;
   floating-point b;
   character      c;

// Statements
   printf("The end of the program.");
   return 0;
}  // main
```

28. Find any errors in the following program.

```
/* This is another program with some
   errors in it to be corrected.
*/
int main (void)
{
// Local Declarations
   a  int;
   b  float, double;
   c, d char;
// Statements
   printf("The end of the program.");
   return 0;
} // main
```

29. Find any errors in the following program.

```
/* This is the last program to be
   corrected in these exercises.
*/
```

continued

```
int main (void)
{
// Local Declarations
   a  int;
   b : c : d char;
   d , e, f double float;
// Statements
printf("The end of the program.");
return 0;
} // main
```

Problems

30. Code the variable declarations for each of the following:

    a. a character variable named option
    b. an integer variable, sum, initialized to 0
    c. a floating-point variable, product, initialized to 1

31. Code the variable declarations for each of the following:

    a. a short integer variable named code
    b. a constant named salesTax initialized to .0825
    c. a floating-point named sum of size double initialized to 0

32. Write a statement to print the following line. Assume the total value is contained in a variable named cost.

```
The sales total is: $    172.53
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

33. Write a program that uses four print statements to print the pattern of asterisks shown below.

```
******
******
******
******
```

34. Write a program that uses four print statements to print the pattern of asterisks shown below.

```
*
**
***
****
```

35. Write a program that uses defined constants for the vowels in the alphabet and memory constants for the even decimal digits (0, 2, 4, 6, 8). It then prints the following three lines using literal constants for the odd digits.

```
a  e  i  o  u
0  2  4  6  8
1  3  5  7  9
```

36. Write a program that defines five integer variables and initializes them to 1, 10, 100, 1000, and 10000. It then prints them on a single line separated by space characters using the decimal conversion code (`%d`), and on the next line with the float conversion code (`%f`). Note the differences between the results. How do you explain them?

37. Write a program that prompts the user to enter a quantity and a cost. The values are to be read into an integer named `quantity` and a float named `unitPrice`. Define the variables, and use only *one statement* to read the values. After reading the values, skip one line and print each value, with an appropriate name, on a separate line.

38. Write a program that prompts the user to enter an integer and then prints the integer first as a character, then as a decimal, and finally as a float. Use separate print statements. A sample run is shown below.

```
The number as a character: K
The number as a decimal  : 75
The number as a float    : 0.000000
```

Projects

39. Write a C program using *printf* statements to print the three first letters of your first name in big blocks. This program does not read anything from the keyboard. Each letter is formed using seven rows and five columns using the letter itself. For example, the letter *B* is formed using 17 `B`'s, as shown below as part of the initials BEF.

```
BBB    EEEEE  FFFFF
B   B  E      F
B   B  E      F
BBB    EEE    FFF
B   B  E      F
B   B  E      F
BBB    EEEEE  F
```

This is just an example. Your program must print the first three letters of your first name. Design your *printf* statements carefully to create enough blank lines at the beginning and end to make your initials readable. Use comments in your program to enhance readability as shown in this chapter.

40. Write a program that reads a character, an integer, and a floating-point number. It then prints the character, first using a character format specification (`%c`) and then using an integer specification (`%d`). After printing the character, it prints the integer and floating-point numbers on separate lines. Be sure to provide complete instructions (prompts) for the user.

41. Write a program that prompts the user to enter three numbers and then prints them vertically (each on one line), first forward and then reversed (the last one first), as shown in the following design.

```
Please enter three numbers: 15 35 72
Your numbers forward:
   15
   35
   72

Your numbers reversed:
   72
   35
   15
```

42. Write a program that reads 10 integers and prints the first and the last on one line, the second and the ninth on the next line, the third and the seventh on the next line, and so forth. Sample input and the results are shown below.

```
Please enter 10 numbers:
10 31 2 73 24 65 6 87 18 9

Your numbers are:
   10   9
   31  18
    2  87
   73   6
   24  65
```

43. Write a program that reads nine integers and prints them three in a line separated by commas as shown below.

```
Input:
   10 31 2 73 24 65 6 87 18
Output
   10, 31,  2
   73, 24, 65
    6, 87, 18
```