The McGraw-Hill Companies

PROGRAMMING IN ANSI C

4E

E BALAGURUSAMY

Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

Eighth reprint 2008
DQLLRDRXRCYDR

ISBN (13 digits): 978-0-07-064822-7
ISBN (10 digits): 0-07-064822-0

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Asst. Sponsoring Editor: SEM & Tech Ed: *Shalini Jha*
General Manager: Marketing –Higher Education & School: *Michael J. Cruz*
Product Manager: SEM & Tech Ed: *Biju Ganesan*
Controller –Production: *Rajender P Ghansela*
Asst. General Manager –Production: *B L Dogra*
Senior Production Executive: *Anjali Razdan*

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi 110 063, text and cover printed at Gopsons Papers Limited, A-2 and 3, Sector 64, Noida 201301

The McGraw-Hill Companies

Contents

Preface to the Fourth Edition

C is a powerful, flexible, portable and elegantly structured programming language. Since C combines the features of high-level language with the elements of the assembler, it is suitable for both systems and applications programming. It is undoubtedly the most widely used general-purpose language today.

Since its standardization in 1989, C has undergone a series of changes and improvements in order to enhance the usefulness of the language. The version that incorporates the new features is now referred to as C99. The fourth edition of ANSI C has been thoroughly revised and enlarged not only to incorporate the numerous suggestions received both from teachers and students across the country but also to highlight the enhancements and new features added by C99.

Organization of the book

The book starts with an overview of C, which talks about the history of C, basic structure of C programs and their execution. The second chapter discusses how to declare the constants, variables and data types. The third chapter describes the built-in operators and how to build expressions using them. The fourth chapter details the input and output operations. Decision making and branching is discussed in the fifth chapter, which talks about the if-else, switch and goto statements. Further, decision making and looping is discussed in Chapter six, which covers while, do and for loops. Arrays and ordered arrangement of data elements are important to any programming language and have been covered in chapters seven and eight. Strings are also covered in Chapter eight. Chapters nine and ten are on functions, structures and unions. Pointers, perhaps the most difficult part of C to understand, is covered in Chapter eleven in the most user-friendly manner. Chapters twelve and thirteen are on file management and dynamic memory allocation respectively. Chapter fourteen deals with the preprocessor, and finally Chapter 15 is on developing a C program, which provides an insight on how to proceed with development of a program. The above organization would help the students in understanding C better if followed appropriately.

New to the edition

The content has been revised keeping the updates which have taken placed in the field of C programming and the present day syllabus needs. As always, the concept of 'learning by example' has been stressed throughout the book. Each major feature of the language is treated in depth followed by a complete program example to illustrate its use. The sample programs are meant to be both simple and educational. Two new projects are added at the end of the book for students to go through and try on their own.

Each chapter includes a section at the beginning to introduce the topic in a proper perspective. It also provides a quick look into the features that are discussed in the chapter. Wherever necessary, pictorial descriptions of concepts are included to improve clarity and to facilitate better understanding. Language tips and other special considerations are highlighted as notes wherever essential. In order to make the book more user-friendly, we have incorporated the following key features.

- ***Codes with comments*** are provided throughout the book to illustrate how the various features of the language are put together to accomplish specified tasks.
- ***Supplementary information and notes*** that complement but stand apart from the general text have been included in boxes.
- ***Guidelines*** for developing efficient C programs are given in the last chapter, together with a ***list of some common mistakes*** that a less experienced C programmer could make.
- ***Case studies*** at the end of the chapters illustrate common ways C features are put together and also show real-life applications.
- The ***Just Remember*** section at the end of the chapters lists out helpful hints and possible problem areas.
- Numerous chapter-end ***questions*** and ***exercises*** provide ample opportunities to the readers to review the concepts learned and to practice their applications.
- ***Programming projects*** discussed in the appendix give insight on how to integrate the various features of C when handling large programs.

Supplementary Material

With this revision we have tried to enhance the online learning center too. The supplementary material would include the following:

For the Instructor

- Solutions to the debugging exercises

For the Student

- Exclusive project for implementation with code, step-by-step description and user manual
- Code for the two projects *(given in the book)*
- Two mini projects
- Reading material on C

This book is designed for all those who wish to be C programmers, regardless of their past knowledge and experience in programming. It explains in a simple and easy-to-understand style the what, why and how of programming with ANSI C.

E Balagurusamy

1

Overview of C

1.1 HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as "*traditional C*". The language became more popular after publication of the book '*The C Programming Language*' by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 1.1.

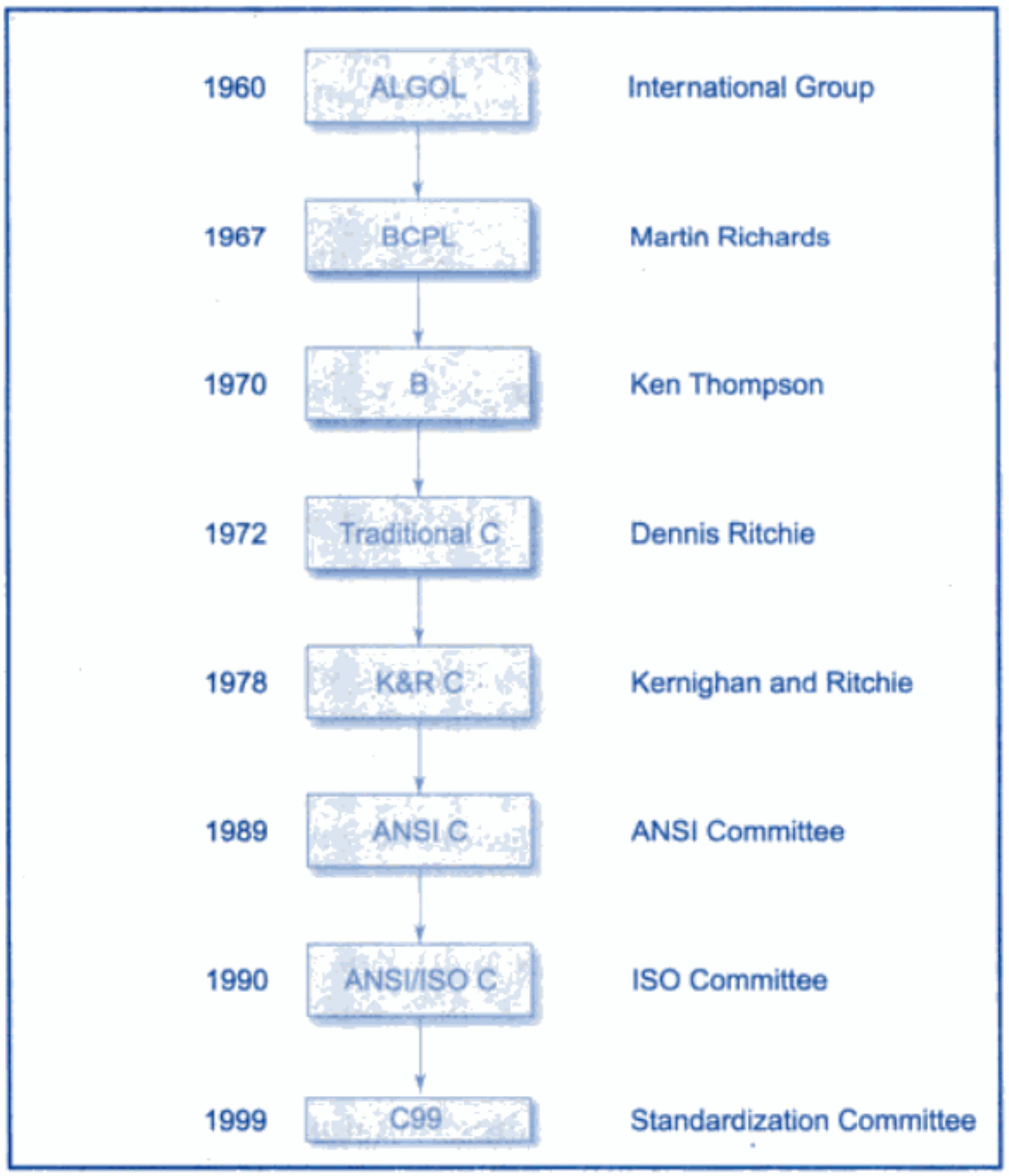


Fig. 1.1 *History of ANSI C*

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

1.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they work.

1.3 SAMPLE PROGRAM 1: PRINTING A MESSAGE

Consider a very simple program given in Fig. 1.2.

```
main( )
{
/*.....printing begins.....*/
    printf("I see, I remember");
/*.....printing ends.....*/
}
```

Fig. 1.2 *A program to print one line of text*

This program when executed will produce the following output:

```
I see, I remember
```

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main( )** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one main* function. If we use more than one **main** function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 9).

The opening brace "{ " in the second line marks the beginning of the function **main** and the closing brace "}" in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with /* and ending with */ are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between /* and */ is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—"but never in the middle of a word ".

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/* = = = =/* = = = = */ = = = = */
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf( )** function, the only executable statement of the program

```
printf("I see, I remember");
```

printf is a predefined standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter. The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

```
I see,
I remember!
```

This can be achieved by adding another **printf** function as shown below:

```
printf("I see, \n");
printf("I remember !");
```

The information contained between the parentheses is called the *argument* of the function. This argument of the first **printf** function is " I see, \n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character **\n** causes the string "I remember !" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

```
I see, I remember !
```

This is similar to the output of the program in Fig. 1.2. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline character at appropriate places. For example, the statement

```
printf("I see,\n I remember !");
```

will output

```
I see,
 I remember !
```

while the statement

```
printf( "I\n.. see,\n… … … I\n… … … remember !");
```

will print out

```
I
.. see,
… … … I
… … … remember !
```

NOTE: Some authors recommend the inclusion of the statement

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

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions ***printf*** and ***scanf*** which have been defined as a part of the C language. See Chapter 4 for more on input and output functions.

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER"

The above example that printed **I see, I remember** is one of the simplest programs. Figure 1.3 highlights the general format of such simple programs. All C programs need a **main** function.

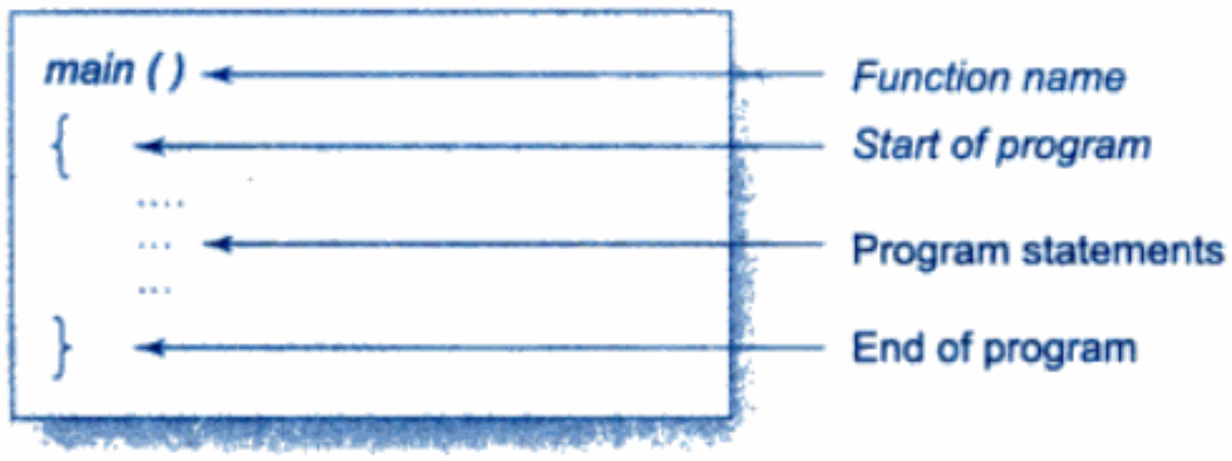


Fig. 1.3 *Format of simple C programs*

The main Function

The main is a part of every C program. C permits different forms of main state ment. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword **void** inside the parentheses. We may also specify the keyword **int** or **void** before the word **main**. The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the operating system. When **int** is specified, the last statement in the program must be "return 0". For the sake of simplicity, we use the first form in our programs.

1.4 SAMPLE PROGRAM 2: ADDING TWO NUMBERS

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 1.4.

```
/* Programm ADDITION                                    line-1 */
/* Written by EBG                                       line-2 */
main()                                                /*  line-3 */
{                                                     /*  line-4 */
```

```
        int number;                                    /*  line-5 */
        float amount;                                  /*  line-6 */
                                                       /*  line-7 */
        number = 100;                                  /*  line-8 */
                                                       /*  line-9 */
        amount = 30.75 + 75.35;                        /* line-10 */
        printf("%d\n",number);                         /* line-11 */
        printf("%5.2f",amount);                        /* line-12 */
}                                                      /* line-13 */
```

Fig. 1.4 *Program to add two numbers*

This program when executed will produce the following output:

```
100
106.10
```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the ***variable names*** are and what ***type of data*** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int**) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig.1.4. All declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 2.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 2.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount.** The statements

```
number = 100;
amount = 30.75 + 75.35;
```

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

```
printf("%d\n", number);
```

contains two arguments. The first argument "%d" tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.

The last statement of the program

```
printf("%5.2f", amount);
```

prints out the value of **amount** in floating point format. The format specification %5.2*f* tells the compiler that the output must be in *floating point*, with five places in all and two places to the right of the decimal point.

1.5 SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 1.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 1.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

```
amount = value ;
```

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*————————— INVESTMENT PROBLEM —————————*/
#define PERIOD        10
#define PRINCIPAL   5000.00
/*————————— MAIN PROGRAM BEGINS —————————*/
main()
{ /*————————— DECLARATION STATEMENTS —————————*/
      int year;
      float amount, value, inrate;
/*————————— ASSIGNMENT STATEMENTS —————————*/
      amount = PRINCIPAL;
      inrate = 0.11;
      year = 0;
/*————————— COMPUTATION STATEMENTS —————————*/
/*————————— COMPUTATION USING While LOOP —————————*/
      while(year <= PERIOD)
      { printf("%2d   %8.2f\n",year, amount);
          value = amount + inrate * amount;
           year = year + 1;
         amount = value;
      }
/*————————— while LOOP ENDS —————————*/
}
/*————————— PROGRAM ENDS —————————*/
```

Fig. 1.5 *Program for investment problem*

Let us consider the new features introduced in this program. The second and third lines begin with **#define** instructions. A **#define** instruction defines value to a *symbolic constant* for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants **PERIOD** and **PRINCIPAL** and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

```
 0        5000.00
 1        5550.00
 2        6160.50
 3        6838.15
 4        7590.35
 5        8425.29
 6        9352.07
 7       10380.00
 8       11522.69
 9       12790.00
10       14197.11
```

Fig. 1.6 *Output of the investment program*

The #define Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** instructions are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Preprocessor directives are discussed in Chapter 14.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

```
PRINCIPAL = 10000.00;
```

is illegal.

The declaration section declares **year** as integer and **amount, value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

```
float amount;
float value;
float inrate;
```

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD**, the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD**. The concept and types of loops are discussed in Chapter 6.

C supports the basic four arithmetic operators (–, +, *, /) along with several others. They are discussed in Chapter 3.

1.6 SAMPLE PROGRAM 4: USE OF SUBROUTINES

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in Fig. 1.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

Figure 1.7 presents a very simple program that uses a **mul ( )** function. The program will print the following output.

Multiplication of 5 and 10 is 50

```
/*———————— PROGRAM USING FUNCTION ————————*/
int mul (int a, int b); /*—— DECLARATION ———*/
/*———————— MAIN PROGRAM BEGINS ————————*/
    main ()
    {
        int a, b, c;
        a = 5;
        b = 10;
        c = mul (a,b);

        printf ("multiplication of %d and %d is %d",a,b,c);
    }
/* ———————          MAIN PROGRAM ENDS
                    MUL() FUNCTION STARTS ————————————*/
        int mul (int x, int y)
        int p;
        {
            p = x*y;
            return(p);
        }
/* ———————————— MUL () FUNCTION ENDS ————————————*/
```

Fig. 1.7 *A program using a user-defined function*

The **mul** ( ) function multiplies the values of **x** and **y** and the result is returned to the **main** ( ) function when it is called in the statement

```
c = mul (a, b);
```

The **mul** ( ) has two *arguments* **x** and **y** that are declared as integers. The values of **a** and **b** are passed on to **x** and **y** respectively when the function **mul** ( ) is called. User-defined functions are considered in detail in Chapter 9.

1.7 SAMPLE PROGRAM 5: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as cos, sin, exp, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C **math library**. If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define,** it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

```
#include <math.h>
```

math.h is the filename containing the required function. Figure 1.8 illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20.....180 and prints out the results with headings.

```
/*------- PROGRAM USING COSINE FUNCTION -------*/
#include <math.h>
#define  PI 3.1416
#define  MAX  180

main ( )
{
     int angle;
     float x,y;

     angle = 0;
     printf(" Angle    Cos(angle)\n\n");

     while(angle <= MAX)
     {
          x = (PI/MAX)*angle;
          y = cos(x);
          printf("%15d %13.4f\n", angle, y);
          angle = angle + 10;
     }
}
```

Output

```
          Angle              Cos(angle)
              0                  1.0000
             10                  0.9848
             20                  0.9397
             30                  0.8660
```

```
 40      0.7660
 50      0.6428
 60      0.5000
 70      0.3420
 80      0.1736
 90     -0.0000
100     -0.1737
110     -0.3420
120     -0.5000
130     -0.6428
140     -0.7660
150     -0.8660
160     -0.9397
170     -0.9848
180     -1.0000
```

Fig. 1.8 *Program using a math function*

Another **#include** instruction that is often required is

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

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The #include Directive

As mentioned earlier, C programs are divided into modules or functions. Some functions are written by users, like us, and many others are stored in the C library. Library functions are grouped category-wise and stored in different files known as *header files*. If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

A list of library functions and header files containing them are given in Appendix III.

1.8 BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *state-*

ments designed to perform a *specific task*. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 1.9.

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

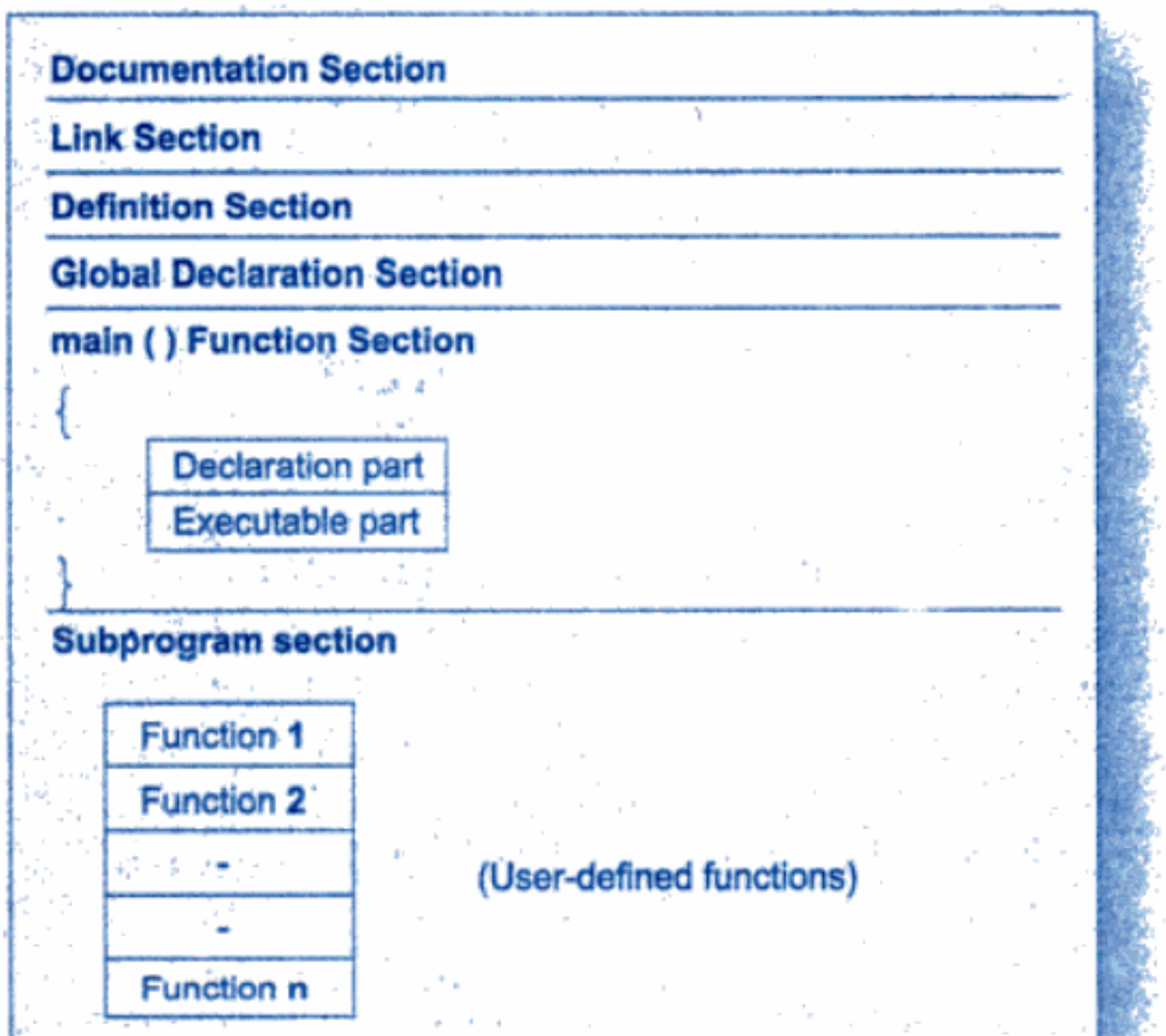


Fig. 1.9 *An overview of a C program*

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the **main** function section may be absent when they are not required.

1.9 PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a *free-form* language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 1.5.

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y + 1;
z = a + x;
```

can be written on one line as

```
a = b; x = y+1; z = a+x;
```

The program

```
main( )
{
      printf("hello C");
}
```

may be written in one line like

```
main( ) {printf("Hello C")};
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

1.10 EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.

Figure 1.10 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system*, system

commands for implementing the steps and conventions for naming *files* may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channeled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The two most popular operating systems today are UNIX (for minicomputers) and MS-DOS (for microcomputers). We shall discuss briefly the procedure to be followed in executing C programs under both these operating systems in the following sections.

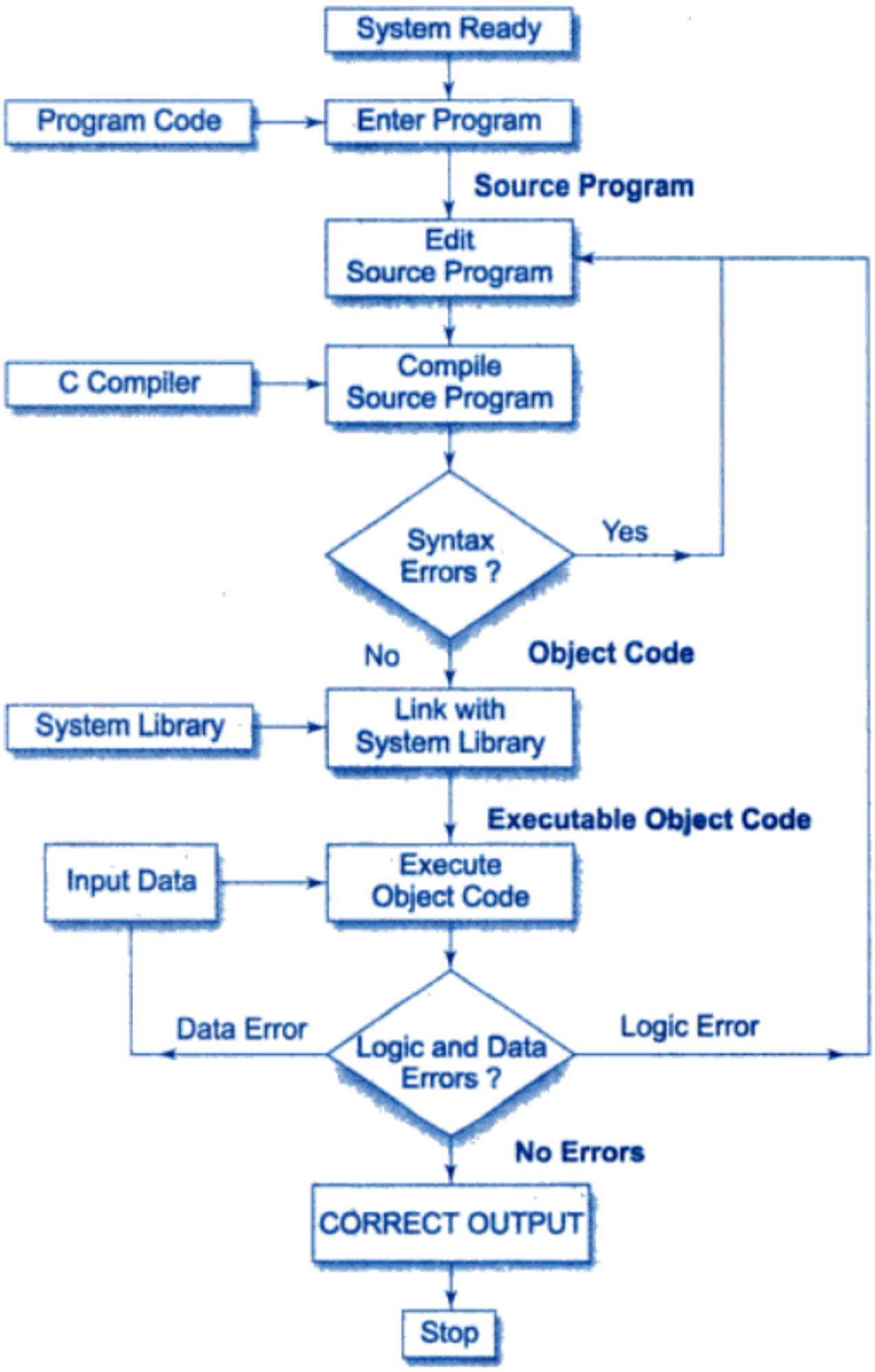


Fig. 1.10 *Process of compiling and running a C program*

1.11 UNIX SYSTEM

Creating the program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter **c**. Examples of valid file names are:

hello.c
program.c
ebg1.c

The file is created with the help of a *text editor*, either **ed** or **vi.** The command for calling the editor and creating the file is

ed *filename*

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the *source program*, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named *ebg1.c*. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

cc *ebg1.c*

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name *ebg1.o*. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using **exp()** function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the **cc** command is used.

If any mistakes in the *syntax* and *semantics* of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the *executable object code* and is stored automatically in another file named **a.out.**

Note that some systems use different compilation command for linking mathematical functions.

cc *filename* **- lm**

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

```
a.out
```

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program *logic* or *data*. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name **a.out**. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

```
mv a.out name
```

We may also achieve this by specifying an option in the cc command as follows:

```
cc -o name source-file
```

This will store the executable object code in the file name and prevent the old file **a.out** from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the **cc** command.

```
cc filename-1.c .... filename-n.c
```

These files will be separately compiled into object files called

```
filename-i.o
```

and then linked to produce an executable program file **a.out** as shown in Fig. 1.11.

It is also possible to compile each file separately and link them later. For example, the commands

```
cc -c mod1.c
cc -c mod2.c
```

will compile the source files *mod1.c* and *mod2.c* into objects files *mod1.o* and *mod2.o*. They can be linked together by the command

```
cc mod1.o mod2.o
```

we may also combine the source files and object files as follows:

```
cc mod1.c mod2.o
```

Only *mod1.c* is compiled and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

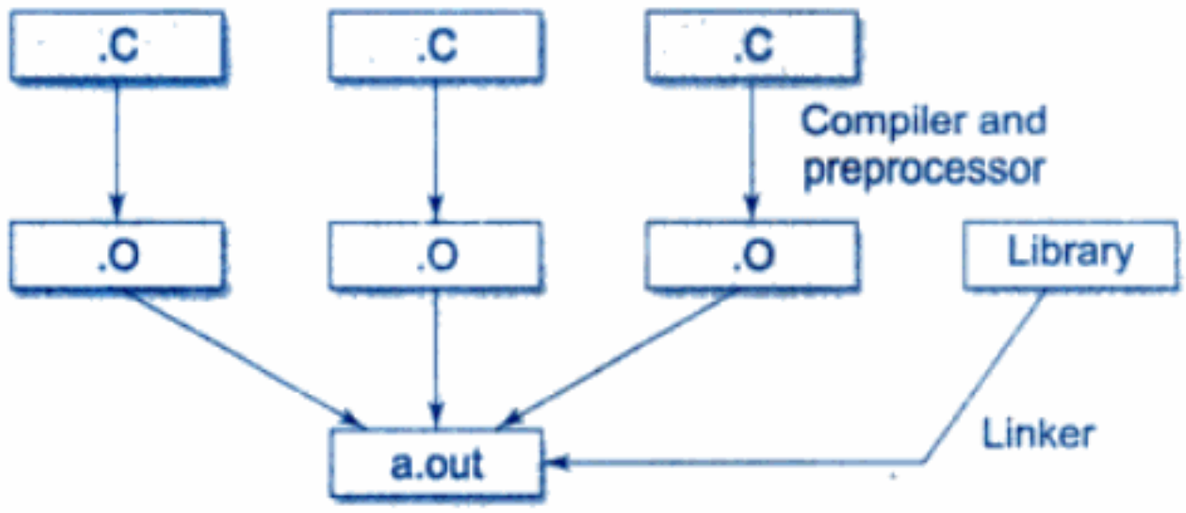


Fig. 1.11 *Compilation of multiple files*

1.12 MS-DOS SYSTEM

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c" like **program.c, pay.c,** etc. Then the command

```
MSC pay.c
```

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code.** This code is stored in another file under name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

```
LINK pay.obj
```

which generates the **executable code** with the filename **pay.exe**. Now the command

```
pay
```

would execute the program and give the results.

Just Remember

- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins.
- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace.
- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings.
- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark.
- Every program statement in a C language must end with a semicolon.
- All variables must be declared for their types before they are used in the program.
- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define.
- Compiler directives such as **define** and **include** are special instructions

to the compiler to help it compile a program. They do not end with a semicolon.

- ✍ The sign # of compiler directives must appear in the first column of the line.
- ✍ When braces are used to group statements, make sure that the opening brace has a corresponding closing brace.
- ✍ C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program.
- ✍ A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and */ appropriately.

Review Questions

1.1 State whether the following statements are *true* or *false*.
- (a) Every line in a C program should end with a semicolon.
- (b) In C language lowercase letters are significant.
- (c) Every C program ends with an END word.
- (d) **main( )** is where the program begins its execution.
- (e) A line in a program may have more than one statement.
- (f) A **printf** statement can generate only one line of output.
- (g) The closing brace of the **main( )** in a program is the logical end of the program.
- (h) The purpose of the header file such as **stdio.h** is to store the source code of a program.
- (i) Comments cause the computer to print the text enclosed between /* and */ when executed.
- (j) Syntax errors will be detected by the compiler.

1.2 Which of the following statements are *true*?
- (a) Every C program must have at least one user-defined function.
- (b) Only one function may be named **main( )**.
- (c) Declaration section contains instructions to the computer.

1.3 Which of the following statements about comments are *false*?
- (a) Use of comments reduces the speed of execution of a program.
- (b) Comments serve as internal documentation for programmers.
- (c) A comment can be inserted in the middle of a statement.
- (d) In C, we can have comments inside comments.

1.4 Fill in the blanks with appropriate words in each of the following statements.
- (a) Every program statement in a C program must end with a _____
- (b) The _____ Function is used to display the output on the screen.
- (c) The _____ header file contains mathematical functions.
- (d) The escape sequence character _____ causes the cursor to move to the next line on the screen.

1.5 Remove the semicolon at the end of the **printf** statement in the program of Fig. 1.2 and execute it. What is the output?

1.6 In the Sample Program 2, delete line-5 and execute the program. How helpful is the error message?

1.7 Modify the Sample Program 3 to display the following output:

Year	Amount
1	5500.00
2	6160.00
-	———
-	———
10	14197.11

1.8 Find errors, if any, in the following program:

```
/* A simple program
int main( )
{
     /* Does nothing */
}
```

1.9 Find errors, if any, in the following program:

```
#include (stdio.h)
void main(void)
{
     print("Hello C");
}
```

1.10 Find errors, if any, in the following program:

```
Include <math.h>
main { }
(
     FLOAT X;
     X = 2.5;
     Y = exp(x);
     Print(x,y);
)
```

1.11 Why and when do we use the **#define** directive?

1.12 Why and when do we use the **#include** directive?

1.13 What does **void main(void)** mean?

1.14 Distinguish between the following pairs:
   (a) main( ) and void main(void)
   (b) int main( ) and void main( )

1.15 Why do we need to use comments in programs?

1.16 Why is the look of a program is important?

1.17 Where are blank spaces permitted in a C program?

1.18 Describe the structure of a C program.

1.19 Describe the process of creating and executing a C program under UNIX system.

1.20 How do we implement multiple source program files?

Programming Exercises

1.1 Write a program that will print your mailing address in the following form:

First line : Name

Second line : Door No, Street
Third line : City, Pin code

1.2 Modify the above program to provide border lines to the address.

1.3 Write a program using one print statement to print the pattern of asterisks as shown below:

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

1.4 Write a program that will print the following figure using suitable characters.

1.5 Given the radius of a circle, write a program to compute and display its area. Use a symbolic constant to define the π value and assume a suitable value for radius.

1.6 Write a program to output the following multiplication table:

$5 \times 1 = 5$

$5 \times 2 = 10$

$5 \times 3 = 15$

. .

. .

$5 \times 10 = 50$

1.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form:

$20 + 10 = 30$

$20 - 10 = 10$

1.8 Given the values of three variables a, b and c, write a program to compute and display the value of x, where

$$x = \frac{a}{b-c}$$

Execute your program for the following values:

(a) a = 250, b = 85, c = 25

(b) a = 300, b = 70, c = 70

Comment on the output in each case.

1.9 Relationship between Celsius and Fahrenheit is governed by the formula

$$F = \frac{9C}{5} + 32$$

Write a program to convert the temperature

(a) from Celsius to Fahrenheit and
(b) from Fahrenheit to Celsius.

1.10 Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where a, b and c are sides of the triangle and 2S = a + b + c. Write a program to compute the area of the triangle given the values of a, b and c.

1.11 Distance between two points (x_1, y_1) and (x_2, y_2) is governed by the formula

$$D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Write a program to compute D given the coordinates of the points.

1.12 A point on the circumference of a circle whose center is (o, o) is (4,5). Write a program to compute perimeter and area of the circle. (Hint: use the formula given in the Ex. 1.11)

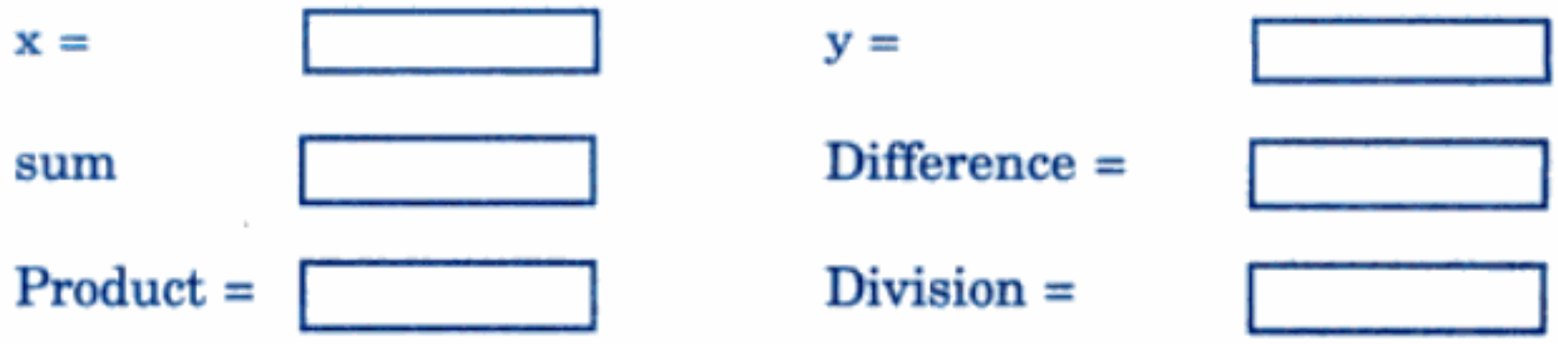
1.13 The line joining the points (2,2) and (5,6) which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle.

1.14 Write a program to display the equation of a line in the form

$$ax + by = c$$

for a = 5, b = 8 and c = 18.

1.15 Write a program to display the following simple arithmetic calculator



2

Constants, Variables, and Data Types

2.1 INTRODUCTION

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

2.2 CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in Table 2.1.

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

Trigraph Characters

Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2. For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

Table 2.1 *C Character Set*

Letters	*Digits*
Uppercase A.....Z Lowercase a.....z	All decimal digits 0 9

Special Characters

, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
: colon	– minus sign
? question mark	+ plus sign
‘ apostrophe	< opening angle bracket
“ quotation mark	(or less than sign)
! exclamation mark	> closing angle bracket
| vertical bar	(or greater than sign)
/ slash	(left parenthesis
\ backslash	) right parenthesis
~ tilde	[left bracket
_ under score	] right bracket
$ dollar sign	{ left brace
% percent sign	} right brace
	# number sign

White Spaces

Blank space
Horizontal tab
Carriage return
New line
Form feed

Table 2.2 *ANSI C Trigraph Sequences*

Trigraph sequence	*Translation*
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

2.3 C TOKENS

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 2.1. C programs are written using these tokens and the syntax of the language.

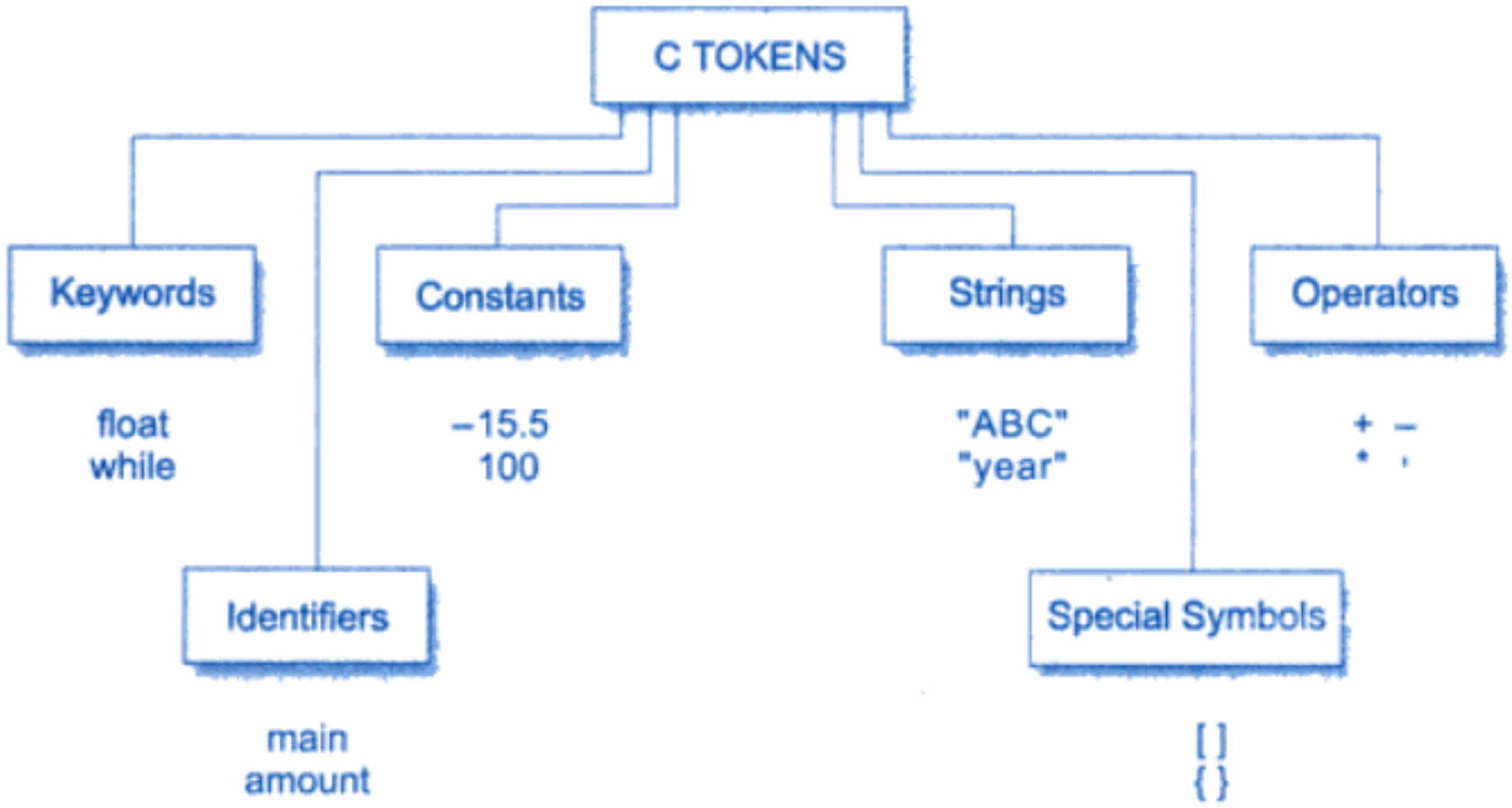


Fig. 2.1 *C tokens and examples*

2.4 KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 2.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

NOTE: C99 adds some more keywords. See the Appendix "C99 Features".

Table 2.3 *ANSI C Keywords*

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

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both

uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

2.5 CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 2.2.

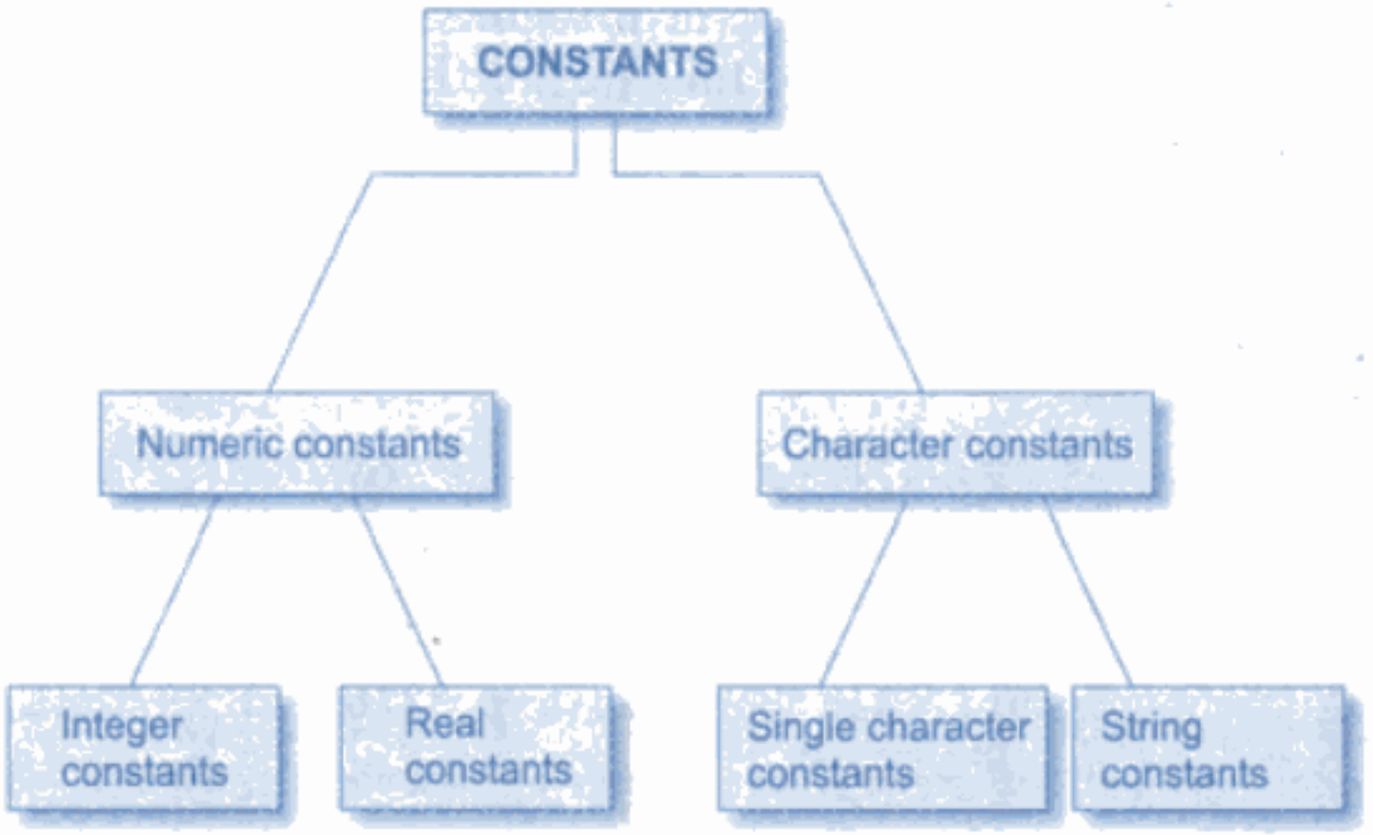


Fig. 2.2 *Basic types of C constants*

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 –321 0 654321 +78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000

are illegal numbers.

Note: ANSI C supports *unary plus* which was not defined earlier.

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter A through F represent the numbers 10 through 15. Following are the examples of valid hex integers:

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U,L and UL to the constants. Examples:

56789U	or 56789u	(unsigned integer)
987612347UL	or 98761234ul	(unsigned long integer)
9876543L	or 9876543l	(long integer)

The concept of unsigned and long integers are discussed in detail in Section 2.7.

Example 2.1 Representation of integer constants on a 16-bit computer.

The program in Fig.2.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig. 2.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```
Program
        main()
        {
            printf("Integer values\n\n");
            printf("%d %d %d\n", 32767,32767+1,32767+10);
            printf("\n");
            printf("Long integer values\n\n");
            printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
        }
  Output
        Integer values
        32767 -32768 -32759
        Long integer values
        32767 32768 32777
```

Fig. 2.3 *Representation of integer constants on 16-bit machine*

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

0.0083 –0.75 435.36 +247.0

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

215. .95 –.71 +.5

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific) notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus sign*. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating-point constants are:

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to –3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 2.4.

Table 2.4 *Examples of Numeric Constants*

Constant	*Valid ?*	*Remarks*
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quote marks. Example of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the *number 5*. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

```
printf("%d", 'a');
```

would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", '97');
```

would output the letter 'a'. ASCII values for all characters are given in Appendix II.

Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in Chapter 8.

String Constants

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs. Manipulation of character strings are considered in detail in Chapter 8.

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 2.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

Table 2.5 *Backslash Character Constants*

Constant	*Meaning*
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

2.6 VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution. In Chapter 1, we used several variables. For instance, we used the variable **amount** in Sample Program 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average
height
Total
Counter_1
class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers. (In C99, at least 63 characters are significant.)
3. Uppercase and lowercase are significant. That is, the varible **Total** is not the same as **total** or **TOTAL**.
4. It should not be a keyword.
5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 2.6.

Table 2.6 *Examples of Variable Names*

Variable name	*Valid?*	*Remark*
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

If only the first eight characters are recognized by a compiler, then the two names

average_height
average_weight

mean the same thing to the computer. Such names can be rewritten as

avg_height and avg_weight

or

ht_average and wt_average

without changing their meanings.

2.7 DATA TYPES

C language is rich in its *data types*. Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

All C compilers support five fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), double-precision floating point (**double**) and **void.** Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 2.4. The range of the basic four types are given in Table 2.7. We discuss briefly each one of them in this section.

NOTE: C99 adds three more data types, namely **_Bool**, **_Complex**, and **_Imaginary**. See the Appendix "C99 Features".

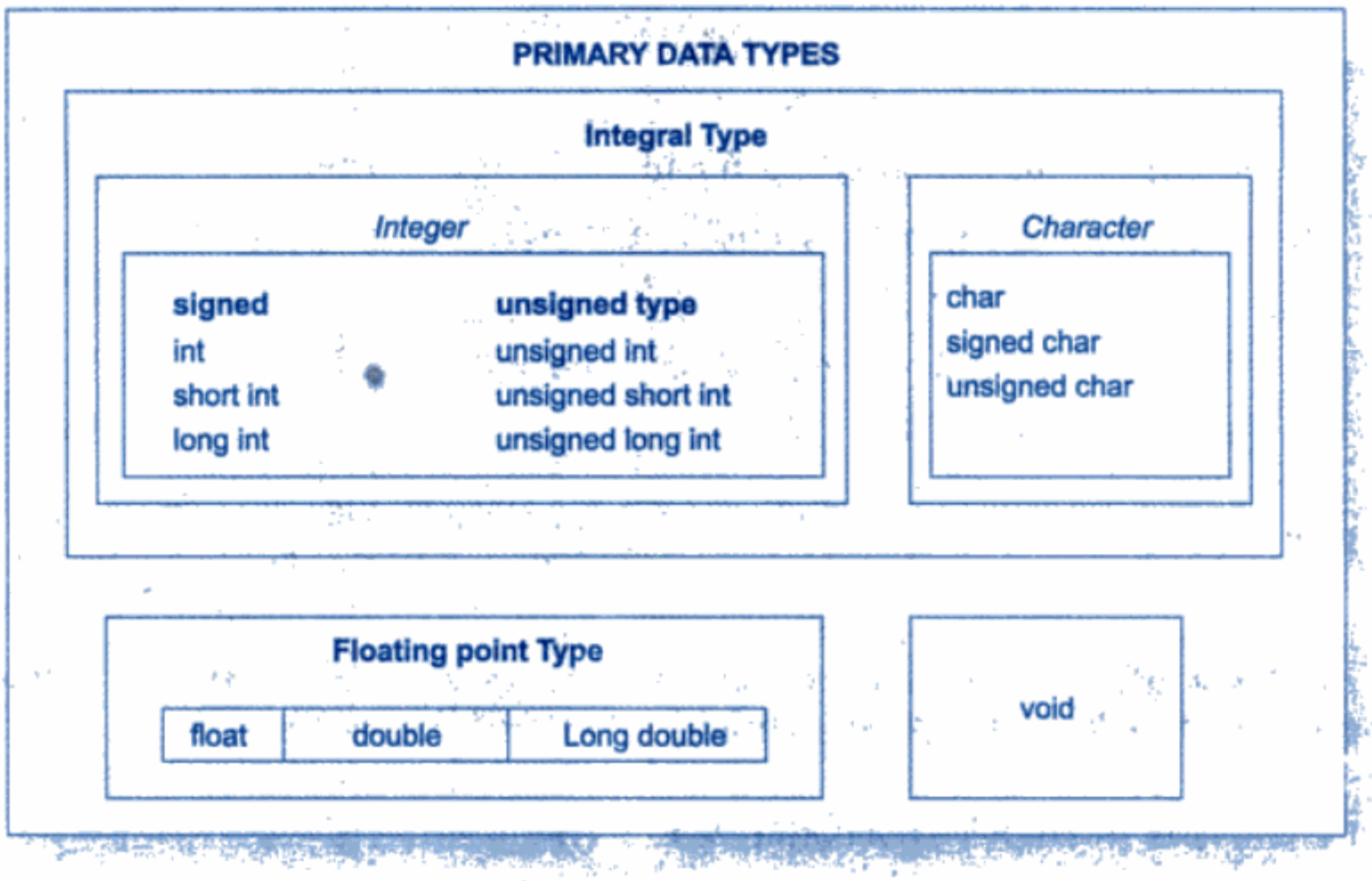


Fig. 2.4 *Primary data types in C*

Table 2.7 *Size and Range of Basic Data Types on 16-bit Machines*

Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range –32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **unsigned** forms. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 2.5. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int** number uses. Unlike signed

integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

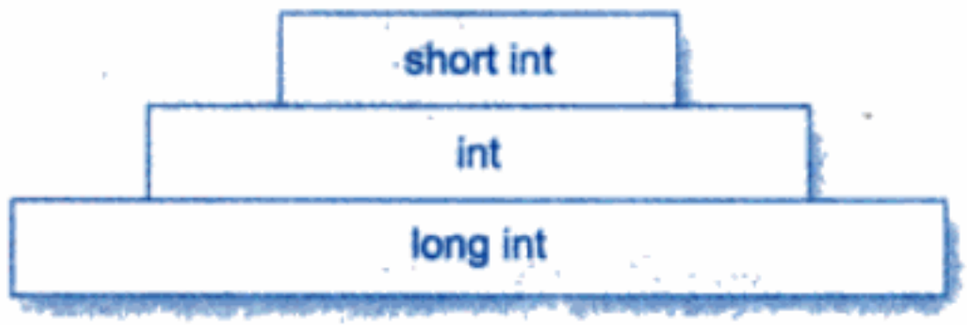


Fig. 2.5 *Integer types*

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 2.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

NOTE: C99 allows **long long** integer types. See the Appendix "C99 Features".

Table 2.8 *Size and Range of Data Types on a 16-bit Machine*

Type	*Size (bits)*	*Range*
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that double type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits. The relationship among floating types is illustrated in Fig. 2.6.

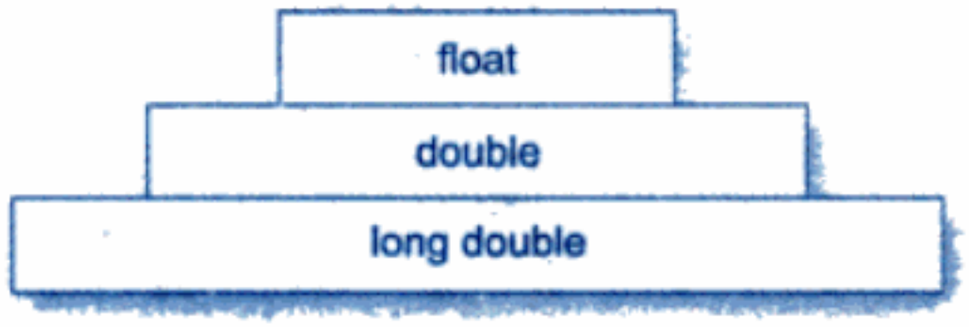


Fig. 2.6 *Floating-point types*

Void Types

The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

Character Types

A single character can be defined as a **character(char)** type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 255, **signed chars** have values from –128 to 127.

2.8 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

data-type v1,v2,....vn ;

v1, v2, ...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

```
int count;
int number, total;
double ratio;
```

int and **double** are the keywords to represent integer type and real type data values respectively. Table 2.9 shows various data types and their keyword equivalents.

Table 2.9 *Data Types and Their Keywords*

Data type	*Keyword equivalent*
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program segment given in Fig. 2.7 illustrates declaration of variables. **main()** is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

Note: C99 permits declaration of variables at any point within a function or block, prior to their use.

```
main() /*.........Program Name.......................... */
{
  /*................Declaration.....................*/
    float       x, y;
    int         code;
    short int   count;
    long int    amount;
    double      deviation;
    unsigned    n;
    char        c;
  /*................Computation.....................  */
     . . . .
     . . . .
     . . . .
} /*.............Program  ends.....................*/
```

Fig. 2.7 *Declaration of variables*

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int.** If we want to declare a character variable as unsigned, then we must do so using both the terms like **unsigned char**.

Default values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
–222	int	–222
45678U	unsigned int	45,678
–56789L	long int	–56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double**, we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below:

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
–1.2f	float	–1.2
1.23456789L	long double	1.23456789

User-Defined Type Declaration

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables . It takes the general form:

typedef ***type*** **identifier;**

Where *type* refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is 'new' only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

```
typedef int units;
typedef float marks;
```

Here, **units** symbolizes **int** and **marks** symbolizes **float.** They can be later used to declare variables as follows:

```
units batch1, batch2;
marks name1[50], name2[50];
```

batch1 and batch2 are inclared as **int** variable and name1[50] and name2[50] are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

enum identifier {*value1, value2, ... valuen*};

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this 'new' type as below:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, ... vn can only have one of the values *value1, value2, ... valuen*. The assignments of the following types are valid:

```
v1 = value3;
v5 = value1;
```

An example:

```
enum day {Monday,Tuesday, ... Sunday};
enum day week_st, week_end;
week_st = Monday;
week_end = Friday;
if(week_st == Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Monday = 1, Tuesday, ... Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement. Example:

```
enum day {Monday, ... Sunday} week_st, week_end;
```

2.9 DECLARATION OF STORAGE CLASS

Variables in C can have not only *data type* but also *storage class* that provides information about their location and visibility. The storage class decides the portion of the program within which the variables are recognized. Consider the following example:

```
   /* Example of storage classes */
int  m;

main()
{
   int i;
   float balance;
   ....
```

```
    ....
    function1();
}
function1()
{
    int i;
    float sum;
    ....
    ....
}
```

The variable **m** which has been declared before the **main** is called *global* variable. It can be used in all the functions in the program. It need not be declared in other functions. A global variable is also known as an *external* variable.

The variables **i, balance** and **sum** are called *local* variables because they are declared inside a function. Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions. Note that the variable **i** has been declared in both the functions. Any change in the value of **i** in one function does not affect its value in the other.

C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables. The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed. For now, remember that there are four storage class specifiers (**auto, register, static**, and **extern**) whose meanings are given in Table 2.10.

The storage class is another qualifier (like **long** or **unsigned**) that can be added to a variable declaration as shown below:

```
auto int count;
register char ch;
static int x;
extern long total;
```

Static and external (**extern**) variables are automatically initialized to zero. Automatic (**auto**) variables contain undefined values (known as 'garbage') unless they are initialized explicitly.

Table 2.10 *Storage Classes and Their Meaning*

Storage class	*Meaning*
auto	Local variable known only to the function in which it is declared. *Default is auto.*
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

2.10 ASSIGNING VALUES TO VARIABLES

Variables are created for use in program statements such as,

The process of giving initial values to variables is called *initialization*. C permits the *initialization* of more than one variables in one statement using multiple assignment operators. For example the statements

```
p = q = s = 0;
x = y = z = MAX;
```

are valid. The first statement initializes the variables **p, q,** and **s** to zero while the second initializes **x, y,** and **z** with **MAX.** Note that **MAX** is a symbolic constant defined at the beginning.

Remember that external and static variables are initialized to zero by *default*. Automatic variables that are not initialized explicitly will contain garbage.

Example 2.2 Program in Fig. 2.8 shows typical declarations, assignments and values stored in various types of variables.

The variables **x** and **p** have been declared as floating-point variables. Note that the way the value of 1.234567890000 that we assigned to **x** is displayed under different output formats. The value of **x** is displayed as 1.234567880630 under %.12lf format, while the actual value assigned is 1.234567890000. This is because the variable **x** has been declared as a **float** that can store values only up to six decimal places.

The variable **m** that has been declared as **int** is not able to store the value 54321 correctly. Instead, it contains some garbage. Since this program was run on a 16-bit machine, the maximum value that an **int** variable can store is only 32767. However, the variable **k** (declared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variable **n** has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but the value is printed as 9.876543 under %lf format. Note that unless specified otherwise, the **printf** function will always display a **float** or **double** value to six decimal places. We will discuss later the output formats for displaying numbers.

Program

```
main()
{
/*.....DECLARATIONS.....*/
    float    x,p ;
    double   y,q ;
    unsigned k ;
/*.....DECLARATIONS AND ASSIGNMENTS.....*/
    int      m = 54321 ;
    long int n = 1234567890 ;
/*.....ASSIGNMENTS.....*/
    x = 1.234567890000 ;
    y = 9.87654321 ;
    k = 54321 ;
    p = q = 1.0 ;
/*.....PRINTING.....*/
```

```
        printf("m = %d\n", m) ;
        printf("n = %ld\n", n) ;
        printf("x = %.12lf\n", x) ;
        printf("x = %f\n", x) ;
        printf("y = %.12lf\n",y) ;
        printf("y = %lf\n", y) ;
        printf("k = %u p = %f q = %.12lf\n", k, p, q) ;
    }
```

Output

```
m = -11215
n = 1234567890
x = 1.234567880630
x = 1.234568
y = 9.876543210000
y = 9.876543
k = 54321 p = 1.000000 q = 1.000000000000
```

Fig. 2.8 *Examples of assignments*

Reading Data from Keyboard

Another way of giving values to variables is to input data through keyboard using the **scanf** function. It is a general input function available in C and is very similar in concept to the **printf** function. It works much like an INPUT statement in BASIC. The general format of **scanf** is as follows:

scanf("control string", &variable1,&variable2,....);

The control string contains the format of data being received. The ampersand symbol **&** before each variable name is an operator that specifies the variable name's *address*. We must always use this operator, otherwise unexpected results may occur. Let us look at an example:

```
scanf("%d", &number);
```

When this statement is encountered by the computer, the execution stops and waits for the value of the variable **number** to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and the 'Return' Key is pressed, the computer then proceeds to the next statement. Thus, the use of **scanf** provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable **number**.

Example 2.3 The program in Fig. 2.9 illustrates the use of **scanf** function.

The first executable statement in the program is a **printf,** requesting the user to enter an integer number. This is known as "prompt message" and appears on the screen like

```
Enter an integer number
```

As soon as the user types in an integer number, the computer proceeds to compare the

value with 100. If the value typed in is less than 100, then a message

Your number is smaller than 100

is printed on the screen. Otherwise, the message

Your number contains more than two digits

is printed. Outputs of the program run for two different inputs are also shown in Fig. 2.9.

```
Program
        main()
        {
             int number;

             printf("Enter an integer number\n");
             scanf ("%d", &number);

             if ( number < 100 )
               printf("Your number is smaller than 100\n\n");
             else
               printf("Your number contains more than two digits\n");
        }

Output
        Enter an integer number
        54
        Your number is smaller than 100
        Enter an integer number
        108
        Your number contains more than two digits
```

Fig. 2.9 *Use of* ***scanf*** *function for interactive computing*

Some compilers permit the use of the 'prompt message' as a part of the control string in **scanf**, like

```
scanf("Enter a number %d",&number);
```

We discuss more about **scanf** in Chapter 4.

In Fig. 2.9 we have used a decision statement **if...else** to decide whether the number is less than 100. Decision statements are discussed in depth in Chapter 5.

Example 2.4 Sample program 3 discussed in Chapter 1 can be converted into a more flexible interactive program using **scanf** as shown in Fig. 2.10.

In this case, computer requests the user to input the values of the amount to be invested, interest rate and period of investment by printing a prompt message

Input amount, interest rate, and period

and then waits for input values. As soon as we finish entering the three values corresponding to the

```
Program
        main()
        {
              int year, period ;
              float amount, inrate, value ;

              printf("Input amount, interest rate, and period\n\n") ;
              scanf ("%f %f %d", &amount, &inrate, &period) ;
              printf("\n") ;
              year = 1 ;

              while( year <= period )
              {
                    value = amount + inrate * amount ;
                    printf("%2d Rs %8.2f\n", year, value) ;
                    amount = value ;
                    year = year + 1 ;
              }
        }
Output
     Input amount, interest rate, and period

       10000 0.14 5

         1 Rs 11400.00
         2 Rs 12996.00
         3 Rs 14815.44
         4 Rs 16889.60
         5 Rs 19254.15
     Input amount, interest rate, and period

       20000 0.12 7

         1 Rs 22400.00
         2 Rs 25088.00
         3 Rs 28098.56
         4 Rs 31470.39
         5 Rs 35246.84
         6 Rs 39476.46
         7 Rs 44213.63
```

Fig. 2.10 *Interactive investment program*

three variables **amount, inrate,** and **period,** the computer begins to calculate the amount at the end of each year, up to 'period' and produces output as shown in Fig. 2.10.

Note that the **scanf** function contains three variables. In such cases, care should be exercised to see that the values entered match the *order* and *type* of the variables in the list. Any mismatch might lead to unexpected results. The compiler may not detect such errors.

2.11 DEFINING SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant **"pi".** Another example is the total number of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs. These are

1. problem in modification of the program and
2. problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of such constants to a *symbolic name* frees us from these problems. For example, we may use the name **STRENGTH** to define the number of students and **PASS_MARK** to define the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

#define symbolic-name value of constant

Valid examples of constant definitions are:

```
#define STRENGTH 100
#define PASS_MARK 50
#define MAX 200
#define PI 3.14159
```

Symbolic names are sometimes called *constant identifiers.* Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define** statement which define a symbolic constant:

1. Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.)
2. No blank space between the pound sign '#' and the word **define** is permitted.
3. '#' must be the first character in the line.
4. A blank space is required between **#define** and *symbolic name* and between the ***symbolic name*** and the *constant*.
5. **#define** statements must not end with a semicolon.
6. After definition, the *symbolic name* should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
7. Symbolic names are NOT declared for data types. Its data type depends on the type of constant.
8. **#define** statements may appear *anywhere* in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

#define statement is a *preprocessor* compiler directive and is much more powerful than what has been mentioned here. More advanced types of definitions will be discussed later. Table 2.11 illustrates some invalid statements of **#define.**

Table 2.11 *Examples of Invalid #define Statements*

Statement	*Validity*	*Remark*
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

2.12 DECLARING A VARIABLE AS CONSTANT

We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization. Example:

```
const int class_size = 40;
```

const is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the **int** variable **class_size** must not be modified by the program. However, it can be used on the right_hand side of an assignment statement like any other variable.

2.13 DECLARING A VARIABLE AS VOLATILE

ANSI standard defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program). For example:

```
volatile int date;
```

The value of **date** may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When we declare a variable as **volatile**, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

Remember that the value of a variable declared as **volatile** can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both **const** and **volatile** as shown below:

```
volatile const int location = 100;
```

NOTE: C99 adds another qualifier called **restrict**. See the Appendix "C99 Features".

2.14 OVERFLOW AND UNDERFLOW OF DATA

Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine. Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.

Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant. C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/output values.

Just Remember

- Do not use the underscore as the first character of identifiers (or variable names) because many of the identifiers in the system library start with underscore.
- Use only 31 or less characters for identifiers. This helps ensure portability of programs.
- Do not use keywords or any system library names for identifiers.
- Use meaningful and intelligent variable names.
- Do not create variable names that differ only by one or two letters.
- Each variable used must be declared for its type at the beginning of the program or function.
- All variables must be initialized before they are used in the program.
- Integer constants, by default, assume **int** types. To make the numbers **long** or **unsigned**, we must append the letters L and U to them.
- Floating point constants default to **double**. To make them to denote **float** or **long double**, we must append the letters F or L to the numbers.
- Do not use lowercase l for long as it is usually confused with the number 1.

- Use single quote for character constants and double quotes for string constants.
- A character is stored as an integer. It is therefore possible to perform arithmetic operations on characters.
- Do not combine declarations with executable statements.
- A variable can be made constant either by using the preprocessor command **#define** at the beginning of the program or by declaring it with the qualifier **const** at the time of initialization.
- Do not use semicolon at the end of **#define** directive.
- The character # should be in the first column.
- Do not give any space between # and **define.**
- C does not provide any warning or indication of overflow. It simply gives incorrect results. Care should be exercised in defining correct data type.
- A variable defined before the main function is available to all the functions in the program.
- A variable defined inside a function is local to that function and not available to other functions.

Case Studies

1. Calculation of Average of Numbers

A program to calculate the average of a set of N numbers is given in Fig. 2.11.

Program

```
#define    N    10                    /* SYMBOLIC CONSTANT */
main()
{
   int     count ;                     /* DECLARATION OF */
   float   sum, average, number ;      /* VARIABLES */
   sum     = 0 ;                       /* INITIALIZATION */
   count   = 0 ;                       /* OF VARIABLES */
   while( count < N )
   {
          scanf("%f", &number) ;
          sum = sum + number ;
          count = count + 1 ;
    }
    average = sum/N ;
    printf("N = %d Sum = %f", N, sum);
    printf(" Average = %f", average);
}
```

Output

```
1
2.3
```

```
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10    Sum = 38.799999 Average = 3.880
```

Fig. 2.11 *Average of N numbers*

The variable **number** is declared as **float** and therefore it can take both integer and real numbers. Since the symbolic constant **N** is assigned the value of 10 using the **#define** statement, the program accepts ten values and calculates their sum using the **while** loop. The variable **count** counts the number of values and as soon as it becomes 11, the **while** loop is exited and then the average is calculated.

Notice that the actual value of sum is 38.8 but the value displayed is 38.799999. In fact, the actual value that is displayed is quite dependent on the computer system. Such an inaccuracy is due to the way the floating point numbers are internally represented inside the computer.

2. Temperature Conversion Problem

The program presented in Fig. 2.12 converts the given temperature in fahrenheit to celsius using the following conversion formula:

$$C = \frac{F - 32}{1.8}$$

```
Program
   #define F_LOW     0          /* ---------------- */
   #define F_MAX   250        /* SYMBOLIC CONSTANTS    */
   #define STEP     25        /* --------------        */

   main()
   {
        typedef float REAL ;          /* TYPE DEFINITION */
   REAL fahrenheit, celsius ;  /* DECLARATION */

   fahrenheit = F_LOW ;          /* INITIALIZATION */
   printf("Fahrenheit Celsius\n\n") ;
   while( fahrenheit <= F_MAX )
   {
        celsius = ( fahrenheit - 32.0 ) / 1.8 ;
        printf(" %5.1f %7.2f\n", fahrenheit, celsius);
```

```
            fahrenheit = fahrenheit + STEP ;
        }
    }
```

Output

```
  Fahrenheit          Celsius

         0.0           -17.78
        25.0            -3.89
        50.0            10.00
        75.0            23.89
       100.0            37.78
       125.0            51.67
       150.0            65.56
       175.0            79.44
       200.0            93.33
       225.0           107.22
       250.0           121.11
```

Fig. 2.12 *Temperature conversion—fahrenheit-celsius*

The program prints a conversion table for reading temperature in celsius, given the fahrenheit values. The minimum and maximum values and step size are defined as symbolic constants. These values can be changed by redefining the **#define** statements. An user-defined data type name **REAL** is used to declare the variables **fahrenheit** and **celsius**.

The formation specifications %5.1f and %7.2 in the second **printf** statement produces two-column output as shown.

Review Questions

2.1 State whether the following statements are *true* or *false*.

(a) Any valid printable ASCII character can be used in an identifier.
(b) All variables must be given a type when they are declared.
(c) Declarations can appear anywhere in a program.
(d) ANSI C treats the variables **name** and **Name** to be same.
(e) The underscore can be used anywhere in an identifier.
(f) The keyword **void** is a data type in C.
(g) Floating point constants, by default, denote **float** type values.
(h) Like variables, constants have a type.
(i) Character constants are coded using double quotes.
(j) Initialization is the process of assigning a value to a variable at the time of declaration.
(k) All **static** variables are automatically initialized to zero.
(l) The **scanf** function can be used to read only one value at a time.

2.2 Fill in the blanks with appropriate words.

(a) The keyword _____ can be used to create a data type identifier.
(b) _____ is the largest value that an unsigned short int type variable can store.
(c) A global variable is also known as _____ variable.
(d) A variable can be made constant by declaring it with the qualifier _____ at the time of initialization.

2.3 What are trigraph characters? How are they useful?

2.4 Describe the four basic data types. How could we extend the range of values they represent?

2.5 What is an unsigned integer constant? What is the significance of declaring a constant unsigned?

2.6 Describe the characteristics and purpose of escape sequence characters.

2.7 What is a variable and what is meant by the "value" of a variable?

2.8 How do variables and symbolic names differ?

2.9 State the differences between the declaration of a variable and the definition of a symbolic name.

2.10 What is initialization? Why is it important?

2.11 What are the qualifiers that an **int** can have at a time?

2.12 A programmer would like to use the word DPR to declare all the double-precision floating point values in his program. How could he achieve this?

2.13 What are enumeration variables? How are they declared? What is the advantage of using them in a program?

2.14 Describe the purpose of the qualifiers **const** and **volatile**.

2.15 When dealing with very small or very large numbers, what steps would you take to improve the accuracy of the calculations?

2.16 Which of the following are invalid constants and why?

0.0001	5 × 1.5	99999
+100	75.45 E-2	"15.75"
–45.6	–1.79 e + 4	0.00001234

2.17 Which of the following are invalid variable names and why?

Minimum	First.name	n1+n2	&name
doubles	3rd_row	n$	Row1
float	Sum Total	Row Total	Column-total

2.18 Find errors, if any, in the following declaration statements.

```
Int x;
float letter,DIGIT;
double = p,q
exponent alpha,beta;
m,n,z: INTEGER
short char c;
long int m; count;
long float temp;
```

2.19 What would be the value of x after execution of the following statements?

```
int x, y = 10;
char z = 'a';
x = y + z;
```

2.20 Identify syntax errors in the following program. After corrections, what output would you expect when you execute it?

```
#define PI 3.14159
main()
{
     int R,C;            /* R-Radius of circle
     float perimeter;    /* Circumference of circle */
     float area;         /* Area of circle */
     C = PI
     R = 5;
     Perimeter = 2.0 * C *R;
     Area    = C*R*R;
     printf("%f",  "%d",&perimeter,&area)
     }
```

Programming Exercises

2.1 Write a program to determine and print the sum of the following harmonic series for a given value of n:

$$1+ 1/2 +1/3 +....+ 1/n$$

The value of n should be given interactively through the terminal.

2.2 Write a program to read the price of an item in decimal form (like 15.95) and print the output in paise (like 1595 paise).

2.3 Write a program that prints the even numbers from 1 to 100.

2.4 Write a program that requests two float type numbers from the user and then divides the first number by the second and display the result along with the numbers.

2.5 The price of one kg of rice is Rs. 16.75 and one kg of sugar is Rs. 15. Write a program to get these values from the user and display the prices as follows:

*** LIST OF ITEMS ***

Item	Price
Rice	Rs 16.75
Sugar	Rs 15.00

2.6 Write program to count and print the number of negative and positive numbers in a given set of numbers. Test your program with a suitable set of numbers. Use **scanf** to read the numbers. Reading should be terminated when the value 0 is encountered.

2.7 Write a program to do the following:

(a) Declare x and y as integer variables and z as a short integer variable.
(b) Assign two 6 digit numbers to x and y
(c) Assign the sum of x and y to z
(d) Output the values of x, y and z

Comment on the output.

2.8 Write a program to read two floating point numbers using a **scanf** statement, assign their sum to an integer variable and then output the values of all the three variables.

2.9 Write a program to illustrate the use of **typedef** declaration in a program.

2.10 Write a program to illustrate the use of symbolic constants in a real-life application.

3

Operators and Expressions

3.1 INTRODUCTION

C supports a rich set of built-in operators. We have already used several of them, such as =, +, –, *, & and <. An *operator* is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical *expressions*.

C operators can be classified into a number of categories. They include:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

An expression is a sequence of operands and operators that reduces to a single value. For example,

```
10 + 15
```

is an expression whose value is 25. The value can be any type other than ***void***.

3.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. They are listed in Table 3.1. The operators +, –, *, and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Table 3.1 *Arithmetic Operators*

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

```
a – b    a + b
a * b    a / b
a % b   –a * b
```

Here **a** and **b** are variables and are known as *operands*. The modulo division operator % cannot be used on floating point data. Note that C does not have an operator for *exponentiation*. Older versions of C does not support unary plus but ANSI C supports it.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if **a** and **b** are integers, then for **a** = 14 and **b** = 4 we have the following results:

```
a – b   =   10
a + b   =   18
a * b   =   56
a / b   =   3 (decimal part truncated)
a % b   =   2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of trunction is implementation dependent. That is,

$$6/7 = 0 \text{ and } -6/-7 = 0$$

but –6/7 may be zero or –1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

```
–14 % 3    =   –2
–14 % –3   =   –2
14 % –3    =   2
```

Example 3.1 The program in Fig. 3.1 shows the use of integer arithmetic to convert a given number of days into months and days.

```
Program
   main ()
   {
      int months, days ;

      printf("Enter days\n") ;
      scanf("%d", &days) ;

      months = days / 30 ;
      days = days % 30 ;
      printf("Months = %d Days = %d", months, days) ;
   }

Output
   Enter days
   265
   Months = 8 Days = 25
   Enter days
   364
   Months = 12 Days = 4
   Enter days
   45
   Months = 1 Days = 15
```

Fig. 3.1 *Illustration of integer arithmetic*

The variables months and days are declared as integers. Therefore, the statement

```
months = days/30;
```

truncates the decimal part and assigns the integer part to months. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If **x, y,** and **z** are **floats**, then we will have:

$$x = 6.0/7.0 = 0.857143$$
$$y = 1.0/3.0 = 0.333333$$
$$z = -2.0/3.0 = -0.666667$$

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

15/10.0 = 1.5

whereas

15/10 = 1

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

3.3 RELATIONAL OPERATORS

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<', meaning 'less than'. An expression such as

a < b or 1 < 20

containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false*. For example

10 < 20 is true

but

20 < 10 is false

C supports six relational operators in all. These operators and their meanings are shown in Table 3.2.

Table 3.2 *Relational Operators*

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

ae-1 relational operator ae-2

ae-1 and *ae-2* are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5 <= 10 TRUE

4.5 < –10 FALSE

–35 >= 0 FALSE

10 < 7+5 TRUE

a+b = c+d TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in *decision statements* such as **if** and **while** to decide the course of action of a running program. We have already used the **while** statement in Chapter 1. Decision statements are discussed in detail in Chapters 5 and 6.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	*Simplified one*
!(x<y)	x >= y
!(x>y)	x <= y
!(x!=y)	x == y
!(x<=y)	x > y
!(x>=y)	x < y
!(x == y)	x != y

3.4 LOGICAL OPERATORS

In addition to the relational operators, C has the following three *logical operators.*

```
&&   meaning logical   AND
||   meaning logical   OR
!    meaning logical   NOT
```

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

```
a > b && x == 10
```

An expression of this kind, which combines two or more relational expressions, is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of *one* or *zero*, according to the truth table shown in Table 3.3. The logical expression given above is true only if **a > b** is *true* and **x == 10** is *true*. If either (or both) of them are false, the expression is *false*.

Table 3.3 *Truth Table*

op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number < 0 || number > 100)

We shall see more of them when we discuss decision statements.

NOTE: Relative precedence of the relational and logical operators is as follows:

```
Highest      !

             > >=   < <=

             == !=

             &&

Lowest       ||
```

It is important to remember this when we use these operators in compound expressions.

3.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='. In addition, C has a set of '*shorthand*' assignment operators of the form

v op= exp;

Where *v* is a variable, *exp* is an expression and *op* is a C binary arithmetic operator. The operator **op=** is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op (exp);
```

with **v** evaluated only once. Consider an example

```
x += y+1;
```

This is same as the statement

```
x = x + (y+1);
```

The shorthand operator **+=** means 'add y+1 to x' or 'increment x by y+1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 3.4.

Table 3.4 *Shorthand Assignment Operators*

Statement with simple assignment operator	*Statement with shorthand operator*
a = a + 1	a += 1
a = a – 1	a –= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

```
value(5*j–2) = value(5*j–2) + delta;
```

With the help of the += operator, this can be written as follows:

```
value(5*j–2) += delta;
```

It is easier to read and understand and is more efficient because the expression 5*j–2 is evaluated only once.

Example 3.2 Program of Fig. 3.2 prints a sequence of squares of numbers. Note the use of the shorthand operator *= .

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

```
a = a*a;
```

replaces the current value of **a** by its square. When the value of **a** becomes equal or greater than **N** (=100) the **while** is terminated. Note that the output contains only three values 2, 4 and 16.

Program

```
#define   N    100
#define   A    2
main()
{
          int a;
          a = A;
          while( a < N )
          {
               printf("%d\n", a);
               a *= a;
          }
}
```

Output

```
2
4
16
```

Fig. 3.2 *Use of shorthand operator *=*

3.6 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

```
++ and --
```

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following form:

```
++m; or m++;
--m; or m--;
```

```
++m; is equivalent to m = m+1; (or m += 1;)
--m; is equivalent to m = m-1; (or m -= 1;)
```

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. *A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.*

Similar is the case, when we use ++ (or – –) in subscripted variables. That is, the statement

```
a[i++] = 10;
```

is equivalent to

```
a[i] = 10;
i = i+1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++ -j+10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

Rules for ++ and – – Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or – –) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++ (or – –) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associatively of ++ and – – operators are the same as those of unary + and unary –.

3.7 CONDITIONAL OPERATOR

A ternary operator pair "? :" is available in C to construct conditional expressions of the form

exp1 ? exp2 : exp3

where *exp1, exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements.

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
   x = a;
else
   x = b;
```

3.8 BITWISE OPERATORS

C has a distinction of supporting special operators known as *bitwise operators* for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to **float** or **double**. Table 3.5 lists the bitwise operators and their meanings. They are discussed in detail in Appendix I.

Table 3.5 *Bitwise Operators*

Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

3.9 SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, **sizeof** operator, pointer operators (**&** and ***) and member selection operators (. and –>). The comma and **sizeof** operators are discussed in this section while the pointer operators are discussed in

Chapter 11. Member selection operators which are used to select members of a structure are discussed in Chapters 10 and 11. ANSI committee has introduced two preprocessor operators known as "string-izing" and "token-pasting" operators (# and ##). They will be discussed in Chapter 14.

The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to **x**, then assigns 5 to **y**, and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

In **for** loops:

```
for ( n = 1, m = 10, n <=m; n++, m++)
```

In **while** loops:

```
while (c = getchar( ), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

The sizeof Operator

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

```
m = sizeof (sum);
n = sizeof (long int);
k = sizeof (235L);
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Example 3.3 In Fig. 3.3, the program employs different kinds of operators. The results of their evaluation are also shown for comparison.

Notice the way the increment operator ++ works when used in an expression. In the statement

```
c = ++a - b;
```

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 before it is used in the expression. However, in the statement

```
d = b++ + a;
```

the old value of **b** (=10) is used in the expression. Here, b is incremented by 1 after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement

```
printf("a%%b = %d\n", a%b);
```

The program also illustrates that the expression

```
c > d ? 1 : 0
```

assumes the value 0 when c is less than d and 1 when c is greater than d.

Program

```
main()
{
    int a, b, c, d;

    a = 15;
    b = 10;
    c = ++a - b;

    printf("a = %d b = %d c = %d\n",a, b, c);

    d = b++ +a;

    printf("a = %d b = %d d = %d\n",a, b, d);
    printf("a/b = %d\n", a/b);
    printf("a%%b = %d\n", a%b);
    printf("a *= b = %d\n", a*=b);
    printf("%d\n", (c>d) ? 1 : 0);
    printf("%d\n", (c<d) ? 1 : 0);
}
```

Output

```
a = 16 b = 10 c = 6
a = 16 b = 11 d = 26
a/b = 1
a%b = 5
a *= b = 176
0
1
```

Fig. 3.3 *Further illustration of arithmetic operators*

3.10 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6. Remember that C does not have an operator for exponentiation.

Table 3.6 *Expressions*

Algebraic expression	*C expression*
a x b - c	a * b - c
(m+n) (x+y)	(m+n) * (x+y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2+2x+1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right)+c$	x/y+c

3.11 EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form:

variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a * b - c;
y = b / c * a;
z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

Example 3.4 The program in Fig. 3.4 illustrates the use of variables in expressions and their evaluation.

Output of the program also illustrates the effect of presence of parentheses in expressions. This is discussed in the next section.

Program

```
main()
{
        float a, b, c, x, y, z;
```

```
    a = 9;
    b = 12;
    c = 3;

    x = a - b / 3 + c * 2 - 1;
    y = a - b / (3 + c) * (2 - 1);
    z = a - (b / (3 + c) * 2) - 1;

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

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Fig. 3.4 *Illustrations of evaluation of expressions*

3.12 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from *left to right* using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High priority * / %

Low priority + –

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program of Fig. 3.4.

x = a–b/3 + c*2–1

When a = 9, b = 12, and c = 3, the statement becomes

x = 9–12/3 + 3*2–1

and is evaluated as follows

First pass

Step1: x = 9–4+3*2–1

Step2: x = 9–4+6–1

Second pass

Step3: x = 5+6–1
Step4: x = 11–1
Step5: x = 10

These steps are illustrated in Fig. 3.5. The numbers inside parentheses refer to step numbers.

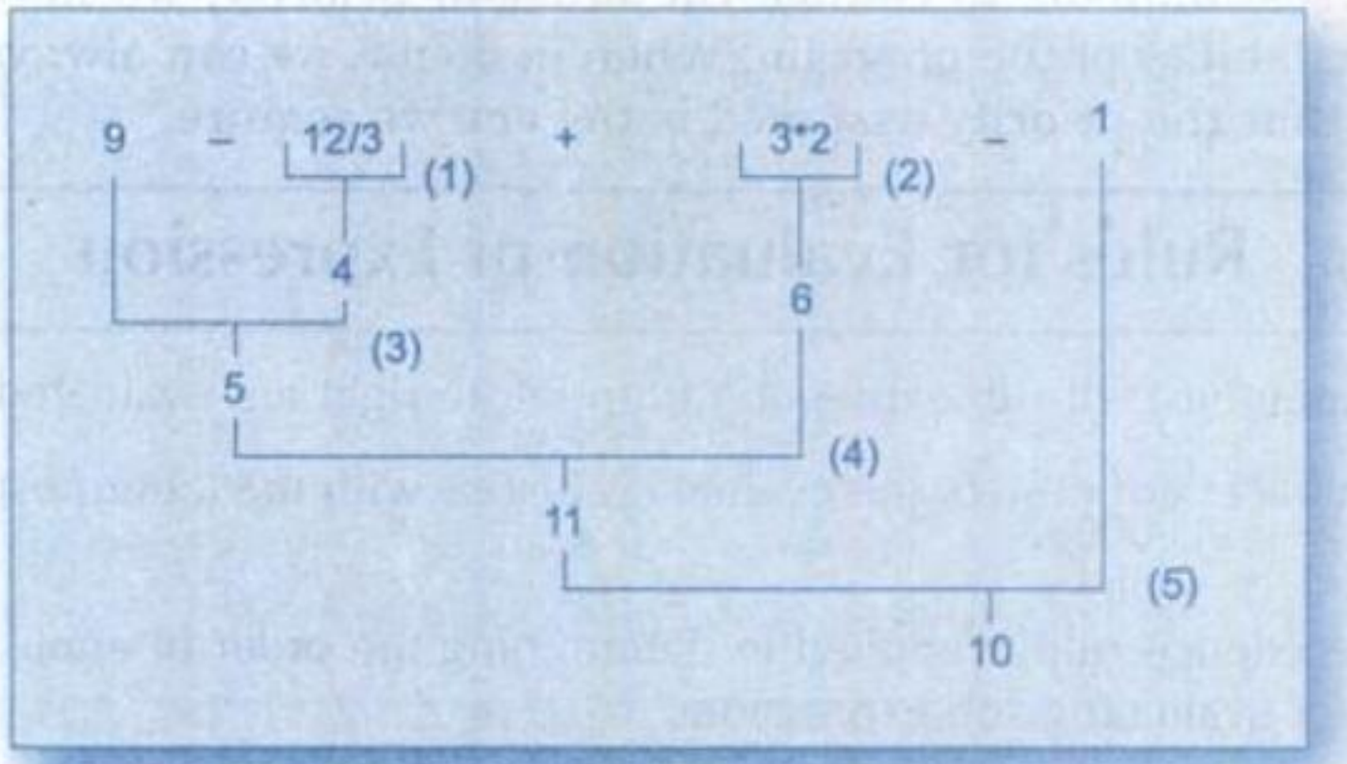


Fig. 3.5 *Illustration of hierarchy of operations*

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

9–12/(3+3)*(2–1)

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step1: 9-12/6 * (2-1)
Step2: 9-12/6 * 1

Second pass

Step3: 9-2 * 1
Step4: 9-2

Third pass

Step5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

$$9 - ((12/3) + 3 * 2) - 1 = -2$$

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

3.13 SOME COMPUTATIONAL PROBLEMS

When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors. We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems. For example, consider the following statements:

```
a = 1.0/3.0;
b = a * 3.0;
```

We know that (1.0/3.0) 3.0 is equal to 1. But there is no guarantee that the value of **b** computed in a program will equal 1.

Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.

The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

Example 3.5 Output of the program in Fig. 3.6 shows round-off errors that can occur in computation of floating point numbers.

Program

```
/*———————— Sum of n terms of 1/n ————————*/
   main()
   {
        float sum, n, term ;
        int count = 1 ;

        sum = 0 ;
        printf("Enter value of n\n") ;
            scanf("%f", &n) ;
        term = 1.0/n ;
        while( count <= n )
        {
               sum = sum + term ;
               count++ ;
        }
        printf("Sum = %f\n", sum) ;
   }
```

Output

```
Enter value of n
99
Sum = 1.000001
Enter value of n
143
Sum = 0.999999
```

Fig. 3.6 *Round-off errors in floating point computations*

We know that the sum of n terms of 1/n is 1. However, due to errors in floating point representation, the result is not always 1.

3.14 TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic conversion is known as *implicit type conversion.*

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig. 3.7.

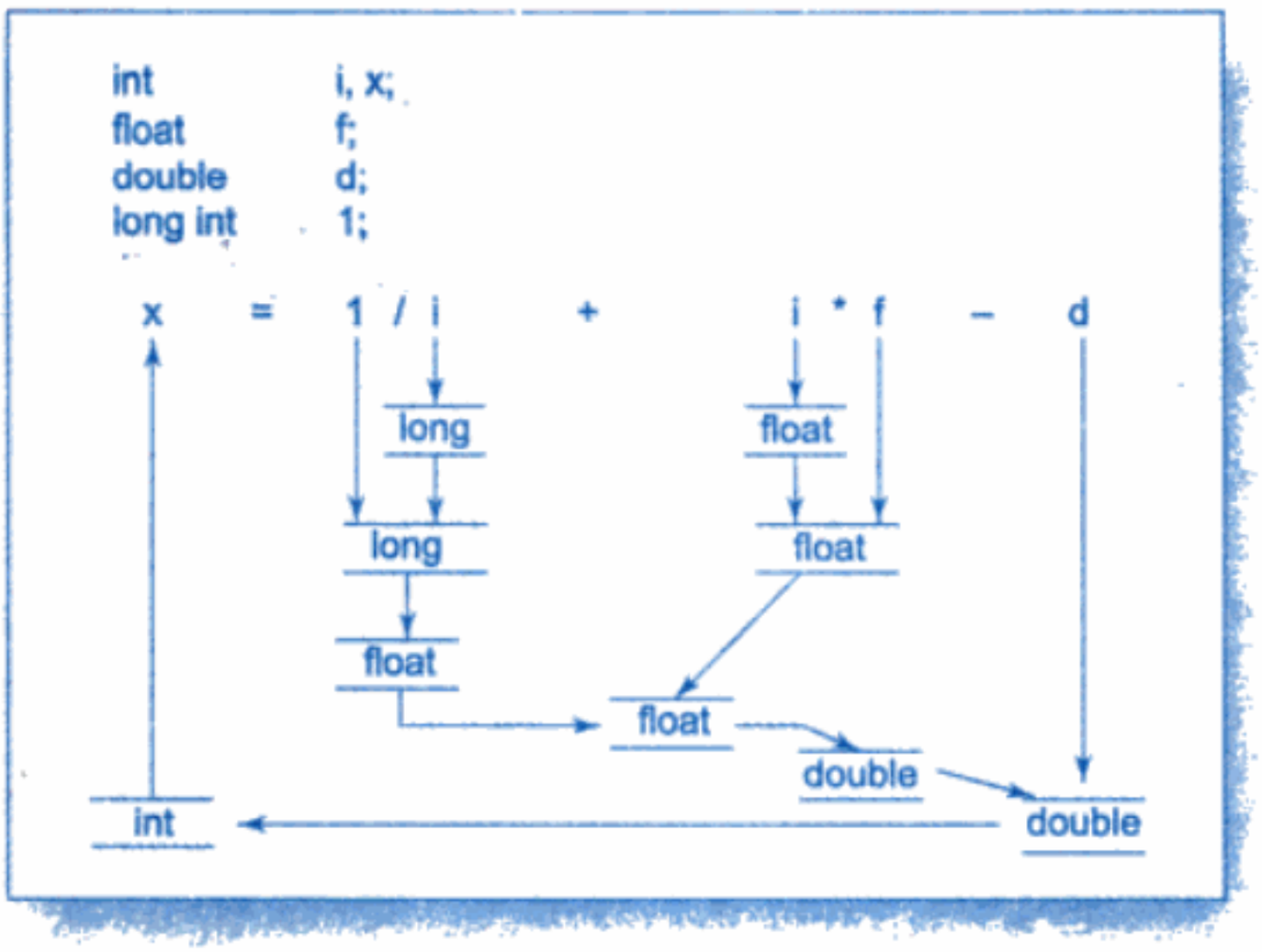


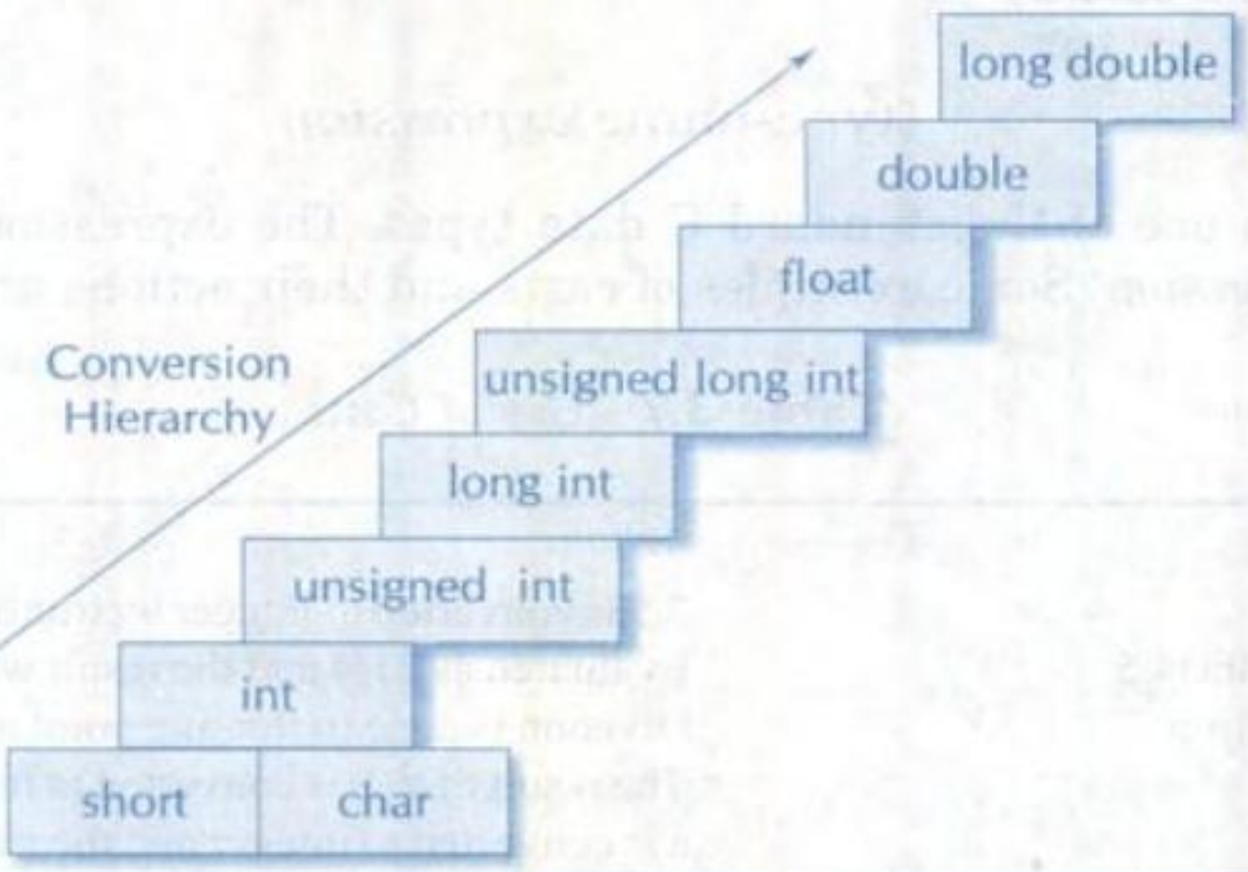
Fig. 3.7 *Process of implicit type conversion*

Given below is the sequence of rules that are applied while evaluating expressions.
All **short** and **char** are automatically converted to **int**; then

1. if one of the operands is **long double,** the other will be converted to **long double** and the result will be **long double**;
2. else, if one of the operands is **double,** the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands is **float,** the other will be converted to **float** and the result will be **float**;
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;
5. else, if one of the operands is **long int** and the other is **unsigned int,** then
   (a) if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
   (b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;
6. else, if one of the operands is **long int,** the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int,** the other will be converted to **unsigned int** and the result will be **unsigned int.**

Conversion Hierarchy

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:



Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

Explicit Conversion

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
ratio = female_number/male_number
```

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

```
ratio = (float) female_number/male_number
```

The operator (**float**) converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (**float**) affect the value of the variable **female number.** And also, the type of **female number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *explicit conversion* or *casting a value.* The general form of a cast is:

(type-name)*expression*

where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 3.7.

Table 3.7 *Use of Casts*

Example	*Action*
x = (**int**) 7.5	7.5 is converted to integer by truncation.
a = (**int**) 21.3/(int)4.5	Evaluated as 21/4 and the result would be 5.
b = (**double**)sum/n	Division is done in floating point mode.
y = (**int**) (a+b)	The result of a+b is converted to integer.
z = (**int**)a+b	a is converted to integer and then added to b.
p = cos((**double**)x)	Converts x to double before using it.

Casting can be used to round-off a given value. Consider the following statement:

x = (**int**) (y+0.5);

If **y** is 27.6, **y**+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression, being cast is not changed.

Example 3.6 Figure 3.8 shows a program using a cast to evaluate the equation

$$\text{sum} = \sum_{i=1}^{n} (1/i)$$

Program

```
main()
{
     float    sum ;
     int              n ;

     sum = 0 ;

     for( n = 1 ; n <= 10 ; ++n )
     {
          sum = sum + 1/(float)n ;
          printf("%2d %6.4f\n", n, sum) ;
     }
}
```

```
Output

 1  1.0000
 2  1.5000
 3  1.8333
 4  2.0833
 5  2.2833
 6  2.4500
 7  2.5929
 8  2.7179
 9  2.8290
10  2.9290
```

Fig. 3.8 *Use of a cast*

3.15 OPERATOR PRECEDENCE AND ASSOCIATIVITY

As mentioned earlier each operator, in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct *levels of precedence* and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level. This is known as the *associativity* property of an operator. Table 3.8 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. The list also includes those operators, which we have not yet been discussed.

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

if (x == 10 + 15 && y < 10)

The precedence rules say that the ***addition*** operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to :

if (x == 25 && y < 10)

The next step is to determine whether **x** is equal to 25 and **y** is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is FALSE (0)
y < 10 is TRUE (1)

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally we get:

if (FALSE && TRUE)

Because one of the conditions is FALSE, the complex condition is FALSE.

In the case of **&&**, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of **| |**, the second operand will not be evaluated if the first is non-zero.

Table 3.8 *Summary of C Operators*

Operator	*Description*	*Associativity*	*Rank*
( )	Function call	Left to right	1
[ ]	Aray element reference		
+	Unary plus		
–	Unary minus	Right to left	2
++	Increment		
– –	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address		
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
–	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
=	Assignment operators	Right to left	14
*= /= %=			
+= -= &=			
^= |=			
<<= >>=			
,	Comma operator	Left to right	15

Rules of Precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

3.16 MATHEMATICAL FUNCTIONS

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. However, there are systems that have a more comprehensive math library and one should consult the reference manual to find out which functions are available. Table 3.9 lists some standard math functions.

Table 3.9 *Math functions*

Function	*Meaning*
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power (e^x)
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x > 0
log10(x)	Base 10 log of x, x > 0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x >= 0

Note: 1. **x** and **y** should be declared as **double.**
2. In trigonometric and hyperbolic functions, **x** and **y** are in radians.
3. All the functions return a **double**.

4. C99 has added **float** and **long double** versions of these fuctions.
5. C99 has added many more mathematical functions.
6. See the Appendix "C99 Features" for details.

As pointed out earlier in Chapter 1, to use any of these functions in a program, we should include the line:

```
# include <math.h>
```

in the beginning of the program.

Just Remember

- Use *decrement* and *increment* operators carefully. Understand the difference between **postfix** and **prefix** operations before using them.
- Add parentheses wherever you feel they would help to make the evaluation order clear.
- Be aware of side effects produced by some expressions.
- Avoid any attempt to divide by zero. It is normally undefined. It will either result in a fatal error or in incorrect results.
- Do not forget a semicolon at the end of an expression.
- Understand clearly the precedence of operators in an expression. Use parentheses, if necessary.
- Associativity is applied when more than one operator of the same precedence are used in an expression. Understand which operators associate from right to left and which associate from left to right.
- Do not use *increment* or *decrement* operators with any expression other than a *variable identifier*.
- It is illegal to apply modules operator % with anything other than integers.
- Do not use a variable in an expression before it has been assigned a value.
- Integer division always truncates the decimal part of the result. Use it carefully. Use casting where necessary.
- The result of an expression is converted to the type of the variable on the left of the assignment before assigning the value to it. Be careful about the loss of information during the conversion.
- All mathematical functions implement *double* type parameters and return *double* type values.
- It is an error if any space appears between the two symbols of the operators ==, !=, <= and >=.
- It is an error if the two symbols of the operators !=, <= and >= are reversed.
- Use spaces on either side of binary operator to improve the readability of the code.
- Do not use increment and decrement operators to floating point variables.
- Do not confuse the equality operator == with the assignment operator =.

Case Studies

1. Salesman's Salary

A computer manufacturing company has the following monthly compensation policy to their sales-persons:

Minimum base salary : 1500.00
Bonus for every computer sold : 200.00
Commission on the total monthly sales : 2 per cent

Since the prices of computers are changing, the sales price of each computer is fixed at the beginning of every month. A program to compute a sales-person's gross salary is given in Fig. 3.9.

Program

```
#define BASE_SALAR          1500.00
#define BONUS_RATE          200.00
#define COMMISSION          0.02
main()
{
    int quantity ;
    float gross_salary, price ;
    float bonus, commission ;
    printf("Input number sold and price\n") ;
    scanf("%d %f", &quantity, &price) ;
    bonus           = BONUS_RATE * quantity ;
    commission      = COMMISSION * quantity * price ;
    gross_salary    = BASE_SALARY + bonus + commission ;
    printf("\n");
    printf("Bonus           = %6.2f\n", bonus) ;
    printf("Commission      = %6.2f\n", commission) ;
    printf("Gross salary    = %6.2f\n", gross_salary) ;
}
```

Output

```
Input number sold and price
5 20450.00
Bonus               = 1000.00
Commission          = 2045.00
Gross salary        = 4545.00
```

Fig. 3.9 *Program of salesman's salary*

Given the base salary, bonus, and commission rate, the inputs necessary to calculate the gross salary are, the price of each computer and the number sold during the month.

The gross salary is given by the equation:

Gross salary = base salary + (quantity * bonus rate)
+ (quantity * Price) * commission rate

2. Solution of the quadratic equation

An equation of the form

$$ax^2 + bx + c = 0$$

is known as the *quadratic equation*. The values of x that satisfy the equation are known as the *roots* of the equation. A quadratic equation has two roots which are given by the following two formulae:

$$\text{root 1} = \frac{-b+\text{sqrt}(b^2-4ac)}{2a}$$

$$\text{root 2} = \frac{-b-\text{sqrt}(b^2-4ac)}{2a}$$

A program to evaluate these roots is given in Fig. 3.10. The program requests the user to input the values of **a**, **b** and **c** and outputs **root 1** and **root 2**.

Program

```
#include <math.h>
main()
{
    float a, b, c, discriminant,
          root1, root2;
    printf("Input values of a, b, and c\n");
    scanf("%f %f %f", &a, &b, &c);
    discriminant = b*b - 4*a*c ;
    if(discriminant < 0)
       printf("\n\nROOTS ARE IMAGINARY\n");
    else
    {
       root1 = (-b + sqrt(discriminant))/(2.0*a);
       root2 = (-b - sqrt(discriminant))/(2.0*a);
       printf("\n\nRoot1 = %5.2f\n\nRoot2 = %5.2f\n",
                  root1,root2 );
    }
}
```

Output

```
Input values of a, b, and c
2 4 -16
Root1 = 2.00
Root2 = -4.00
Input values of a, b, and c
1 2 3
ROOTS ARE IMAGINARY
```

Fig. 3.10 *Solution of a quadratic equation*

The term (b^2–4ac) is called the *discriminant*. If the discriminant is less than zero, its square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbers and the program outputs an appropriate message.

Review Questions

3.1 State whether the following statements are *true* or *false*.
(a) All arithmetic operators have the same level of precedence.
(b) The modulus operator % can be used only with integers.
(c) The operators <=, >= and != all enjoy the same level of priority.
(d) During modulo division, the sign of the result is positive, if both the operands are of the same sign.
(e) In C, if a data item is zero, it is considered false.
(f) The expression *!(x<=y)* is same as the expression *x>y*.
(g) A unary expression consists of only one operand with no operators.
(h) Associativity is used to decide which of several different expressions is evaluated first.
(i) An expression statement is terminated with a period.
(j) During the evaluation of mixed expressions, an implicit cast is generated automatically.
(k) An explicit cast can be used to change the expression.
(l) Parentheses can be used to change the order of evaluation expressions.

3.2 Fill in the blanks with appropriate words.
(a) The expression containing all the integer operands is called_____ expression.
(b) The operator _____cannot be used with real operands.
(c) C supports as many as _____relational operators.
(d) An expression that combines two or more relational expressions is termed as _____expression.
(e) The _____operator returns the number of bytes the operand occupies.
(f) The order of evaluation can be changed by using _____ in an expression.
(g) The use of _____ on a variable can change its type in the memory.
(h) _____is used to determine the order in which different operators in an expression are evaluated.

3.3 Given the statement
int a = 10, b = 20, c;
determine whether each of the following statements are true or false.
(a) The statement a = + 10, is valid.
(b) The expression a + 4/6 * 6/2 evaluates to 11.
(c) The expression b + 3/2 * 2/3 evaluates to 20.
(d) The statement a + = b; gives the values 30 to a and 20 to b.
(e) The statement ++a++; gives the value 12 to a
(f) The statement a = 1/b; assigns the value 0.5 to a

3.4 Declared **a** as *int* and **b** as *float*, state whether the following statements are true or false.

(a) The statement a = 1/3 + 1/3 + 1/3; assigns the value 1 to a.
(b) The statement b = 1.0/3.0 + 1.0/3.0 + 1.0/3.0; assigns a value 1.0 to b.
(c) The statement b = 1.0/3.0 * 3.0 gives a value 1.0 to b.
(d) The statement b = 1.0/3.0 + 2.0/3.0 assigns a value 1.0 to b.
(e) The statement a = 15/10.0 + 3/2; assigns a value 3 to a.

3.5 Which of the following expressions are true?
(a) !(5 + 5 >=10)
(b) 5 + 5 = = 10 | | 1 + 3 = = 5
(c) 5 > 10 | | 10 < 20 && 3 < 5
(d) 10 ! = 15 && !(10<20) | | 15 > 30

3.6 Which of the following arithmetic expressions are valid ? If valid, give the value of the expression; otherwise give reason.

(a) 25/3 % 2	(e) –14 % 3
(b) +9/4 + 5	(f) 15.25 + – 5.0
(c) 7.5 % 3	(g) (5/3) * 3 + 5 % 3
(d) 14 % 3 + 7 % 2	(h) 21 % (int)4.5

3.7 Write C assignment statements to evaluate the following equations:
(a) Area = $\pi r^2 + 2 \pi rh$

(b) Torque = $\dfrac{2m_1m_2}{m_1 + m_2} \cdot g$

(c) Side = $\sqrt{a^2+b^2-2ab\cos(x)}$

(d) Energy = mass $\left[\text{acceleration} \times \text{height} + \dfrac{(\text{velocity})^2}{2}\right]$

3.8 Identify unnecessary parentheses in the following arithmetic expressions.
(a) ((x–(y/5)+z)%8) + 25
(b) ((x–y) * p)+q
(c) (m*n) + (–x/y)
(d) x/(3*y)

3.9 Find errors, if any, in the following assignment statements and rectify them.
(a) `x = y = z = 0.5, 2.0. –5.75;`
(b) `m = ++a * 5;`
(c) `y = sqrt(100);`
(d) `p * = x/y;`
(e) `s = /5;`
(f) `a = b++ –c*2`

3.10 Determine the value of each of the following logical expressions if a = 5, b = 10 and c = –6
(a) a > b && a < c
(b) a < b && a > c
(c) a == c | | b > a
(d) b > 15 && c < 0 | | a > 0
(e) (a/2.0 == 0.0 && b/2.0 != 0.0) | | c < 0.0

3.11 What is the output of the following program?

```
main ( )
{
        char x;
        int y;
        x = 100;
        y = 125;
        printf ("%c\n", x) ;
        printf ("%c\n", y) ;
        printf ("%d\n", x) ;
}
```

3.12 Find the output of the following program?

```
main ( )
{
     int x = 100;
     printf("%d/n", 10 + x++);
     printf("%d/n", 10 + ++x);
}
```

3.13 What is printed by the following program?

```
main
{
        int x = 5, y = 10, z = 10 ;
        x = y == z;
        printf("%d",x ) ;
}
```

3.14 What is the output of the following program?

```
main ( )
{
        int x = 100, y = 200;
        printf ("%d", (x > y)? x : y);
}
```

3.15 What is the output of the following program?

```
main ( )
{
        unsigned x = 1 ;
        signed char y = -1 ;
        if(x > y)
                printf(" x > y");
        else
                printf("x<= y") ;
}
```

Did you expect this output? Explain.

3.16 What is the output of the following program? Explain the output.

```
main ( )
{
        int x = 10 ;
        if(x = 20) printf("TRUE") ;
        else printf("FALSE") ;
}
```

3.17 What is the error in each of the following statements?

(a) if (m == 1 & n ! = 0)
printf("OK");

(b) if (x = < 5)
printf ("Jump");

3.18 What is the error, if any, in the following segment?

```
int x = 10 ;
float y = 4.25 ;
x = y%x ;
```

3.19 What is printed when the following is executed?

```
for (m = 0; m <3; ++m)
printf("%d/n", (m%2) ? m: m+2);
```

3.20 What is the output of the following segment when executed?

```
int m = - 14, n = 3;
printf("%d\n", m/n * 10) ;
n = -n;
printf("%d\n", m/n * 10);
```

Programming Exercises

3.1 Given the values of the variables x, y and z, write a program to rotate their values such that x has the value of y, y has the value of z, and z has the value of x.

3.2 Write a program that reads a floating-point number and then displays the right-most digit of the integral part of the number.

3.3 Modify the above program to display the two right-most digits of the integral part of the number.

3.4 Write a program that will obtain the length and width of a rectangle from the user and compute its area and perimeter.

3.5 Given an integer number, write a program that displays the number as follows:

First line : all digits
Second line : all except first digit
Third line : all except first two digits
.....
Last line : The last digit

For example, the number 5678 will be displayed as:

5 6 7 8
6 7 8
7 8
8

3.6 The straight-line method of computing the yearly depreciation of the value of an item is given by

$$\text{Depreciation} = \frac{\text{Purchase Price} - \text{Salvage Value}}{\text{Years of Service}}$$

Write a program to determine the salvage value of an item when the purchase price, years of service, and the annual depreciation are given.

3.7 Write a program that will read a real number from the keyboard and print the following output in one line:

Smallest integer not less than the number	The given number	Largest integer not greater than the number

3.8 The total distance travelled by a vehicle in t seconds is given by

$$\text{distance} = ut + (at^2)/2$$

Where u is the initial velocity (metres per second), a is the acceleration (metres per second ^{2}). Write a program to evaluate the distance travelled at regular intervals of time, given the values of u and a. The program should provide the flexibility to the user to select his own time intervals and repeat the calculations for different values of u and a.

3.9 In inventory management, the Economic Order Quantity for a single item is given by

$$\text{EOQ} = \sqrt{\frac{2 \times \text{demand rate} \times \text{setup costs}}{\text{holding cost per item per unit time}}}$$

and the optimal Time Between Orders

$$\text{TBO} = \sqrt{\frac{2 \times \text{setup costs}}{\text{demand rate} \times \text{holding cost per item per unit time}}}$$

Write a program to compute EOQ and TBO, given demand rate (items per unit time), setup costs (per order), and the holding cost (per item per unit time).

3.10 For a certain electrical circuit with an inductance L and resistance R, the damped natural frequency is given by

$$\text{Frequency} = \sqrt{\frac{1}{LC} - \frac{R^2}{4C^2}}$$

It is desired to study the variation of this frequency with C (capacitance). Write a program to calculate the frequency for different values of C starting from 0.01 to 0.1 in steps of 0.01.

3.11 Write a program to read a four digit integer and print the sum of its digits.
Hint: Use / and % operators.

3.12 Write a program to print the size of various data types in C.

3.13 Given three values, write a program to read three values from keyboard and print out the largest of them without using **if** statement.

3.14 Write a program to read two integer values m and n and to decide and print whether m is a multiple of n.

3.15 Write a program to read three values using **scanf** statement and print the following results:

(a) Sum of the values
(b) Average of the three values
(c) Largest of the three
(d) Smallest of the three

3.16 The cost of one type of mobile service is Rs. 250 plus Rs. 1.25 for each call made over and above 100 calls. Write a program to read customer codes and calls made and print the bill for each customer.

3.17 Write a program to print a table of **sin** and **cos** functions for the interval from 0 to 180 degrees in increments of 15 as shown below.

x (degrees)	*sin (x)*	*cos (x)*
0	.....	.....
15	.....	.....
...		
...		
180	.....	.....

3.18 Write a program to compute the values of square-roots and squares of the numbers 0 to 100 in steps 10 and print the output in a tabular form as shown below.

Number	*Square-root*	*Square*
0	0	0
100	10	10000

3.19 Write a program that determines whether a given integer is odd or even and displays the number and description on the same line.

3.20 Write a program to illustrate the use of cast operator in a real life situation.

4

Managing Input and Output Operations

4.1 INTRODUCTION

Reading, processing, and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as *information* or *results*, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as **x = 5; a = 0;** and so on. Another method is to use the input function **scanf** which can read data from a keyboard. We have used both the methods in most of our earlier example programs. For outputting results we have used extensively the function **printf** which sends results out to a terminal.

Unlike other high-level languages, C does not have any built-in input/output statements as part of its syntax. All input/output operations are carried out through function calls such as **printf** and **scanf**. There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library. In this chapter we shall discuss some common I/O functions that can be used on many machines without any change. However, one should consult the system reference manual for exact details of these functions and also to see what other functions are available.

It may be recalled that we have included a statement

```
#include <math.h>
```

in the Sample Program 5 in Chapter 1, where a math library function cos(x) has been used. This is to instruct the compiler to fetch the function cos(x) from the math library, and that it is not a part of C language. Similarly, each program that uses a standard input/output function must contain the statement

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

at the beginning. However, there might be exceptions. For example, this is not necessary for the functions **printf** and **scanf** which have been defined as a part of the C language.

The file name **stdio.h** is an abbrevation for *standard input-output header* file. The instruction **#include <*stdio.h*> tells the compiler 'to search for a file named stdio.h and place its contents at this point in the program'. The contents of the header file become part of the source code when it is compiled.

4.2 READING A CHARACTER

The simplest of all input/output operations is reading a character from the 'standard input' unit (usually the keyboard) and writing it to the 'standard output' unit (usually the screen). Reading a single character can be done by using the function **getchar.** (This can also be done with the help of the **scanf** function which is discussed in Section 4.4.) The **getchar** takes the following form:

***variable_name* = getchar( );**

variable_name is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function. Since **getchar** is used on the right-hand side of an assignment statement, the character value of **getchar** is in turn assigned to the variable name on the left. For example

```
char name;
name = getchar();
```

Will assign the character 'H' to the variable **name** when we press the key H on the keyboard. Since **getchar** is a function, it requires a set of parentheses as shown.

Example 4.1 The program in Fig. 4.1 shows the use of **getchar** function in an interactive environment.

The program displays a question of YES/NO type to the user and reads the user's response in a single character (Y or N). If the response is Y or y, it outputs the message

My name is BUSY BEE

otherwise, outputs

You are good for nothing

NOTE: There is one line space between the input text and output message.

Program

```
#include <stdio.h>
main()
{
   char answer;
   printf("Would you like to know my name?\n");

   printf("Type Y for YES and N for NO: ");
   answer = getchar(); /* .... Reading a character...*/
```

```
        if(answer == 'Y' || answer == 'y')
          printf("\n\nMy name is BUSY BEE\n");
        else
          printf("\n\nYou are good for nothing\n");
    }
```

Output

```
Would you like to know my name?
Type Y for YES and N for NO: Y

My name is BUSY BEE
Would you like to know my name?
Type Y for YES and N for NO: n

You are good for nothing
```

Fig. 4.1 *Use of* **getchar** *function to read a* **character** *from keyboard*

The **getchar** function may be called successively to read the characters contained in a line of text. For example, the following program segment reads characters from keyboard one after another until the 'Return' key is pressed.

```
--------

--------
char character;
character = ' ';
while(character != '\n')
{
    character = getchar();
}
--------

--------
```

WARNING

The **getchar()** function accepts any character keyed in. This includes RETURN and TAB. This means when we enter single character input, the newline character is waiting in the input queue after **getchar()** returns. This could create problems when we use **getchar()** in a loop interactively. A dummy **getchar()** may be used to 'eat' the unwanted newline character. We can also use the **fflush** function to flush out the unwanted characters.

NOTE: We shall be using decision statements like **if, if...else** and **while** extensively in this chapter. They are discussed in detail in Chapters 5 and 6.

Table 4.1 *Character Test Functions*

Function	*Test*
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case letter?

4.3 WRITING A CHARACTER

Like **getchar**, there is an analogous function **putchar** for writing characters one at a time to the terminal. It takes the form as shown below:

putchar ***(variable_name);***

where ***variable_name*** is a type **char** variable containing a character. This statement displays the character contained in the ***variable_name*** at the terminal. For example, the statements

```
answer = 'Y';
putchar (answer);
```

will display the character Y on the screen. The statement

```
putchar ('\n');
```

would cause the cursor on the screen to move to the beginning of the next line.

Example 4.3 A program that reads a character from keyboard and then prints it in reverse case is given in Fig. 4.3. That is, if the input is upper case, the output will be lower case and vice versa.

The program uses three new functions: **islower, toupper**, and **tolower**. The function **islower** is a conditional function and takes the value TRUE if the argument is a lowercase alphabet; otherwise takes the value FALSE. The function **toupper** converts the lowercase argument into an uppercase alphabet while the function **tolower** does the reverse.

Program

```
#include <stdio.h>
#include <ctype.h>
main()
{
   char alphabet;
   printf("Enter an alphabet");
   putchar('\n'); /* move to next line */
   alphabet = getchar();
   if (islower(alphabet))
```

```
            putchar(toupper(alphabet));/* Reverse and display */
          else
            putchar(tolower(alphabet)); /* Reverse and display */
      }
Output
          Enter an alphabet
          a
          A
          Enter an alphabet
          Q
          q
          Enter an alphabet
          z
          Z
```

Fig. 4.3 *Reading and writing of alphabets in reverse case*

4.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

```
15.75 123 John
```

This line contains three pieces of data, arranged in a particular form. Such data has to be read conforming to the format of its appearance. For example, the first part of the data should be read into a variable **float**, the second into **int**, and the third part into **char**. This is possible in C using the **scanf** function. (**scanf** means *scan* formatted.)

We have already used this input function in a number of examples. Here, we shall explore all of the options that are available for reading the formatted data with **scanf** function. The general form of **scanf** is

scanf ("control string", arg1, arg2, ... argn);

The *control string* specifies the field format in which the data is to be entered and the arguments *arg*1, *arg2*, ..., *argn* specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string (also known as *format string*) contains field specifications, which direct the interpretation of input data. It may include:

- Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an *optional* number, specifying the field width.
- Blanks, tabs, or newlines.

Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional. The discussions that follow will clarify these concepts.

Inputting Integer Numbers

The field specification for reading an integer number is:

% w sd

The percentage sign (%) indicates that a conversion specification follows. ***w*** is an integer number that specifies the *field width* of the number to be read and **d**, known as data type character, indicates that the number to be read is in integer mode. Consider the following example:

```
scanf ("%2d %5d", &num1, &num2);
```

Data line:

```
50  31426
```

The value 50 is assigned to **num1** and 31426 to **num2**. Suppose the input data is as follows:

```
31426 50
```

The variable **num1** will be assigned 31 (because of %2d) and **num2** will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next **scanf** call. This kind of errors may be eliminated if we use the field specifications without the field width specifications. That is, the statement

```
scanf("%d %d", &num1, &num2);
```

will read the data

```
31426 50
```

correctly and assign 31426 to **num1** and 50 to **num2.**

Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators. When the **scanf** function searches the input data line for a value to be read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, **scanf** may skip reading further input.

When the **scanf** reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.

An input field may be skipped by specifying * in the place of field width. For example, the statement

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

will assign the data

```
123 456 789
```

as follows:

```
123 to a
456 skipped (because of *)
789 to b
```

The data type character **d** may be preceded by 'l' (letter ell) to read long integers and **h** to read short integers.

NOTE: We have provided white space between the field specifications. These spaces are not necessary with the numeric input, but it is a good practice to include them.

Example 4.4 Various input formatting options for reading integers are experimented in the program shown in Fig. 4.4.

```
Program
      main()
      {
         int a,b,c,x,y,z;
         int p,q,r;
         printf("Enter three integer numbers\n");
         scanf("%d %*d %d",&a,&b,&c);
         printf("%d %d %d \n\n",a,b,c);
         printf("Enter two 4-digit numbers\n");
         scanf("%2d %4d",&x,&y);
         printf("%d %d\n\n", x,y);
         printf("Enter two integers\n");
         scanf("%d %d", &a,&x);
         printf("%d %d \n\n",a,x);
         printf("Enter a nine digit number\n");
         scanf("%3d %4d %3d",&p,&q,&r);
         printf("%d %d %d \n\n",p,q,r);
         printf("Enter two three digit numbers\n");
         scanf("%d %d",&x,&y);
         printf("%d %d",x,y);
      }
Output
         Enter three integer numbers
         1 2 3
         1 3 -3577
         Enter two 4-digit numbers
         6789 4321
         67 89
         Enter two integers
         44 66
         4321 44
      Enter a nine-digit number
      123456789
      66 1234 567
      Enter two three-digit numbers
      123 456
      89 123
```

Fig. 4.4 *Reading integers using* ***scanf***

The first **scanf** requests input data for three integer values **a, b,** and **c,** and accordingly three values 1, 2, and 3 are keyed in. Because of the specification %*d the value 2 has been skipped and 3 is assigned to the variable **b**. Notice that since no data is available for **c**, it contains garbage.

The second **scanf** specifies the format %2d and %4d for the variables **x** and **y** respectively. Whenever we specify field width for reading integer numbers, the input numbers should not contain more digits that the specified size. Otherwise, the extra digits on the right-hand side will be truncated and assigned to the next variable in the list. Thus, the second **scanf** has truncated the four digit number 6789 and assigned 67 to **x** and 89 to **y**. The value 4321 has been assigned to the first variable in the immediately following **scanf** statement.

NOTE: It is legal to use a non-whitespace character between field specifications. However, the **scanf** expects a matching character in the given location. For example,

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

accepts input like

```
123-456
```

to assign 123 to a and 456 to b.

Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using the simple specification **%f** for both the notations, namely, decimal point notation and exponential notation. For example, the statement

```
scanf("%f %f %f", &x, &y, &z);
```

with the input data

```
475.89 43.21E-1 678
```

will assign the value 475.89 to **x**, 4.321 to **y**, and 678.0 to **z**. The input field specifications may be separated by any arbitrary blank spaces.

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f.** A number may be skipped using **%*f** specification.

Example 4.5 Reading of real numbers (in both decimal point and exponential notation) is illustrated in Fig. 4.5.

Program

```
main()
{
   float x,y;
double p,q;
printf("Values of x and y:");
scanf("%f %e", &x, &y);
printf("\n");
printf("x = %f\ny = %f\n\n", x, y);
printf("Values of p and q:");
```

```
        scanf("%lf %lf", &p, &q);
        printf("\n\np = %.12lf\np = %.12e", p,q);
    }
```

Output

```
Values of x and y:12.3456 17.5e-2
x = 12.345600
y = 0.175000

Values of p and q:4.142857142857 18.5678901234567890

p = 4.142857142857
q = 1.856789012346e+001
```

Fig. 4.5 *Reading of* ***real*** *numbers*

Inputting Character Strings

We have already seen how a single character can be read from the terminal using the **getchar** function. The same can be achieved using the **scanf** function also. In addition, a **scanf** function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or %wc

The corresponding argument should be a pointer to a character array. However, %c may be used to read a single character when the argument is a pointer to a **char** variable.

Example 4.6 Reading of strings using **%wc** and **%ws** is illustrated in Fig. 4.6.

The program in Fig. 4.6 illustrates the use of various field specifications for reading strings. When we use **%wc** for reading a string, the system will wait until the w^{th} character is keyed in. **Note** that the specification **%s** terminates reading at the encounter of a blank space. Therefore, **name2** has read only the first part of "New York" and the second part is automatically assigned to **name3**. However, during the second run, the string "New-York" is correctly assigned to **name2**.

Program

```
main()
{
   int no;
   char name1[15], name2[15], name3[15];
   printf("Enter serial number and name one\n");
   scanf("%d %15c", &no, name1);
   printf("%d %15s\n\n", no, name1);
   printf("Enter serial number and name two\n");
```

```
        scanf("%d %s", &no, name2);
        printf("%d %15s\n\n", no, name2);
        printf("Enter serial number and name three\n");
        scanf("%d %15s", &no, name3);
        printf("%d %15s\n\n", no, name3);
    }
```

Output

```
Enter serial number and name one
1 123456789012345
1 123456789012345r
Enter serial number and name two
2 New York
2              New
Enter serial number and name three
2              York
Enter serial number and name one
1 123456789012
1 123456789012r
Enter serial number and name two
2 New-York
2           New-York
Enter serial number and name three
3 London
3           London
```

Fig. 4.6 *Reading of strings*

Some versions of **scanf** support the following conversion specifications for strings:

```
%[characters]
%[^characters]
```

The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[^characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

Example 4.7 The program in Fig. 4.7 illustrates the function of %( ) specification.

```
Program-A
    main()
    {
       char address[80];
```

```
        printf("Enter address\n");
        scanf("%[a-z]", address);
        printf("%-80s\n\n", address);
    }
```

Output

```
Enter address
new delhi 110002
new delhi
```

Program-B

```
main()
{
    char address[80];
    printf("Enter address\n");
    scanf("%[^\n]", address);
    printf("%-80s", address);
}
```

Output

```
Enter address
New Delhi 110 002
New Delhi 110 002
```

Fig. 4.7 *Illustration of conversion specification%[] for strings*

Reading Blank Spaces

We have earlier seen that %s specifier cannot be used to read strings with blank spaces. But, this can be done with the help of %[ ] specification. Blank spaces may be included within the brackets, thus enabling the **scanf** to read strings with spaces. Remember that the lowercase and uppercase letters are distinct. See Fig. 4.7.

Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items match the control specifications *in order* and *type*. When an attempt is made to read an item that does not match the type expected, the **scanf** function does not read any further and immediately returns the values read. The statement

```
scanf ("%d %c %f %s", &count, &code, &ratio, name);
```

will read the data

```
15 p 1.575 coffee
```

correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.

NOTE: A space before the *%c* specification in the format string is necessary to skip the white space before p.

Detection of Errors in Input

When a **scanf** function completes reading its list, it returns the value of number of items that are successfully read. This value can be used to test whether any errors occurred in reading the input. For example, the statement

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

will return the value 3 if the following data is typed in:

```
20 150.25 motor
```

and will return the value 1 if the following line is entered

```
20 motor 150.25
```

This is because the function would encounter a string when it was expecting a floating-point value, and would therefore terminate its scan after reading the first value.

Example 4.8 The program presented in Fig.4.8 illustrates the testing for correctness of reading of data by **scanf** function.

The function **scanf** is expected to read three items of data and therefore, when the values for all the three variables are read correctly, the program prints out their values. During the third run, the second item does not match with the type of variable and therefore the reading is terminated and the error message is printed. Same is the case with the fourth run.

In the last run, although data items do not match the variables, no error message has been printed. When we attempt to read a real number for an **int** variable, the integer part is assigned to the variable, and the truncated decimal part is assigned to the next variable.

NOTE: The character '2' is assigned to the character variable c.

Program

```
main()
{
   int a;
   float b;
   char c;
   printf("Enter values of a, b and c\n");
   if (scanf("%d %f %c", &a, &b, &c) == 3)
     printf("a = %d b = %f c = %c\n" , a, b, c);
   else
     printf("Error in input.\n");
}
```

Output

```
Enter values of a, b and c
    12 3.45 A
    a = 12    b = 3.450000     c = A
    Enter values of a, b and c
    23 78 9
    a = 23    b = 78.000000    c = 9
    Enter values of a, b and c
    8 A 5.25
    Error in input.
    Enter values of a, b and c
    Y 12 67
    Error in input.
    Enter values of a, b and c
    15.75 23 X
    a = 15    b = 0.750000     c = 2
```

Fig. 4.8 *Detection of errors in* ***scanf*** *input*

Commonly used **scanf** format codes are given in Table 4.2

Table 4.2 *Commonly used scanf Format Codes*

Code	*Meaning*
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

The following letters may be used as prefix for certain conversion characters.

- h for short integers
- l for long integers or double
- L for long double

NOTE: C99 adds some more format codes. See the Appendix "C99 Features".

Points to Remember While Using scanf

If we do not plan carefully, some 'crazy' things can happen with **scanf**. Since the I/O routines are not a part of C language, they are made available either as a separate module of the C

library or as a part of the operating system (like UNIX). New features are added to these routines from time to time as new versions of systems are released. We should consult the system reference manual before using these routines. Given below are some of the general points to keep in mind while writing a **scanf** statement.

1. All function arguments, except the control string, *must* be pointers to variables.
2. Format specifications contained in the control string should match the arguments in order.
3. Input data items must be separated by spaces and must match the variables receiving the input in the same order.
4. The reading will be terminated, when **scanf** encounters a 'mismatch' of data or a character that is not valid for the value being read.
5. When searching for a value, **scanf** ignores line boundaries and simply looks for the next appropriate character.
6. Any unread data items in a line will be considered as part of the data input line to the next **scanf** call.
7. When the field width specifier *w* is used, it should be large enough to contain the input data size.

Rules for scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a match ing character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The scanf reads until:
  - A whitespace character is found in a numberic specification, or
  - The maximum number of characters have been read or
  - An error is detected, or
  - The end of file is reached

4.5 FORMATTED OUTPUT

We have seen the use of **printf** function for printing captions and numerical results. It is highly desirable that the outputs are produced in such a way that they are understandable and are in an easy-to-use form. It is therefore necessary for the programmer to give careful consideration to the appearance and clarity of the output produced by his program.

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general form of **printf** statement is:

```
printf("control string", arg1, arg2, ....., argn);
```

Control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. *Escape sequence* characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments *arg1, arg2, ..., argn* are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number, order and type with the format specifications.

A simple format specification has the following form:

```
% w.p type-specifier
```

where *w* is an integer number that specifies the total number of columns for the output value and *p* is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string. Both ***w*** and *p* are optional. Some examples of formatted **printf** statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

printf never supplies a *newline* automatically and therefore multiple **printf** statements may be used to build one line of output. A *newline* can be introduced by the help of a newline character '\n' as shown in some of the examples above.

Output of Integer Numbers

The format specification for printing an integer number is:

```
% w d
```

where ***w*** specifies the minimum field width for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification. ***d*** specifies that the value to be printed is an integer. The number is written *right-justified* in the given field width. Leading blanks will appear as necessary. The following examples illustrate the output of the number 9876 under different formats:

Format	*Output*
printf("%d", 9876)	9876
printf("%6d", 9876)	␣␣9876
printf("%2d", 9876)	9876
printf("%-6d", 9876)	9876␣␣
printf("%06d", 9876)	009876

It is possible to force the printing to be left-*justified* by placing a *minus* sign directly after the % character, as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier as shown in the last item above. The minus (–) and zero (0) are known as *flags*.

Long integers may be printed by specifying **ld** in the place of **d** in the format specification. Similarly, we may use **hd** for printing short integers.

Example 4.9 The program in Fig. 4.9 illustrates the output of integer numbers under various formats.

```
Program
    main()
    {
       int m = 12345;
       long n = 987654;

       printf("%d\n",m);
       printf("%10d\n",m);
       printf("%010d\n",m);
       printf("%-10d\n",m);
       printf("%10ld\n",n);
       printf("%10ld\n",-n);
    }
Output
       12345
            12345
       0000012345
       12345
            987654
          - 987654
```

Fig. 4.9 *Formatted output of integers*

Output of Real Numbers

The output of a real number may be displayed in decimal notation using the following format specification:

% w.p f

The integer ***w*** indicates the minimum number of positions that are to be used for the display of the value and the integer ***p*** indicates the number of digits to be displayed after the decimal point (*precision).* The value, when displayed, is *rounded to p decimal places* and printed *right-justified* in the field of *w* columns. Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form [–] mmm-nnn.

We can also display a real number in exponential notation by using the specification:

% w.p e

The display takes the form

[-] m.nnnne[±]xx

where the length of the string of n's is specified by the precision *p*. The default precision is 6. The field width **w** should satisfy the condition.

$$w \geq p+7$$

The value will be rounded off and printed right justified in the field of **w** columns.

Padding the leading blanks with zeros and printing with *left-justification* are also possible by using flags 0 or – before the field width specifier **w**.

The following examples illustrate the output of the number y = 98.7654 under different format specifications:

Format	*Output*
printf("%7.4f",y)	9 8 . 7 6 5 4
printf("%7.2f",y)	␣ ␣ 9 8 . 7 7
printf("%-7.2f",y)	9 8 . 7 7 ␣ ␣
printf("%f",y)	9 8 . 7 6 5 4
printf("%10.2e",y)	␣ ␣ 9 . 8 8 e + 0 1
printf("%11.4e",-y)	– 9 . 8 7 6 5 e + 0 1
printf("%-10.2e",y)	9 . 8 8 e + 0 1 ␣ ␣
printf("%e",y)	9 . 8 7 6 5 4 0 e + 0 1

Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:

printf("%*.*f", width, precision, number);

In this case, both the field width and the precision are given as arguments which will supply the values for **w** and **p**. For example,

```
printf("%*.*f",7,2,number);
```

is equivalent to

```
printf("%7.2f",number);
```

The advantage of this format is that the values for *width* and *precision* may be supplied at run time, thus making the format a *dynamic* one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
........
........
printf("%*.*f", width, precision, number);
```

Example 4.10 All the options of printing a real number are illustrated in Fig. 4.10.

```
Program
    main()
    {
      float y = 98.7654;
      printf("%7.4f\n", y);
      printf("%f\n", y);
      printf("%7.2f\n", y);
      printf("%-7.2f\n", y);
      printf("%07.2f\n", y);
      printf("%*.*f", 7, 2, y);
      printf("\n");
      printf("%10.2e\n", y);
      printf("%12.4e\n", -y);
      printf("%-10.2e\n", y);
      printf("%e\n", y);
    }
Output
      98.7654
      98.765404
      98.77
      98.77
      0098.77
      98.77
      9.88e+001
      -9.8765e+001
      9.88e+001
      9.876540e+001
```

Fig. 4.10 *Formatted output of* ***real*** *numbers*

Printing of a Single Character

A single character can be displayed in a desired position using the format:

%wc

The character will be displayed *right-justified* in the field of *w* columns. We can make the display *left-justified* by placing a minus sign before the integer w. The default value for w is 1.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

%w.ps

where *w* specifies the field width for display and *p* instructs that only the first p characters of the string are to be displayed. The display is *right-justified*.

The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).

Specification	Output
	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
%s	N E W D E L H I 1 1 0 0 0 1
%20s	N E W D E L H I 1 1 0 0 0 1
%20.10s	N E W D E L H I
%.5s	N E W D
%-20.10s	N E W D E L H I
%5s	N E W D E L H I 1 1 0 0 0 1

Example 4.11 Printing of characters and strings is illustrated in Fig. 4.11.

Program

```
main()
{
   char x = 'A';
   char name[20] = "ANIL KUMAR GUPTA";
   printf("OUTPUT OF CHARACTERS\n\n");
   printf("%c\n%3c\n%5c\n", x,x,x);
   printf("%3c\n%c\n", x,x);
```

```
        printf("\n");
        printf("OUTPUT OF STRINGS\n\n");
        printf("%s\n", name);
        printf("%20s\n", name);
        printf("%20.10s\n", name);
        printf("%.5s\n", name);
        printf("%-20.10s\n", name);
        printf("%5s\n", name);
    }
```

Output

```
OUTPUT OF CHARACTERS
A
   A
      A
   A
A
OUTPUT OF STRINGS
ANIL KUMAR GUPTA
   ANIL KUMAR GUPTA
         ANIL KUMAR
ANIL
ANIL KUMAR
ANIL KUMAR GUPTA
```

Fig. 4.11 *Printing of characters and strings*

Mixed Data Output

It is permitted to mix data types in one **printf** statement. For example, the statement of the type

```
printf("%d %f %s %c", a, b, c, d);
```

is valid. As pointed out earlier, **printf** uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type. If there are not enough variables or if they are of the wrong type, the output results will be incorrect.

Table 4.3 *Commonly used printf Format Codes*

Code	*Meaning*
%c	print a single character
%d	print a decimal integer
%e	print a floating point value in exponent form
%f	print a floating point value without exponent
%g	print a floating point value either e-type or f-type depending on

Code	Meaning
%i	print a signed decimal integer
%o	print an octal integer, without leading zero
%s	print a string
%u	print an unsigned decimal integer
%x	print a hexadecimal integer, without leading Ox

The following letters may be used as prefix for certain conversion characters.

h for short integers
l for long integers or double
L for long double.

Table 4.4 *Commonly used* **Output** *Format Flags*

Flag	Meaning
–	Output is left-justified within the field. Remaining field will be blank.
+	+ or – will precede the signed numeric item.
0	Causes leading zeros to appear.
# (with o or x)	Causes octal and hex items to be preceded by O and Ox, respectively.
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zeros in g-type conversion.

NOTE: C99 adds some more format codes. See the Appendix " C99 Features".

Enhancing the Readability of Output

Computer outputs are used as information for analysing certain relationships between variables and for making decisions. Therefore the correctness and clarity of outputs are of utmost importance. While the correctness depends on the solution procedure, the clarity depends on the way the output is presented. Following are some of the steps we can take to improve the clarity and hence the readability and understandability of outputs.

1. Provide enough blank space between two numbers.
2. Introduce appropriate headings and variable names in the output.
3. Print special messages whenever a peculiar condition occurs in the output.
4. Introduce blank lines between the important sections of the output.

The system usually provides two blank spaces between the numbers. However, this can be increased by selecting a suitable field width for the numbers or by introducing a 'tab' character between the specifications. For example, the statement

```
printf("a = %d\t b = %d", a, b);
```

will provide four blank spaces between the two fields. We can also print them on two separate lines by using the statement

```
printf("a = %d\n b = %d", a, b);
```

Messages and headings can be printed by using the character strings directly in the **printf** statement. Examples:

(i) To print the data left-justified, we must use _____in the field specification.
(j) The specifier _____ prints floating-point values in the scientific notation.

4.3 Distinguish between the following pairs:

(a) ***getchar*** and ***scanf*** functions.
(b) %s and %c specifications for reading.
(c) %s and %[ ] specifications for reading.
(d) %g and %f specification for printing.
(e) %f and %e specifications for printing.

4.4 Write **scanf** statements to read the following data lists:

(a) 78 B 45
(b) 123 1.23 45A
(c) 15-10-2002
(d) 10 TRUE 20

4.5 State the outputs produced by the following **printf** statements.

(a) printf ("%d%c%f", 10, 'x', 1.23);
(b) printf ("%2d %c %4.2f", 1234,, 'x', 1.23);
(c) printf ("%d\t%4.2f", 1234, 456);
(d) printf ("\"%08.2f\"", 123.4);
(e) printf ("%d%d %d", 10, 20);

For questions 4.6 to 4.10 assume that the following declarations have been made in the program:

```
int year, count;
float amount, price;
char code, city[10];
double root;
```

4.6 State errors, if any, in the following input statements.

(a) scanf("%c%f%d", city, &price, &year);
(b) scanf("%s%d", city, amount);
(c) scanf("%f, %d, &amount, &year);
(d) scanf(\n"%f", root);
(e) scanf("%c %d %ld", *code, &count, Root);

4.7 What will be the values stored in the variables **year** and **code** when the data

1988, x

is keyed in as a response to the following statements:

(a) scanf("%d %c", &year, &code);
(b) scanf("%c %d", &year, &code);
(c) scanf("%d %c", &code, &year);
(d) scanf("%s %c", &year, &code);

4.8 The variables **count, price**, and **city** have the following values:

count <—— 1275
price <—— –235.74
city <—— Cambridge

Show the exact output that the following output statements will produce:

(a) printf("%d %f", count, price);
(b) printf("%2d\n%f", count, price);
(c) printf("%d %f", price, count);
(d) `printf("%10dxxxx%5.2f",count, price);`

The McGraw-Hill Companies

PROGRAMMING IN

ANSI C 4E

This, the fourth edition of *Programming in ANSI C*, presents a detailed exposition of C in an extremely simple and lucid style that has now become the hallmark of this book. As always, the concept of 'learning by example' has been stressed throughout the book. The various features of the language have been systematically discussed including the new features added to C99.

Salient features:

- *Codes with comments are provided throughout the book to illustrate how the various features of the language are put together to accomplish specified tasks.*
- *Case Studies at the end of the chapters illustrate real-life applications using C.*
- *Programming Projects discussed in the appendix give insight on how to integrate the various features of C when handling large programs.*
- *'Just Remember' section at the end of the chapters lists out helpful hints and possible problem areas.*
- *Guidelines for developing efficient C programs are given in the last chapter, together with a list of some common mistakes that a less experienced C programmer could make.*

New to this edition:

- *Two new programming projects on 'INVENTORY' and 'RECORD ENTRY'.*
- *Coverage of the new features added to C99.*

Dedicated Website: ***http://www.mhhe.com/balagurusamy/ansic4ed***

visit us at www.tatamcgrawhill.com
ISBN :13 978-0-07-064822-7
ISBN :10 0-07-064822-0
9 780070 648227

Tata McGraw-Hill