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 *statements* 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.

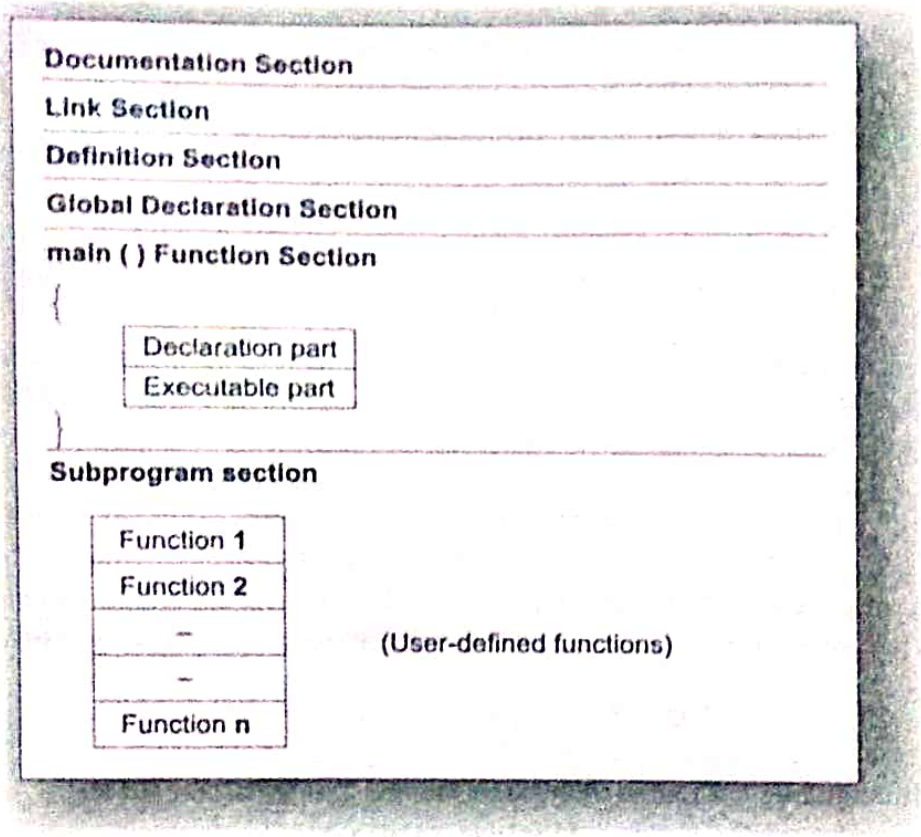

Fig. 1.9 *An overview of a C program*

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(;).

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.

2 CONSTANTS, VARIABLES, AND DATA TYPES

Key Terms

Identifiers I Constant I String constant I Variable I scanf

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	All decimal digits 0 9
Lowercase a.....z	

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.

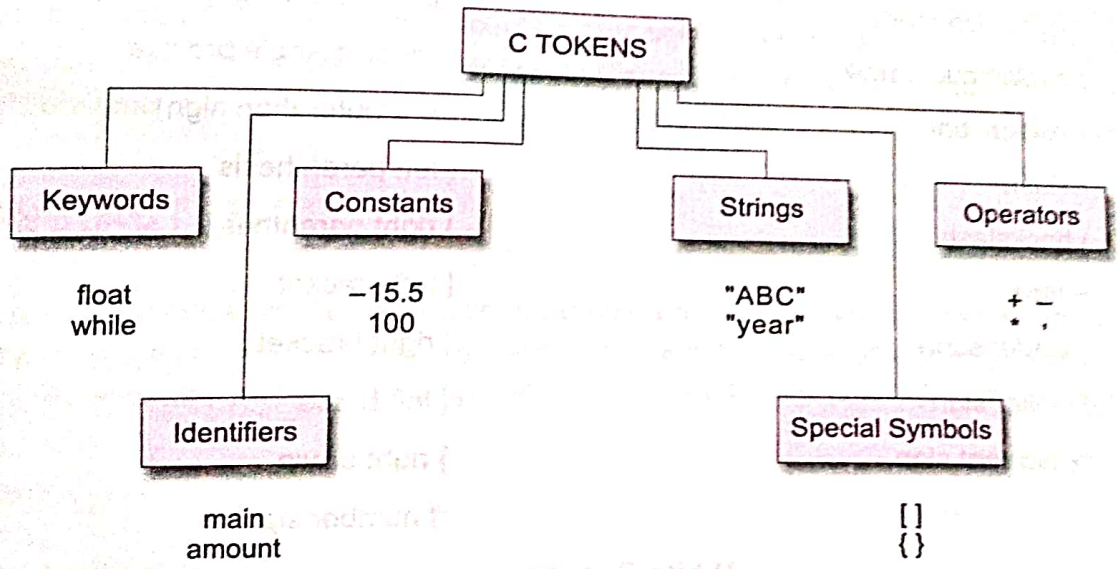

Fig. 2.1 *C tokens and examples*

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 Keyword*

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.

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

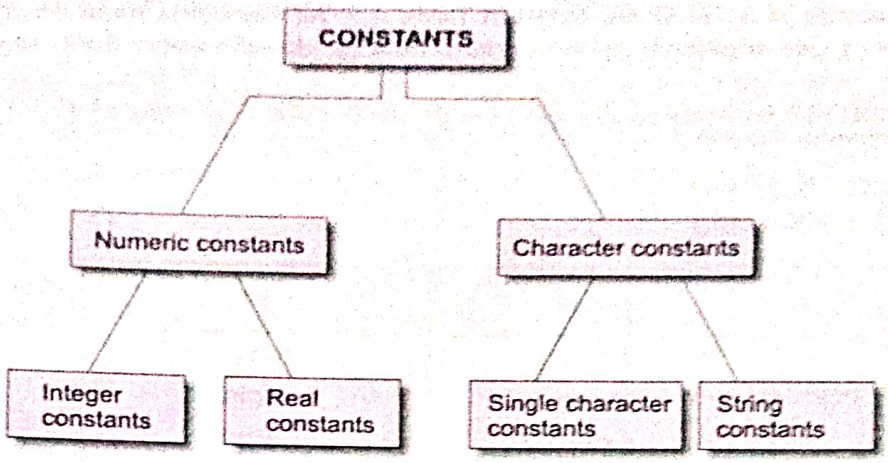

Fig. 2.2 *Basic types of C constants*

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.

Program 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 3777
```

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 for 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
'10'	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 variable **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 "C99Fatures".*

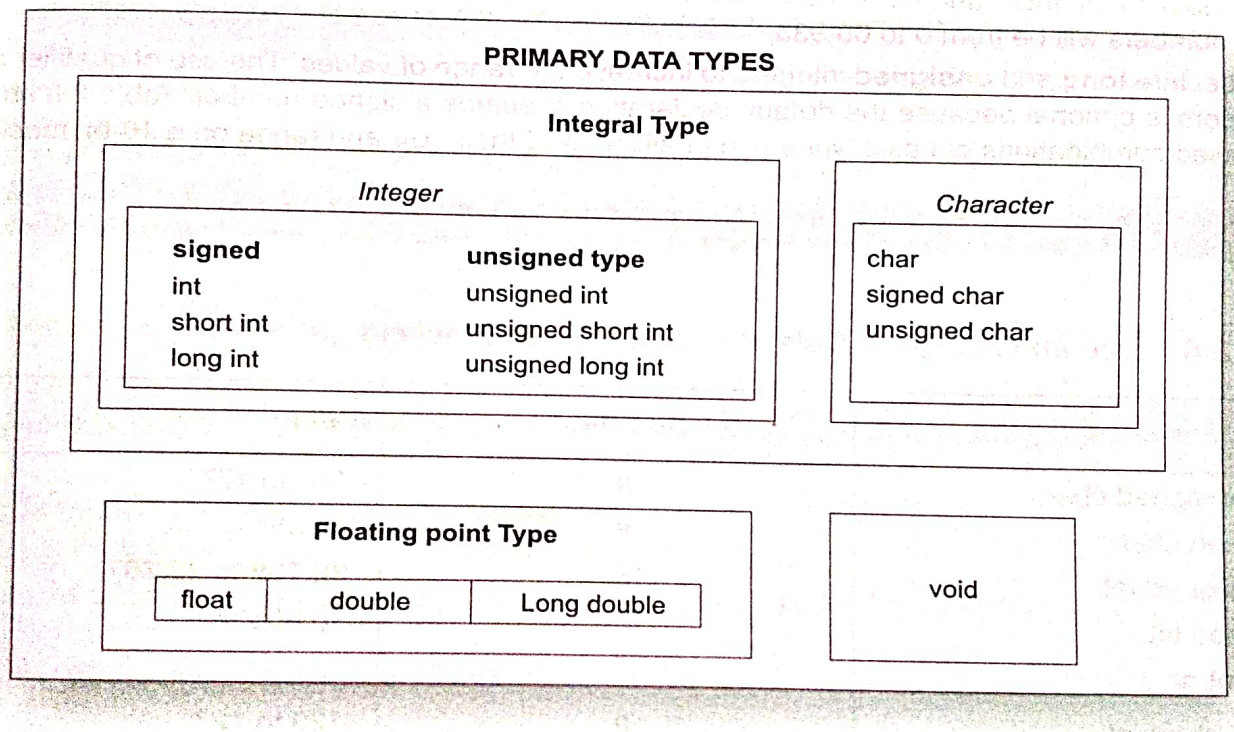

Fig. 2.4 *Primary data types in C*

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.

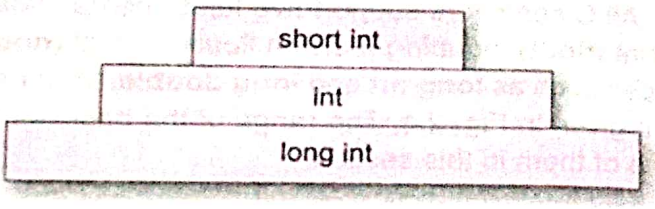

Fig. 2.5 *Integer types*

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 Fig. 2.6.

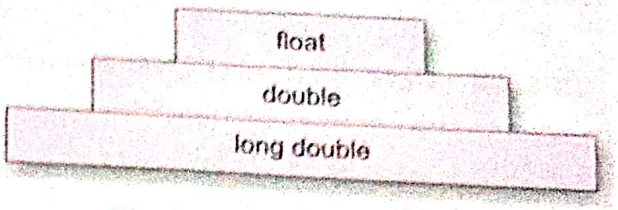

Fig. 2.6 *Floating-point types*

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;
```

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.

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,

```
value = amount + inrate * amount;
while (year <= PERIOD)
{
   ....
   ....
   year = year + 1;
}
```

In the first statement, the numeric value stored in the variable **inrate** is multiplied by the value stored in **amount** and the product is added to **amount**. The result is stored in the variable value. This process is possible only if the variables **amount** and inrate have already been given values. The variable **value** is called the *target variable*. While all the variables are declared for their type, the variables that are used in expressions (on the right side of equal (=) sign of a computational statement) *must* be assigned values before they are encountered in the program. Similarly, the variable **year** and the symbolic constant **PERIOD** in the **while** statement must be assigned values before this statement is encountered.

Assignment Statement

Values can be assigned to variables using the assignment operator = as follows:

variable_name = constant;

We have already used such statements in Chapter 1. Further examples are:

```
initial_value    = 0;
final_value      = 100;
balance          = 75.84;
yes              = 'x';
```

C permits multiple assignments in one line. For example

```
initial_value = 0; final_value = 100;
```

are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement

```
year = year + 1;
```

means that the 'new value' of **year** is equal to the 'old value' of **year** plus 1.

During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

```
data-type variable_name = constant;
```

Some examples are:

```
int final_value   = 100;
char yes          = 'x';
double balance    = 75.84;
```

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.

Program 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.

```
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 no 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

3 OPERATORS AND EXPRESSIONS

Key Terms

Operator | Expression | Integer expression | Real arithmetic | Relational operators | Logical operators | Assignment operators | Bitwise operators | Arithmetic operations

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 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
```

Program 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.

-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.

Program 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.

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.

Program 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.

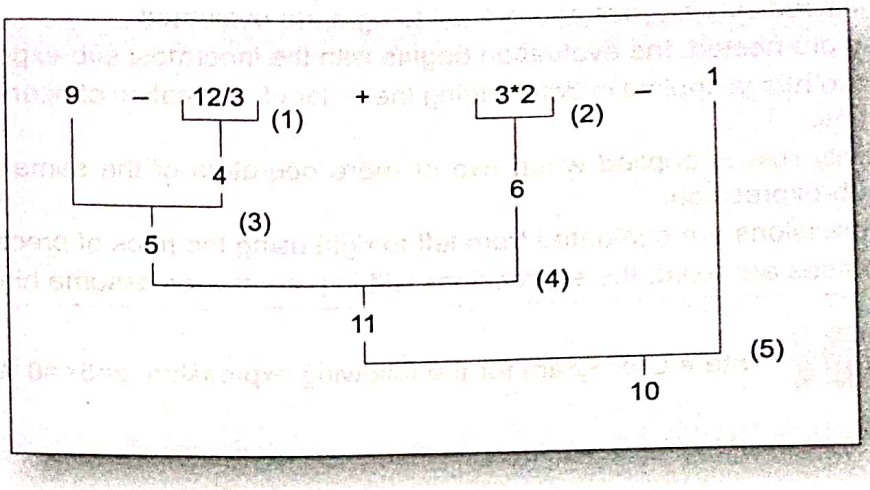

Fig. 3.5 *Illustration of hierarchy of operations*

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

whereas

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

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

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.

Program 3.5 Write a C program for the following expression: a=5<=8 && 6!=5.

```
#include <stdio.h>
#include <conio.h>
   void main()
   {
   int a;
   a = 5<=8 && 6!=5;
   printf("%d", a);
   getch();
   }
Output
   1
```

Fig. 3.6 *Program for the expression: a = 5 < = 8 && 6! = 5*

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.

4 MANAGING INPUT AND OUTPUT OPERATIONS

Key Terms

Formatted input | Control string | Formatted output.

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 abbreviation 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.

Program 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?
```

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.

Program 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 cast*

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 *arg1*, *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 the.*

Program 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.

Program 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.

Program 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.

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.

Program 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 = 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 matching 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** statemen 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)	9 8 7 6
printf("%6d", 9876)	_ _ 9 8 7 6
printf("%2d", 9876)	9 8 7 6

printf("%06d" 9876)

9	8	7	6		

printf("%06d" 9876)

0	0	9	8	7	6

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.

Program 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);
```

Program 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 banks).

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

Program 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
%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.