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	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 (or less than sign)
" quotation mark	> closing angle bracket (or greater than sign)
! exclamation mark	(left parenthesis
| vertical bar	) right parenthesis
/ slash	[left bracket
\ backslash	] right bracket
~ tilde	{ left brace
_ under score	} right brace
$ dollar sign	# number sign
% percent 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.

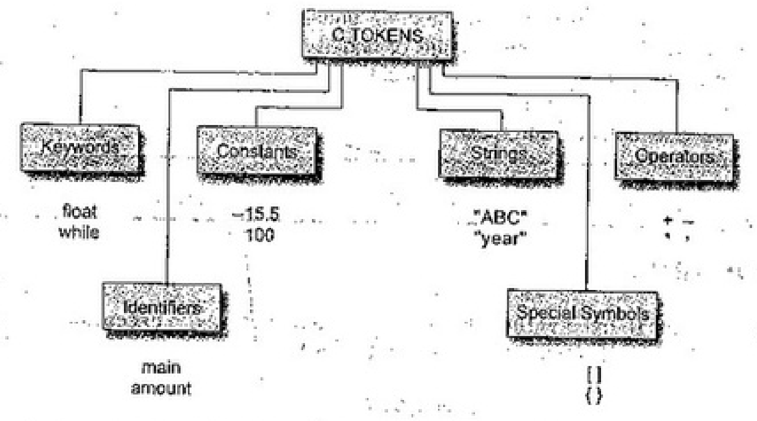


Fig. 2.1 *C tokens and examples*

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.

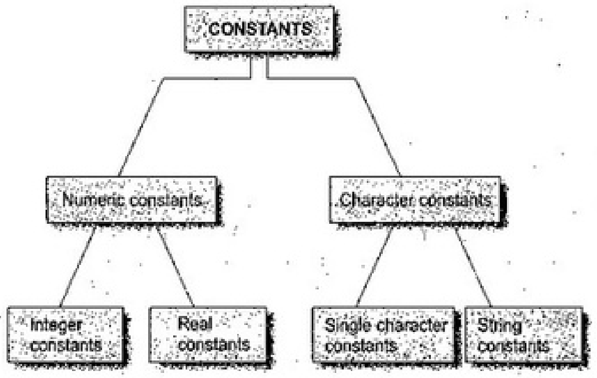


Fig. 2.2 *Basic types of C constants*

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 decimal point and the fractional part. It is possible to omit digits before the decimal point, o 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 exampl the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply b 10^2. The general form is:

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The expo *nent* is an integer number with an optional *plus* or *minus sign*. The letter e separating th mantissa and the exponent can be written in either lowercase or uppercase. Since the expo nent causes the decimal point to "float", this notation is said to represent a real number i *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 ver 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. How ever, the suffixes f or F may be used to force single-precision and l or L to extend doubl 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 en closed 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 th remain unchanged during the execution of a program, a variable may take different valu at different times during execution. In Chapter 1, we used several variables. For instanc we used the variable **amount** in Sample Program 3 to store the value of money at the end 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 it 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 charact
2. ANSI standard recognizes a length of 31 characters. However, length should not normally more than eight characters, since only the first eight characters are treat 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 **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".

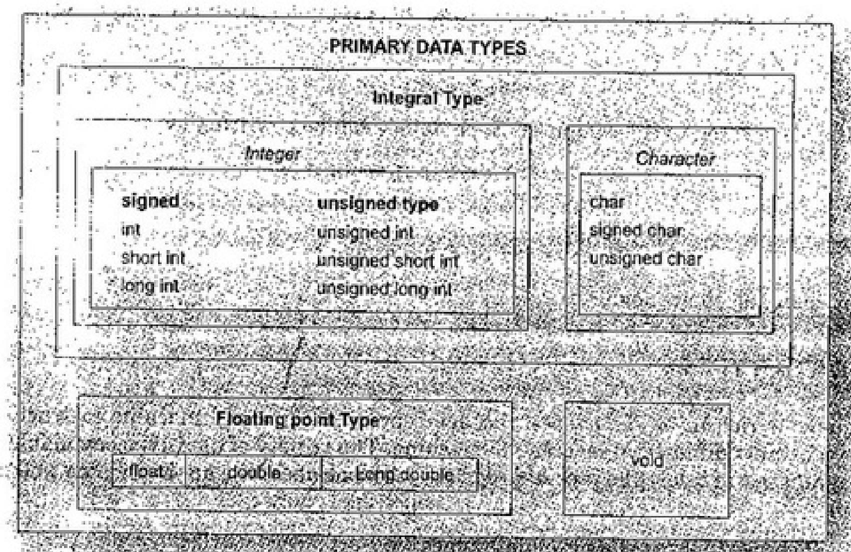


Fig. 2.4 Primary data types in C

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. Gen
erally, integers occupy one word of storage, and since the word sizes of machines vary (typi
cally, 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 +3276
(that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitud
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 thre
classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **un
signed** forms. ANSI C defines these types so that they can be organized from the smallest t
the largest, as shown in Fig. 2.5. For example, **short int** represents fairly small intege
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.

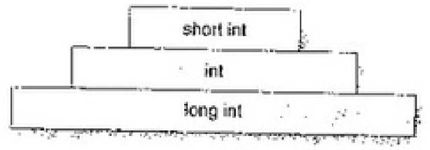


Fig. 2.5 Integer types

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.

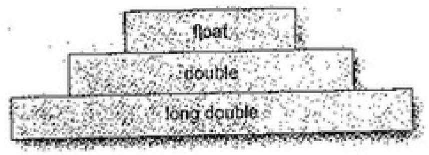


Fig. 2.6 Floating-point types

Void Types

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

Character Types

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

2.8 DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaratio 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 d 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 declaratio 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 respe tively. 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 thal would represent an existing data type. The user-defined data type identifier can later b 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 t the data type. The existing data type may belong to any class of type, including the use defined ones. Remember that the new type is 'new' only in name, but not the data typ **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 declar 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 ca 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 declare inside a function. Local variables are visible and meaningful only inside the functions i which they are declared. They are not known to other functions. Note that the variable **i** ha been declared in both the functions. Any change in the value of **i** in one function does no affect its value in the other.

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

The storage class is another qualifier (like **long** or **unsigned**) that can be added to 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. Automati (**auto**) variables contain undefined values (known as 'garbage') unless they are initializ 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 in *tialization* of more than one variables in one statement using multiple assignment operato 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 seco initializes **x, y,** and **z** with **MAX.** Note that **MAX** is a symbolic constant defined at the begi ning.

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

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

The variables **x** and **p** have been declared as floating-point variables. Note that the w the value of 1.234567890000 that we assigned to **x** is displayed under different output f mats. The value of **x** is displayed as 1.234567880630 under %.12lf format, while the act value assigned is 1.234567890000. This is because the variable **x** has been declared as a flo 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 correct Instead, it contains some garbage. Since this program was run on a 16-bit machine, t maximum value that an **int** variable can store is only 32767. However, the variable **k** (d clared as **unsigned**) has stored the value 54321 correctly. Similarly, the **long int** variabl has stored the value 1234567890 correctly.

The value 9.87654321 assigned to **y** declared as double has been stored correctly but t value is printed as 9.876543 under %lf format. Note that unless specified otherwise, t **printf** function will always display a **float** or **double** value to six decimal places. We w 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
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 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 mo 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 investe 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, representin the value of the mathematical constant **"pi"**. Another example is the total number of stu dents 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. W 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 o calculations or the number 50 to 100 to process the test results of another class. In both th cases, we will have to search throughout the program and explicitly change the value of th constant wherever it has been used. If any value is left unchanged, the program may pr duce disastrous outputs.

Understandability

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

Assignment of such constants to a *symbolic name* frees us from these problems. For exam ple, 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 name at the beginning of the program. Subsequent use of the names **STRENGTH** an **PASS_MARK** in the program has the effect of causing their defined values to be automati cally 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 th 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 th represent?
2.5 What is an unsigned integer constant? What is the significance of declaring a constan 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 o 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-precisi floating point values in his program. How could he achieve this?
2.13 What are enumeration variables? How are they declared? What is the advantag 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 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 wou 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 a
+, –, *, & and <. An *operator* is a symbol that tells the computer to perform certain ma
ematical or logical manipulations. Operators are used in programs to manipulate data a
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.
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 a
built-in data type allowed in C. The unary minus operator, in effect, multiplies its sin
operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Table 3.1 *Arithmetic Operators*

Operator	*Meaning*
[illegible]	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
[illegible]	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 truncation 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
```

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 statem

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

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

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real opera may assume values either in decimal or exponential notation. Since floating point values a rounded to the number of significant digits permissible, the final value is an approximati 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

$$a < b \text{ or } 1 < 20$$

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

$$10 < 20 \text{ is true}$$

but

$$20 < 10 \text{ is false}$$

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, variabl combination of them. Given below are some examples of simple relational expressions a 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 valu c and d.

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

Relational expressions are used in *decision statements* such as **if** and **while** to decid course of action of a running program. We have already used the **while** statement in Cha 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 independentl they behave differently when they are used in expressions on the right-hand side of a 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 statemen 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 opera and then the result is assigned to the variable on left. On the other hand, a postfix operat 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 statemen

```
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 evaluati 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 a discussed in Chapters 10 and 11. ANSI committee has introduced two preprocessor operato known as "string-izing" and "token-pasting" operators (# and ##). They will be discussed Chapter 14.

The Comma Operator

The comma operator can be used to link the related expressions together. A comma-link list of expressions are evaluated *left to right* and the value of *right-most* expression is t 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 valu Since comma operator has the lowest precedence of all operators, the parentheses a 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 num of bytes the operand occupies. The operand may be a variable, a constant or a data ty 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 structur when their sizes are not known to the programmer. It is also used to allocate memory spa dynamically to variables during execution of a program.

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

Notice the way the increment operator **++** works when used in an expression. In the stat ment

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

new value of **a** (= 16) is used thus giving the value 6 to c. That is, a is incremented by 1 befo 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 us 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 expressio evaluated first and the result then replaces the previous value of the variable on the l hand side. All variables used in the expression must be assigned values before evaluatio 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 readabil When these statements are used in a program, the variables a, b, c, and d must be defi before they are used in the expressions.

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

Output of the program also illustrates the effect of presence of parentheses in expressi 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.

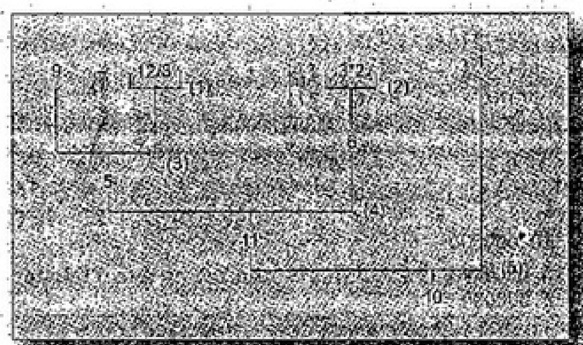


Fig. 3.5 *Illustration of hierarchy of operations*

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 responsibili guarantee that operands are of the correct type and range, and the result may not pr any overflow or underflow.

Example 3.5 Output of the program in Fig. 3.6 shows round-off errors that can oc 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 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 auto cally converts any intermediate values to the proper type so that the expression c evaluated without loosing any significance. This automatic conversion is known as im *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.

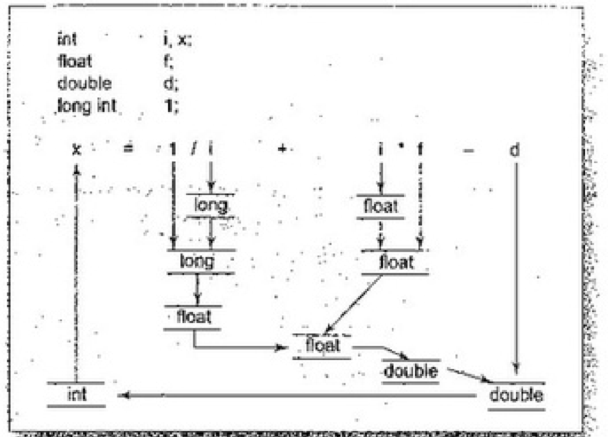


Fig. 3.7 *Process of implicit type conversion*

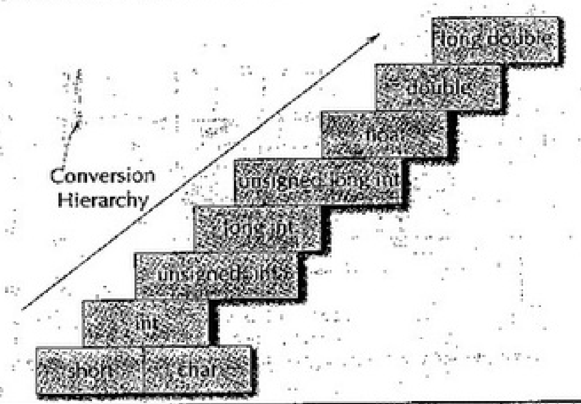
Given below is the sequence of rules that are applied while evaluating expressions.

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 d
precision.

The final result of an expression is converted to the type of the variable on the left
assignment sign before assigning the value to it. However, the following change
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
are instances when we want to force a type conversion in a way that is different from
automatic conversion. Consider, for example, the calculation of ratio of females to males
town.

ratio = female_number/male_number

Since **female_number** and **male_number** are declared as integers in the program
decimal part of the result of the division would be lost and **ratio** would represent a
figure. This problem can be solved by converting locally one of the variables to the float
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 precedenc is used to determine how an expression involving more than one operator is evaluated. Ther are distinct *levels of precedence* and an operator may belong to one of these levels. Th operators at the higher level of precedence are evaluated first. The operators of the sam precedence are evaluated either from 'left to right' or from 'right to left', depending on th level. This is known as the *associativity* property of an operator. Table 3.8 provides complete list of operators, their precedence levels, and their rules of association. The group are listed in the order of decreasing precedence. Rank 1 indicates the highest precedenc level and 15 the lowest. The list also includes those operators, which we have not yet bee discussed.

It is very important to note carefully, the order of precedence and associativity o 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 logica 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 assum 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	Right to left	2
–	Unary minus		
++	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-lif problems. Most of the C compilers support these basic math functions. However, there ar systems that have a more comprehensive math library and one should consult the referenc manual to find out which functions are available. Table 3.9 lists some standard mat 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, square roots cannot be evaluated. In such cases, the roots are said to be imaginary numbe 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 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 evalua first.
- (i) An expression statement is terminated with a period.
- (j) During the evaluation of mixed expressions, an implicit cast is generated au matically.
- (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_______ expressio
- (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 _________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 expr sion 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 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);
```

rogramming 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 ite
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 pr
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 fol
ing 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 p
second ^{2}). Write a program to evaluate the distance travelled at regular interval
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 value
u and a.

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

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

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 ti
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 damp
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). Writ
program to calculate the frequency for different values of C starting from 0.01 to 0.1
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 compu program. Most programs take some data as input and display the processed data known as *information* or *results*, on a suitable medium. So far we have seen two metho providing data to the program variables. One method is to assign values to variables thr the assignment statements such as **x = 5; a = 0;** and so on. Another method is to use the in function **scanf** which can read data from a keyboard. We have used both the method most of our earlier example programs. For outputting results we have used extensively function **printf** which sends results out to a terminal.

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

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 u This is to instruct the compiler to fetch the function cos(x) from the math library, and th is not a part of C language. Similarly, each program that uses a standard input/outp function must contain the statement

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

at the beginning. However, there might be exceptions. For example, this is not necessary 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.

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.

Example 4.2 The program of Fig. 4.2 requests the user to enter a character and displays a message on the screen telling the user whether the character is an alphabet or digit, or any other special character.

This program receives a character from the keyboard and tests whether it is a letter or digit and prints out a message accordingly. These tests are done with the help of the following functions:

```
isalpha(character)
isdigit(character)
```

For example, **isalpha** assumes a value non-zero (TRUE) if the argument **character** contains an alphabet; otherwise it assumes 0 (FALSE). Similar is the case with the function **isdigit**.

```
Program:
    #include <stdio.h>
    #include <ctype.h>
    main()
    {
      char character;
      printf("Press any key\n");
      character = getchar();
      if (isalpha(character) > 0)/* Test for letter */
        printf("The character is a letter.");
      else
        if (isdigit (character) > 0)/* Test for digit */
          printf("The character is a digit.");
        else
          printf("The character is not alphanumeric.");
    }
Output
      Press any key
      h
      The character is a letter.
      Press any key
      5
      The character is a digit.
      Press any key
      *

      The character is not alphanumeric.
```

Fig. 4.2 Program to test the character type

C supports many other similar functions, which are given in Table 4.1. These character functions are contained in the file **ctype.h** and therefore the statement

```
#include <ctype.h>
```

must be included in the program.

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 intege number that specifies the *field width* of the number to be read and **d**, known as data typ character, indicates that the number to be read is in integer mode. Consider the followin 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 a follows:

```
31426 50
```

The variable **num1** will be assigned 31 (because of %2d) and **num2** will be assigned 42 (unread part of 31426). The value 50 that is unread will be assigned to the first variable i the next **scanf** call. This kind of errors may be eliminated if we use the field specification 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 n count as separators. When the **scanf** function searches the input data line for a value to b read, it will always bypass any white space characters.

What happens if we enter a floating point number instead of an integer? The fractiona 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 soo as the number of characters specified by the field width is reached (if specified) or until 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, th 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** t read short integers.

NOTE: We have provided white space between the field specifications. These spaces are not nece sary 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 the [illegible] 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. Som systems accept integers in the place of real numbers and vice versa, and the input data 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 spac before p.

Detection of Errors in Input

When a **scanf** function completes reading its list, it returns the value of number of item that are successfully read. This value can be used to test whether any errors occurred 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-poin 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 reading of data by **scanf** function.

The function **scanf** is expected to read three items of data and therefore, when the values fo all the three variables are read correctly, the program prints out their values. During th third run, the second item does not match with the type of variable and therefore the readin 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 ha been printed. When we attempt to read a real number for an **int** variable, the integer part 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 aft the % character, as shown in the fourth example above. It is also possible to pad with zer the leading blanks by placing a 0 (zero) before the field width specifier as shown in the las 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 specificatio Similarly, we may use **hd** for printing short integers.

Example 4.9 The program in Fig. 4.9 illustrates the output of integer numbers unde 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)	`98.7654`
printf("%7.2f",y)	`  98.77`
printf("%-7.2f",y)	`98.77  `
printf("%f",y)	`98.7654`
printf("%10.2e",y)	`  9.88e+01`
printf("%11.4e",-y)	`-9.8765e+01`
printf("%-10.2e",y)	`9.88e+01  `
printf("%e",y)	`9.876540e+01`

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
%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:

```
printf("\n OUTPUT RESULTS \n");
printf("Code\t Name\t Age\n");
printf("Error in input data\n");

printf("Enter your name\n");
```

Just Remember

- While using **getchar** function, care should be exercised to clear an unwanted characters in the input stream.
- Do not forget to include **<stdio.h>** headerfiles when using functions fro standard input/output library.
- Do not forget to include **<ctype.h>** header file when using functions from character handling library.
- Provide proper field specifications for every variable to be read or printe
- Enclose format control strings in double quotes.
- Do not forget to use address operator & for basic type variables in th input list of **scanf**.
- Use double quotes for character string constants.
- Use single quotes for single character constants.
- Provide sufficient field with to handle a value to be printed.
- Be aware of the situations where output may be imprecise due formatting.
- Do not specify any precision in input field specifications.
- Do not provide any white-space at the end of format string of a scan statement.
- Do not forget to close the format string in the **scanf** or **printf** statem with double quotes.
- Using an incorrect conversion code for data type being read or written wi result in runtime error.
- Do not forget the comma after the format string in **scanf** and **prin** statements.
- Not separating read and write arguments is an error.
- Do not use commas in the format string of a **scanf** statement.
- Using an address operator & with a variable in the **printf** statement wi result in runtime error.

Case Studies

1. Inventory Report

Problem: The ABC Electric Company manufactures four consumer products. Their inventory position on a particular day is given below:

Code	Quantity	Rate (Rs)
F105	275	575.00
H220	107	99.95
I019	321	215.50
M315	89	725.00

It is required to prepare the inventory report table in the following format:

INVENTORY REPORT

Code	Quantity	Rate	Value
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
		Total Value:	—

The value of each item is given by the product of quantity and rate.

Program: The program given in Fig. 4.12 reads the data from the terminal and generates the required output. The program uses subscripted variables which are discussed in Chapter 7.

```
Program
    #define ITEMS 4
    main()
    { /* BEGIN */
      int i, quantity[5];
      float rate[5], value, total_value;
      char code[5][5];
      /* READING VALUES */
      i = 1;
      while ( i <= ITEMS)
      {
        printf("Enter code, quantity, and rate:");
        scanf("%s %d %f", code[i], &quantity[i],&rate[i]);
        i++;
      }
    /*.......Printing of Table and Column Headings.......*/
      printf("\n\n");
      printf("      INVENTORY REPORT      \n");
      printf("--------------------\n");
      printf("  Code Quantity Rate Value  \n");
      printf("--------------------\n");
    /*.......Preparation of Inventory Position..........*/
      total_value = 0;
      i = 1;
      while ( i <= ITEMS)
      {
        value = quantity[i] * rate[i];
        printf("%5s %10d %10.2f %e\n",code[i],quantity[i],
          rate[i],value);
        total_value += value;
        i++;
      }
    /*.......Printing of End of Table..................*/
     printf("----------------\n");
     printf("          Total Value = %e\n",total_value);
     printf("----------------\n");
    } /* END */
```

Output

```
Enter code, quantity, and rate:F105 275 575.00
Enter code, quantity, and rate:H220 107 99.95
Enter code, quantity, and rate:I019 321 215.50
Enter code, quantity, and rate:M315 89 725.00
          INVENTORY REPORT
-------------------------------------------
Code   Quantity   Rate       Value
F105      275    575.00   1.581250e+005
H220      107     99.95   1.069465e+004
I019      321    215.50   6.917550e+004
M315       89    725.00   6.452500e+004
-------------------------------------------
          Total Value = 3.025202e+005
-------------------------------------------
```

Fig. 4.12 *Program for inventory report*

2. Reliability Graph

Problem: The reliability of an electronic component is given by

reliability (r) = $e^{-\lambda t}$

where λ is the component failure rate per hour and t is the time of operation in hours. A graph is required to determine the reliability at various operating times, from 0 to 3000 hours. The failure rate λ (lambda) is 0.001.

Problem

```
    #include <math.h>
    #define LAMBDA 0.001
    main()
    {
      double t;
      float r;
      int i, R;
      for (i=1; i<=27; ++i)
      {
```

```
        printf("--");
      }
      printf("\n");
      for (t=0; t<=3000; t+=150)
      {
        r = exp(-LAMBDA*t);
        R = (int)(50*r+0.5);
        printf(" |");
        for (i=1; i<=R; ++i)
        {
          printf("*");
        }
        printf("#\n");
      }
      for (i=1; i<3; ++i)
      {
        printf(" |\n");
      }
    }
```

Output

```
 ------------------------------------------------------
 |**************************************************#
 |*******************************************#
 |*************************************#
 |********************************#
 |***************************#
 |************************#
 |********************#
 |*****************#
 |***************#
 |*************#
 |***********#
 |**********#
 |********#
 |*******#
 |******#
 |*****#
 |*****#
 |****#
 |***#
 |***#
 |**#
```

Fig. 4.13 *Program to draw reliability graph*

Program: The program given in Fig. 4.13 produces a shaded graph. The values of t are self-generated by the **for** statement

```
for (t=0; t <= 3000; t = t+150)
```

in steps of 150. The integer 50 in the statement

```
R = (int)(50*r+0.5)
```

is a scale factor which converts r to a large value where an integer is used for plotting the curve. Remember r is always less than 1.

Review Questions

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

(a) The purpose of the header file <studio.h> is to store the programs created by th users.

(b) The C standard function that receives a single character from the keyboard i **getchar.**

(c) The **getchar** cannot be used to read a line of text from the keyboard.

(d) The input list in a **scanf** statement can contain one or more variables.

(e) When an input stream contains more data items than the number of specifica tions in a **scanf** statement, the unused items will be used by the next **scanf** call i the program.

(f) Format specifiers for output convert internal representations for data to readab characters.

(g) Variables form a legal element of the format control string of a **printf** statemen

(h) The **scanf** function cannot be used to read a single character from the keyboar

(i) The format specification %+ –8d prints an integer left-justified in a field width o 8 with a plus sign, if the number is positive.

(j) If the field width of a format specifier is larger than the actual width of the valu the value is printed right-justified in the field.

(k) The print list in a **printf** statement can contain function calls.

(l) The format specification %5s will print only the first 5 characters of a given strin to be printed.

4.2 Fill in the blanks in the following statements.

(a) The ________specification is used to read or write a short integer.

(b) The conversion specifier ________is used to print integers in hexadecimal for

(c) For using character functions, we must include the header file _______ in th program.

(d) For reading a double type value, we must use the specification ________.

(e) The specification _______is used to read a data from input list and discard it wi out assigning it to many variable.

(f) The specification ____ may be used in **scanf** to terminate reading at the encou ter of a particular character.

(g) The specification %[ ] is used for reading strings that contain ________.

(h) By default, the real numbers are printed with a precision of ______decimal plac

(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);`

(e) printf("%s"; city);

(f) printf(%-10d %-15s", count, city);

4.9 State what (if anything) is wrong with each of the following output statements:

(a) printf(%d 7.2%f, year, amount);

(b) printf("%-s, %c"\n, city, code);

(c) printf("%f, %d, %s, price, count, city);

(d) printf("%c%d%f\n", amount, code, year);

4.10 In response to the input statement

scanf("%4d%*%d", &year, &code, &count);

the following data is keyed in:

19883745

What values does the computer assign to the variables **year**, **code**, and **count**?

4.11 How can we use the **getchar( )** function to read multicharacter strings?

4.12 How can we use the **putchar ( )** function to output multicharacter strings?

4.13 What is the purpose of **scanf( )** function?

4.14 Describe the purpose of commonly used conversion characters in a **scanf( )** function

4.15 What happens when an input data item contains

(a) more characters than the specified field width and

(b) fewer characters than the specified field width?

4.16 What is the purpose of **print( )** function?

4.17 Describe the purpose of commonly used conversion characters in a **printf( )** function

4.18 How does a control string in a **printf( )** function differ from the control string in a **scanf( )** function?

4.19 What happens if an output data item contains

(a) more characters than the specified field width and

(b) fewer characters than the specified field width?

4.20 How are the unrecognized characters within the control string are interpreted in

(a) **scanf** function; and

(b) **printf** function?

Programming Exercises

4.1 Given the string "WORDPROCESSING", write a program to read the string from th terminal and display the same in the following formats:

(a) WORD PROCESSING

(b) WORD
PROCESSING

(c) W.P.

4.2 Write a program to read the values of x and y and print the results of the followin expressions in one line:

(a) $\frac{x+y}{x-y}$ (b) $\frac{x+y}{2}$ (c) (x+y)(x–y)

4.3 Write a program to read the following numbers, round them off to the nearest integers and print out the results in integer form:

35.7 50.21 – 23.73 – 46.45

4.4 Write a program that reads 4 floating point values in the range, 0.0 to 20.0, and prints a horizontal bar chart to represent these values using the character * as the fill character. For the purpose of the chart, the values may be rounded off to the nearest integer. For example, the value 4.36 should be represented as follows.

```
*   *   *   *
*   *   *   *   4.36
*   *   *   *
```

Note that the actual values are shown at the end of each bar.

4.5 Write an interactive program to demonstrate the process of multiplication. The program should ask the user to enter two two-digit integers and print the product of integers as shown below.

```
                  45
              ×   37
              ------
7 × 45 is        315
3 × 45 is       135
              ------
Add   them      1665
              ------
```

4.6 Write a program to read three integers from the keyboard using one **scanf** statement and output them on one line using:

(a) three **printf** statements,

(b) only one **printf** with conversion specifiers, and

(c) only one **printf** without conversion specifiers.

4.7 Write a program that prints the value 10.45678 in exponential format with the following specifications:

(a) correct to two decimal places;

(b) correct to four decimal places; and

(c) correct to eight decimal places.

4.8 Write a program to print the value 345.6789 in fixed-point format with the following specifications:

(a) correct to two decimal places;

(b) correct to five decimal places; and

(c) correct to zero decimal places.

4.9 Write a program to read the name ANIL KUMAR GUPTA in three parts using the **scanf** statement and to display the same in the following format using the **printf** statement.

(a) ANIL K. GUPTA

(b) A.K. GUPTA

(c) GUPTA A.K.

4.10 Write a program to read and display the following table of data.

Name	Code	Price
Fan	67831	1234.50
Motor	450	5786.70

The name and code must be left-justified and price must be right-justified.

5

Decision Making and Branching

5.1 INTRODUCTION

We have seen that a C program is a set of statements which are normally execut sequentially in the order in which they appear. This happens when no options or n repetitions of certain calculations are necessary. However, in practice, we have a number situations where we may have to change the order of execution of statements based o certain conditions, or repeat a group of statements until certain specified conditions are me This involves a kind of decision making to see whether a particular condition has occurred o not and then direct the computer to execute certain statements accordingly.

C language possesses such decision-making capabilities by supporting the followin statements:

1. **if** statement
2. **switch** statement
3. Conditional operator statement
4. **goto** statement

These statements are popularly known as *decision-making statements*. Since these stat ments 'control' the flow of execution, they are also known as *control statements*.

We have already used some of these statements in the earlier examples. Here, we sha discuss their features, capabilities and applications in more detail.

5.2 DECISION MAKING WITH IF STATEMENT

The **if** statement is a powerful decision-making statement and is used to control the flow o execution of statements. It is basically a two-way decision statement and is used i conjunction with an expression. It takes the following form:

if (test expression)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or non-zero) or 'false' (zero), it

transfers the control to a particular statement. This point of program has two *paths* to follow, one for the *true* condition and the other for the *false* condition as shown in Fig. 5.1.

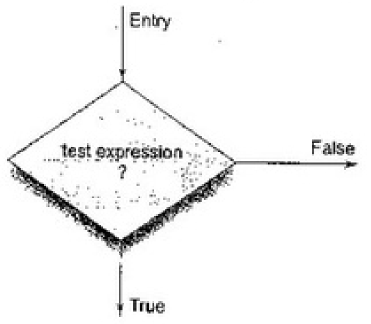


Fig. 5.1 *Two-way branching*

Some examples of decision making, using **if** statements are:

1. **if** (bank balance is zero)
   borrow money
2. **if** (room is dark)
   put on lights
3. **if** (code is 1)
   person is male
4. **if** (age is more than 55)
   person is retired

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

1. Simple **if** statement
2. **if.....else** statement
3. Nested **if....else** statement
4. **else if** ladder.

We shall discuss each one of them in the next few sections.

5.3 SIMPLE IF STATEMENT

The general form of a simple **if** statement is

```
if (test expression)
{
    statement-block;
}
statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x*. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence. This is illus-trated in Fig. 5.2.

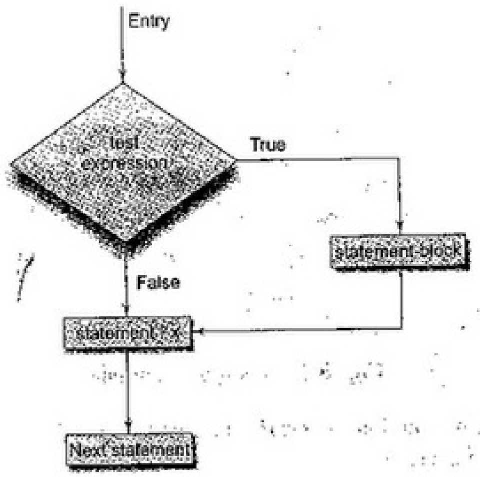


Fig. 5.2 *Flowchart of simple if control*

Consider the following segment of a program that is written for processing of marks ob-tained in an entrance examination.

```
.........
.........
if (category == SPORTS)
{
    marks = marks + bonus_marks;
}
printf("%f", marks);
.........
.........
```

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bonus_marks are added to his marks before they are printed. For others, bonus_marks are not added.

Example 5.1 The program in Fig. 5.3 reads four values a, b, c, and d from the termin and evaluates the ratio of (a+b) to (c–d) and prints the result, c–d is not equal to zero.

The program given in Fig. 5.3 has been run for two sets of data to see that the paths functio properly. The result of the first run is printed as,

Ratio = –3.181818

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

        printf("Enter four integer values\n");
        scanf("%d %d %d %d", &a, &b, &c, &d);

        if (c-d != 0) /* Execute statement block */
        {
            ratio = (float)(a+b)/(float)(c-d);
            printf("Ratio = %f\n", ratio);
        }
  }

Output

  Enter four integer values
  12 23 34 45
  Ratio = -3.181818

  Enter four integer values
  12  23  34  34
```

Fig. 5.3 *Illustration of simple if statement*

The second run has neither produced any results nor any message. During the second run, the value of (c–d) is equal to zero and therefore, the statements contained in the statement-block are skipped. Since no other statement follows the statement-block, program stops without producing any output.

Note the use of **float** conversion in the statement evaluating the **ratio**. This is necessary to avoid truncation due to integer division. Remember, the output of the first run –3.181818 is printed correct to six decimal places. The answer contains a round off error. If we wish to have higher accuracy, we must use **double** or **long double** data type.

The simple **if** is often used for counting purposes. The Example 5.2 illustrates this.

Example 5.2 The program in Fig. 5.4 counts the number of boys whose weight is less than 50 kg and height is greater than 170 cm.

The program has to test two conditions, one for weight and another for height. This is done using the compound relation

if (weight < 50 && height > 170)

[illegible] ve been equivalently done using two **if** statements as follows:

if (weight < 50)
  if (height > 170)
    count = count +1;

If the value of **weight** is less than 50, then the following statement is executed, which i
turn is another **if** statement. This **if** statement tests **height** and if the **height** is great
than 170, then the **count** is incremented by 1.

```
Program
main()
{
      int count, i;
      float weight, height;

      count = 0;
      printf("Enter weight and height for 10 boys\n");

      for (i =1; i <= 10; i++)
      {
          scanf("%f %f", &weight, &height);
          if (weight < 50 && height > 170)
              count = count + 1;
      }
      printf("Number of boys with weight < 50 kg\n");
      printf("and height > 170 cm = %d\n", count);
}
```

Output

```
Enter weight and height for 10 boys
45   176.5
55   174.2
47   168.0
49   170.7
54   169.0
53   170.5
49   167.0
48   175.0
47   167
51   170
Number of boys with weight < 50 kg
and height > 170 cm = 3
```

Fig. 5.4 *Use of if for counting*

Applying De Morgan's Rule

While designing decision statements, we often come across a situation where the logical NOT operator is applied to a compound logical expression, like !(x&&y||!z). However, a positive logic is always easy to read and comprehend than a negative logic. In such cases, we may apply what is known as **De Morgan's** rule to make the total expression positive. This rule is as follows:

"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators"

That is,

x becomes !x

!x becomes x

&& becomes ||

|| becomes &&

Examples:

!(x && y || !z) becomes !x || !y && z

!(x < =0 || !condition) becomes x >0&& condition

5.4 THE IF.....ELSE STATEMENT

The **if...else** statement is an extension of the simple **if** statement. The general form is

```
If (test expression)
    {
      True-block statement(s)
    }
else
    {
      False-block statement(s)
    }
statement-x
```

If the *test expression* is true, then the *true-block statement(s)*, immediately following the **if** statements are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. This is illustrated in Fig. 5.5. In both the cases, the control is transferred subsequently to the statement-x.

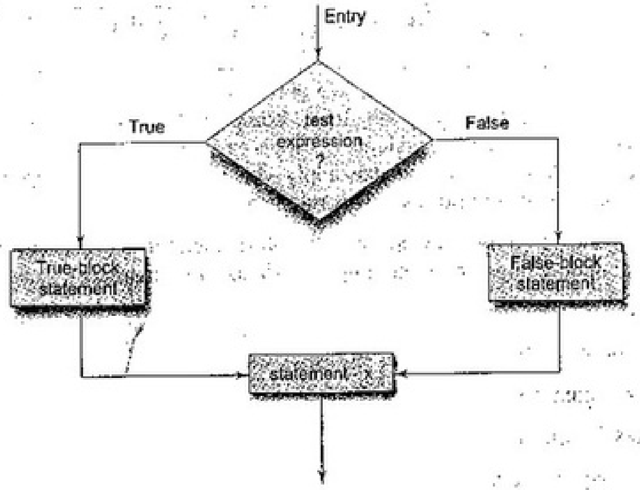


Fig. 5.5 *Flowchart of if......else control*

Let us consider an example of counting the number of boys and girls in a class. We u code 1 for a boy and 2 for a girl. The program statement to do this may be written as follow

```
..........
..........
if (code == 1)
  boy = boy + 1;
if (code == 2)
  girl = girl+1;
..........
..........
```

The first test determines whether or not the student is a boy. If yes, the number of boys increased by 1 and the program continues to the second test. The second test again det mines whether the student is a girl. This is unnecessary. Once a student is identified a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not bo The above program segment can be modified using the **else** clause as follows:

```
..........
..........
if (code == 1)
  boy = boy + 1;
else
  girl = girl + 1;
xxxxxxxxxx
..........
```

Here, if the code is equal to 1, the statement **boy = boy + 1**; is executed and the control is transferred to the statement **xxxxxx**, after skipping the else part. If the code is not equal to 1, the statement **boy = boy + 1**; is skipped and the statement in the **else** part **girl = girl + 1**; is executed before the control reaches the statement **xxxxxxxx**.

Consider the program given in Fig. 5.3. When the value (c–d) is zero, the ratio is not calculated and the program stops without any message. In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the **else** clause as follows:

```
..........
..........
if (c-d != 0)
  {
    ratio = (float)(a+b)/(float)(c-d);
    printf("Ratio = %f\n", ratio);
  }
else
  printf("c-d is zero\n");
..........
..........
```

Example 5.3 A program to evaluate the power series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!},\ 0 < x < 1$$

is given in Fig. 5.6. It uses **if......else** to test the accuracy.

The power series contains the recurrence relationship of the type

$$T_n = T_{n-1}\left(\frac{x}{n}\right) \text{ for } n > 1$$

$$T_1 = x \text{ for } n = 1$$

$$T_0 = 1$$

If T_{n-1} (usually known as *previous term*) is known, then T_n (known as *present term*) can be easily found by multiplying the previous term by x/n. Then

$$e^x = T_0 + T_1 + T_2 + \ldots\ldots + T_n = \text{sum}$$

```
Program
  #define ACCURACY 0.0001
  main()
  {
    int n, count;
    float x, term, sum;

    printf("Enter value of x:");
    scanf("%f", &x);
```

```
        n = term = sum = count = 1;
        while (n <= 100)
        {
           term = term * x/n;
           sum = sum + term;
           count = count + 1;
           if (term < ACCURACY)
              n = 999;
           else
              n = n + 1;
        }
        printf("Terms = %d Sum = %f\n", count, sum);
```

Output

```
        Enter value of x:0
        Terms = 2 Sum = 1.000000
        Enter value of x:0.1
        Terms = 5 Sum = 1.105171
        Enter value of x:0.5
        Terms = 7 Sum = 1.648720
        Enter value of x:0.75
        Terms = 8 Sum = 2.116997
        Enter value of x:0.99
        Terms = 9 Sum = 2.691232
        Enter value of x:1
        Terms = 9 Sum = 2.718279
```

Fig. 5.6 *Illustration of if...else statement*

The program uses **count** to count the number of terms added. The program stops wh the value of the term is less than 0.0001 (**ACCURACY**). Note that when a term is less th **ACCURACY**, the value of n is set equal to 999 (a number higher than 100) and therefore t **while** loop terminates. The results are printed outside the **while** loop.

5.5 NESTING OF IF....ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one **if...el** statement in *nested* form as shown below:
The logic of execution is illustrated in Fig. 5.7. If the *condition-1* is false, the statement-3 w be executed; otherwise it continues to perform the second test. If the *condition-2* is true, t

```
If    (test condition-1)
   {
   if (test condition-2);
      {
      statement -1;
      }
   else
      {
      statement -2;
      }
   }
else
   {
   statement -3;
   }
statement -x;
```

statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

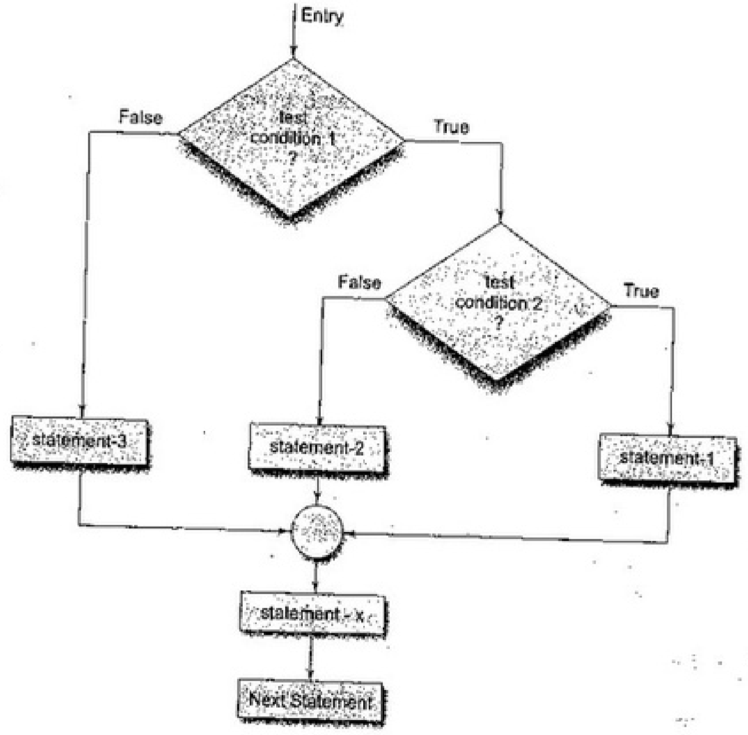


Fig. 5.7 *Flow chart of nested if...else statements*

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the balance held on 31st December is given to every one, irrespective of their balance, and 5 per cent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded as follows:

```
.........
  if (sex is female)
{
  if (balance > 5000)
     bonus = 0.05 * balance;
  else
     bonus = 0.02 * balance;
}
else
{
     bonus = 0.02 * balance;
}
balance = balance + bonus;
.........
.........
```

When nesting, care should be exercised to match every **if** with an **else**. Consider the following alternative to the above program (which looks right at the first sight):

```
if (sex is female)
  if (balance > 5000)
    bonus = 0.05 * balance;
  else
    bonus = 0.02 * balance;
  balance = balance + bonus;
```

There is an ambiguity as to over which **if** the **else** belongs to. In C, an else is linked to the closest non-terminated if. Therefore, the **else** is associated with the inner **if** and there is no else option for the outer **if**. This means that the computer is trying to execute the statement

```
balance = balance + bonus;
```

without really calculating the bonus for the male account holders.
Consider another alternative, which also looks correct:

```
if (sex is female)
  {
    if (balance > 5000)
    bonus = 0.05 * balance;
  }
  else
    bonus = 0.02 * balance;
  balance = balance + bonus;
```

In this case, **else** is associated with the outer **if** and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing **else** option for the inner **if**.

Example 5.4 The program in Fig. 5.8 selects and prints the largest of the three numbers using nested **if....else** statements.

```
Program
      main()
      {
      float A, B, C;
      printf("Enter three values\n");
      scanf("%f %f %f", &A, &B, &C);
      printf("\nLargest value is ");
      if (A>B)
      {
        if (A>C)
          printf("%f\n", A);
        else
          printf("%f\n", C);
      }
      else
      {
        if (C>B)
          printf("%f\n", C);
        else
          printf("%f\n", B);
      }
    }
Output
      Enter three values
      23445 67379 88843
      Largest value is 88843.000000
```

Fig 5.8 *Selecting the largest of three numbers*

Dangling Else Problem

One of the classic problems encountered when we start using nested **if....else** statements is the dangling else. This occurs when a matching **else** is not available for an **if**. The answer to this problem is very simple. Always match an **else** to the most recent unmatched if in the current block. In some cases, it is possible that the false condition is not required. In such situations, **else** statement may be omitted

"**else is always paired with the most recent unpaired if**"

5.6 THE ELSE IF LADDER

There is another way of putting **ifs** together when multipath decisions are involved. multipath decision is a chain of **ifs** in which the statement associated with each **else** is an if. It takes the following general form:

```
        if ( condition 1)
            statement-1;

        else if ( condition 2)
                statement-2;

          else if ( condition 3)
                  statement-3;

             else if ( condition n)
                     statement-n;
                  else
                     default-statement;
        statement-x;
```

This construct is known as the **else if** ladder. The conditions are evaluated from the (of the ladder), downwards. As soon as a true condition is found, the statement associ with it is executed and the control is transferred to the statement-x (skipping the rest of ladder). When all the n conditions become false, then the final **else** containing the *defa statement* will be executed. Fig. 5.9 shows the logic of execution of **else if** ladder stateme

Let us consider an example of grading the students in an academic institution. grading is done according to the following rules:

Average marks	**Grade**
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

This grading can be done using the **else if** ladder as follows:

```
        if (marks > 79)
            grade = "Honours";
          else if (marks > 59)
              grade = "First Division";
            else if (marks > 49)
                grade = "Second Division";
              else if (marks > 39)
                     grade = "Third Division";
                else
```

```
                     grade = "Fail";
              printf ("%s\n", grade);
```

Consider another example given below:

```
        ----
        ----
        if (code == 1)
          colour = "RED";
        else if (code == 2)
            colour = "GREEN";
          else if (code == 3)
                colour = "WHITE";
              else
                colour = "YELLOW";
        ---
        ---
```

Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested **if...else** statements.

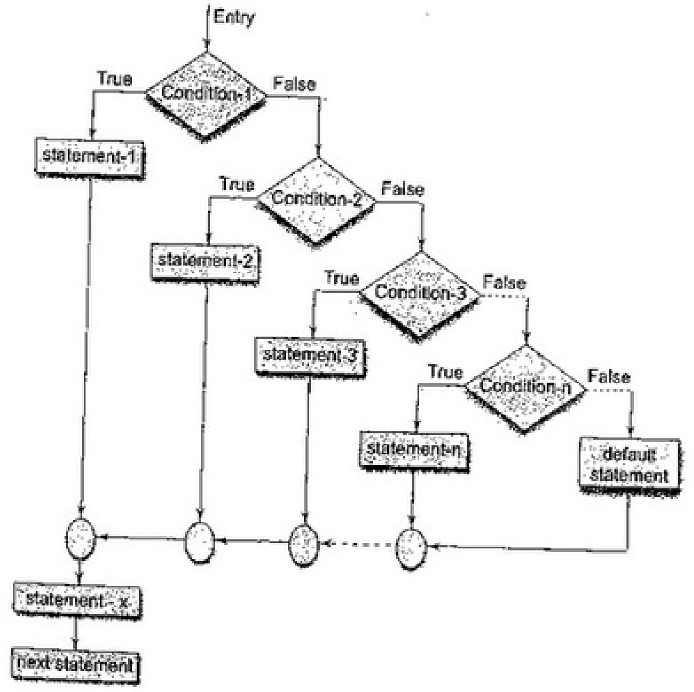


Fig. 5.9 *Flow chart of* ***else..if*** *ladder*

```
if (code != 1)
  if (code != 2)
    if (code != 3)
      colour = "YELLOW";
    else
      colour = "WHITE";
  else
      colour = "GREEN";
else
      colour = "RED";
```

In such situations, the choice is left to the programmer. However, in order to choose an if structure that is both effective and efficient, it is important that the programmer is fully aware of the various forms of an **if** statement and the rules governing their nesting.

Example 5.5 An electric power distribution company charges its domestic consume as follows:

Consumption Units	*Rate of Charge*
0 - 200	Rs. 0.50 per unit
201 - 400	Rs. 100 plus Rs. 0.65 per unit excess of 200
401 - 600	Rs. 230 plus Rs. 0.80 per unit excess of 400
601 and above	Rs. 390 plus Rs. 1.00 per unit excess of 600

The program in Fig. 5.10 reads the customer number and power consumed and prin the amount to be paid by the customer.

```
Program
  main()
  {
    int units, custnum;
    float charges;
    printf("Enter CUSTOMER NO. and UNITS consumed\n");
    scanf("%d %d", &custnum, &units);
    if (units <= 200)
      charges = 0.5 * units;
    else if  (units <= 400)
            charges = 100 + 0.65 * (units - 200);
              else if (units <= 600)
              charges = 230 + 0.8 * (units - 400);
                else
                charges = 390 + (units - 600);
    printf("\n\nCustomer No: %d: Charges = %.2f\n",
      custnum, charges);
  }
Output
  Enter CUSTOMER NO. and UNITS consumed 101 150
```

```
Customer No:101 Charges = 75.00
Enter CUSTOMER NO. and UNITS consumed 202 225
Customer No:202 Charges = 116.25
Enter CUSTOMER NO. and UNITS consumed 303 375
Customer No:303 Charges = 213.75
Enter CUSTOMER NO. and UNITS consumed 404 520
Customer No:404 Charges = 326.00
Enter CUSTOMER NO. and UNITS consumed 505 625
Customer No:505 Charges = 415.00
```

Fig. 5.10 *Illustration of else..if ladder*

Rules for Indentation

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use *indentation* to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

5.7 THE SWITCH STATEMENT

We have seen that when one of the many alternatives is to be selected, we can use an if statement to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a **switch**. The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown below:

```
switch (expression)
{
  case value-1:
             block-1
             break;
  case value-2:
             block-2
             break;
  ......
  ......
  default:
             default-block
             break;
}
statement-x;
```

The *expression* is an integer expression or characters. *Value-1, value-2* ..... are constants o constant expressions (evaluable to an integral constant) and are known as *case labels*. Eac of these values should be unique within a **switch** statement. **block-1, block-2** .... a statement lists and may contain zero or more statements. There is no need to put brac around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successfully compared again the values *value-1, value-2*,.... If a case is found whose value matches with the value of t expression, then the block of statements that follows the case are executed.

The **break** statement at the end of each block signals the end of a particular case a causes an exit from the **switch** statement, transferring the control to the **statement** following the **switch.**

The **default** is an optional case. When present, it will be executed if the value of t *expression* does not match with any of the case values. If not present, no action takes plac all matches fail and the control goes to the **statement-x**. (ANSI C permits the use of as ma as 257 case labels).

The selection process of **switch** statement is illustrated in the flow chart shown Fig. 5.11.

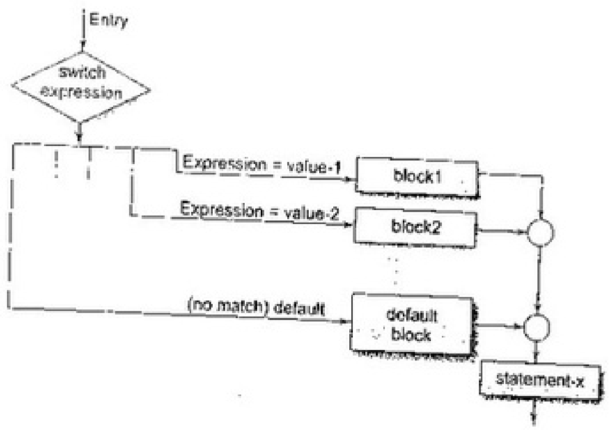


Fig. 5.11 *Selection process of the* ***switch*** *statement*

The **switch** statement can be used to grade the students as discussed in the last section. This is illustrated below:

```
---
---
index = marks/10
switch (index)
{
  case 10:
  case 9:
  case 8:
       grade = "Honours";
       break;
  case 7:
  case 6:
       grade = "First Division";
       break;
  case 5:
       grade = "Second Division";
       break;
  case 4:
       grade = "Third Division";
       break;
  default:
       grade = "Fail";
       break;
}
printf("%s\n", grade);
---
---
```

Note that we have used a conversion statement

```
index = marks / 10;
```

where, index is defined as an integer. The variable index takes the following integer value

Marks	Index
100	10
90 - 99	9
80 - 89	8
70 - 79	7
60 - 69	6
50 - 59	5
40 - 49	4
.	.
.	.
0	0

This segment of the program illustrates two important features. First, it uses empty ca
The first three cases will execute the same statements

```
grade = "Honours";
break;
```

Same is the case with case 7 and case 6. Second, default condition is used for all other ca
where marks is less than 40.

The **switch** statement is often used for menu selection. For example:

```
          ----
          ----
          printf(" TRAVEL GUIDE\n\n");
          printf(" A Air Timings\n" );
          printf(" T Train Timings\n");
          printf(" B Bus Service\n" );
          printf(" X To skip\n" );
          printf("\n Enter your choice\n");
          character = getchar();
          switch (character)
          {
            case 'A' :
                       air-display();
                       break;
            case 'B' :
                       bus-display();
                       break;
            case 'T' :
                       train-display();
                       break;
          default :
                       printf(" No choice\n");
          }
            ----
            ----
```

It is possible to nest the **switch** statements. That is, a **switch** may be part of a **case** statement. ANSI C permits 15 levels of nesting.

Rules for switch statement

- The **switch** expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.
- The **break** statement transfers the control out of the **switch** statement.
- The **break** statement is optional. That is, two or more case labels may belong to the same statements.
- The **default** label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one **default** label.
- The **default** may be placed anywhere but usually placed at the end.
- It is permitted to nest **switch** statements.

5.8 THE ? : OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use of the conditional operator is as follows:

conditional expression ? expression1 : expression2

The *conditional expression* is evaluated first. If the result is nonzero, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
  flag = 0;
else
  flag = 1;
```

can be written as

```
flag = ( x < 0 ) ? 0 : 1;
```

Consider the evaluation of the following function:

$y = 1.5x + 3$ for $x \le 2$

$y = 2x + 5$ for $x > 2$

This can be evaluated using the conditional operator as follows:

```
y = ( x > 2 ) ? (2 * x + 5) : (1.5 * x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

```
salary = (x != 40) ? ((x < 40) ? (4*x+100) : (4.5*x+150)) : 300;
```

The same can be evaluated using **if...else** statements as follows:

```
if (x <= 40)
    if (x < 40)
        salary = 4 * x+100;
    else
        salary = 300;
else
        salary = 4.5 * x+150;
```

When the conditional operator is used, the code becomes more concise and perhaps, mo efficient. However, the readability is poor. It is better to use **if** statements when more tha single nesting of conditional operator is required.

Example 5.6 An employee can apply for a loan at the beginning of every six mo but he will be sanctioned the amount according to the following c pany rules:

Rule 1 : An employee cannot enjoy more than two loans at any poi time.

Rule 2 : Maximum permissible total loan is limited and depends upon category of the employee.

A program to process loan applications and to sanction loans is give Fig. 5.12.

```
Program
    #define MAXLOAN 50000
    main()
    {
        long int loan1, loan2, loan3, sancloan, sum23;
        printf("Enter the values of previous two loans:\n");
        scanf(" %ld %ld", &loan1, &loan2);
        printf("\nEnter the value of new loan:\n");
        scanf(" %ld", &loan3);
        sum23 = loan2 + loan3;
        sancloan = (loan1>0)? 0 : ((sum23>MAXLOAN)?
```

```
                    MAXLOAN - loan2 : loan3);
        printf("\n\n");
        printf("Previous loans pending:\n%ld %ld\n",loan1,loan2);
        printf("Loan requested = %ld\n", loan3);
        printf("Loan sanctioned = %ld\n", sancloan);
    }
Output
        Enter the values of previous two loans:
        0 20000
        Enter the value of new loan:
        45000
        Previous loans pending:
        0 20000
        Loan requested = 45000
        Loan sanctioned = 30000
        Enter the values of previous two loans:
        1000 15000
        Enter the value of new loan:
        25000
        Previous loans pending:
        1000 15000
        Loan requested  = 25000
        Loan sanctioned = 0
```

Fig. 5.12 *Illustration of the conditional operator*

The program uses the following variables:

loan3	-	present loan amount requested
loan2	-	previous loan amount pending
loan1	-	previous to previous loan pending
sum23	-	sum of loan2 and loan3
sancloan	-	loan sanctioned

The rules for sanctioning new loan are:

1. loan1 should be zero.
2. loan2 + loan3 should not be more than MAXLOAN.

Note the use of **long int** type to declare variables.

Some Guidelines for Writing Multiway Selection Statements

Complex multiway selection statements require special attention. The readers should be able to understand the logic easily. Given below are some guidelines that would help improve readability and facilitate maintenance.

- Avoid compound negative statements. Use positive statements wherever possible.

- Keep logical expressions simple. We can achieve this using nested if state ments, if necessary (KISS - Keep It Simple and Short).
- Try to code the normal/anticipated condition first.

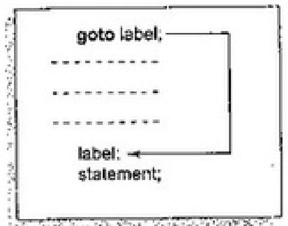
...st probable condition first. This will eliminate unnecessary tests thus improving the efficiency of the program.

- The choice between the nested if and switch statements is a matter of individual's preference. A good rule of thumb is to use the switch when alternative paths are three to ten.
- Use proper indentations (See Rules for Indentation).
- Have the habit of using default clause in switch statements.
- Group the case labels that have similar actions.

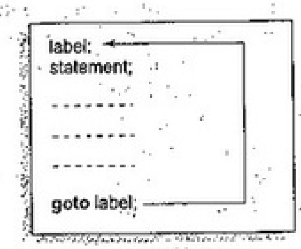
5.9 THE GOTO STATEMENT

So far we have discussed ways of controlling the flow of execution based on certain specifi conditions. Like many other languages, C supports the **goto** statement to branc unconditionally from one point to another in the program. Although it may not be essenti to use the **goto** statement in a highly structured language like C, there may be occasio when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is place immediately before the statement where the control is to be transferred. The general form of **goto** and *label* statements are shown below:



Forward jump



Backward jump

The *label*: can be anywhere in the program either before or after the **goto** label; statemen During running of a program when a statement like

goto begin;

is met, the flow of control will jump to the statement immediately following the label begin This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the *label*: before the statement **goto** *label*; a *loop* will be formed and some statements will be execute repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *label*: is placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward jump*.

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
     double x, y;
     read:
     scanf("%f", &x);
     if (x < 0) goto read;
     y = sqrt(x);
     printf("%f %f\n", x, y);
     goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided. Example 5.7 illustrates how such infinite loops can be eliminated.

Example 5.7 Program presented in Fig. 5.13 illustrates the use of the **goto** statement. The program evaluates the square root for five numbers. The variable count keeps the count of numbers read. When count is less than or equal to 5, **goto read;** directs the control to the label **read**; otherwise, the program prints a message and stops.

```
Program
       #include <math.h>
       main()
       {
            double x, y;
            int count;
            count = 1;
            printf("Enter FIVE real values in a LINE \n");
       read:
            scanf("%lf", &x);
            printf("\n");
            if (x < 0)
              printf("Value - %d is negative\n",count);
            else
            {
               y = sqrt(x);
               printf("%lf\t %lf\n", x, y);
            }
            count = count + 1;
            if (count <= 5)
        goto read;
            printf("\nEnd of computation");
        }
```

Output

```
        Enter FIVE real values in a LINE
        50.70 40 -36 75 11.25
        50.750000      7.123903
        40.000000      6.324555
        Value -3 is negative
        75.000000      8.660254
        11.250000      3.354102
        End of computation
```

Fig. 5.13 Use of the goto statement

Another use of the **goto** statement is to transfer the control out of a loop (or nested loop when certain peculiar conditions are encountered. Example:

```
        ----
        ----
        while (----)
        {
          for (----)
          {
          ----
          ----
          if (----)goto end_of_program;──┐
          ----                           │
          }                              │    Jumping
        ----                             │    out of
        ----                             │    loops
        }                                │
        end_of_program:   <──────────────┘
```

We should try to avoid using **goto** as far as possible. But there is nothing wrong, if we use to enhance the readability of the program or to improve the execution speed.

Just Remember

- Be aware of dangling **else** statements.
- Be aware of any side effects in the control expression such as if(x++).
- Use braces to encapsulate the statements in **if** and **else** clauses of an if.... else statement.
- Check the use of =operator in place of the equal operator = =.
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=.
- Writing !=, >= and <= operators like =!, => and =< is an error.
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression.
- It is an error to place a semicolon after the if expression.
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement.
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.
- Avoid using operands that have side effects in a logical binary expression such as (x--&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

Case Studies

1. Range of Numbers

Problem: A survey of the computer market shows that personal computers are sold at varying costs by the vendors. The following is the list of costs (in hundreds) quoted by some vendors:

35.00,	40.50,	25.00,	31.25,	68.15,
47.00,	26.65,	29.00	53.45,	62.50

Determine the average cost and the range of values.

Problem analysis: Range is one of the measures of dispersion used in statistical analysis of a series of values. The range of any series is the difference between the highest and the lowest values in the series. That is

Range = highest valuc – lowest value

It is therefore necessary to find the highest and the lowest values in the series.

Program: A program to determine the range of values and the average cost of a persona computer in the market is given in Fig. 5.14.

```
Program
  main()
  {
     int count;
     float value, high, low, sum, average, range;
     sum = 0;
     count = 0;
     printf("Enter numbers in a line :
     input a NEGATIVE number to end\n");
input:
     scanf("%f", &value);
     if (value < 0) goto output;
     count = count + 1;
     if (count == 1)
        high = low = value;
     else if (value > high)
          high = value;
        else if (value < low)
             low = value;
     sum = sum + value;
     goto input;
Output:
     average = sum/count;
     range = high - low;
     printf("\n\n");
     printf("Total values : %d\n", count);
     printf("Highest-value: %f\nLowest-value : %f\n",
          high, low);
     printf("Range        : %f\nAverage : %f\n",
          range, average);
  }
Output
  Enter numbers in a line : input a NEGATIVE number to end
  35 40.50 25 31.25 68.15 47 26.65 29 53.45 62.50 -1

  Total values  : 10
  Highest-value : 68.150002
  Lowest-value  : 25.000000
  Range  : 43.150002
  Average  : 41.849998
```

Fig. 5.14 *Calculation of range of values*

When the value is read the first time, it is assigned to two buckets, **high** and **low**, through the statement

```
high = low = value;
```

For subsequent values, the value read is compared with high; if it is larger, the value is assigned to high. Otherwise, the value is compared with low; if it is smaller, the value is assigned to low. Note that at a given point, the buckets high and low hold the highest and the lowest values read so far.

The values are read in an input loop created by the **goto** input; statement. The control is transferred out of the loop by inputting a negative number. This is caused by the statement

```
if (value < 0) goto output;
```

Note that this program can be written without using **goto** statements. Try.

2. Pay-Bill Calculations

Problem: A manufacturing company has classified its executives into four levels for the benefit of certain perks. The levels and corresponding perks are shown below:

Level	*Perks*	
	Conveyance allowance	*Entertainment allowance*
1	1000	500
2	750	200
3	500	100
4	250	–

An executive's gross salary includes basic pay, house rent allowance at 25% of basic pay and other perks. Income tax is withheld from the salary on a percentage basis as follows:

Gross salary	*Tax rate*
Gross <= 2000	No tax deduction
2000 < Gross <= 4000	3%
4000 < Gross <= 5000	5%
Gross > 5000	8%

Write a program that will read an executive's job number, level number, and basic pay and then compute the net salary after withholding income tax.

Problem analysis:

Gross salary = basic pay + house rent allowance + perks

Net salary = Gross salary – income tax.

The computation of perks depends on the level, while the income tax depends on the gross salary. The major steps are:

1. Read data.
2. Decide level number and calculate perks.
3. Calculate gross salary.
4. Calculate income tax.

5. Compute net salary.
6. Print the results.

Program: A program and the results of the test data are given in Fig. 5.15. Note that the last statement should be an executable statement. That is, the label **stop:** cannot be the last

Program

```
#define CA1 1000
#define CA2 750
#define CA3 500
#define CA4 250
#define EA1 500
#define EA2 200
#define EA3 100
#define EA4 0
main()
{
  int level, jobnumber;
  float  gross,
         basic,
         house_rent,
         perks,
         net,
         incometax;
  input:
  printf("\nEnter level, job number, and basic pay\n");
  printf("Enter 0 (zero) for level to END\n\n");
  scanf("%d", &level);
  if (level == 0) goto stop;
  scanf("%d %f", &jobnumber, &basic);
  switch (level)
  {
    case 1:
          perks = CA1 + EA1;
          break;
    case 2:
          perks = CA2 + EA2;
          break;
    case 3:
          perks = CA3 + EA3;
          break;
    case 4:
          perks = CA4 + EA4;
          break;
    default:
          printf("Error in level code\n");
```

```
          goto stop;
  }
  house_rent = 0.25 * basic;
  gross = basic + house_rent + perks;
  if (gross <= 2000)
    incometax = 0;
  else if (gross <= 4000)
          incometax = 0.03 * gross;
        else if (gross <= 5000)
            incometax = 0.05 * gross;
          else
            incometax = 0.08 * gross;
  net = gross - incometax;
  printf("%d %d %.2f\n", level, jobnumber, net);
  goto input;
  stop: printf("\n\nEND OF THE PROGRAM");
}
```

Output

```
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
1 1111 4000
1 1111 5980.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
2 2222 3000
2 2222 4465.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
3 3333 2000
3 3333 3007.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
4 4444 1000
4 4444 1500.00
Enter level, job number, and basic pay
Enter 0 (zero) for level to END
0
END OF THE PROGRAM
```

Fig. 5.15 *Pay-bill calculations*

Review Questions

5.1 State whether the following are *true* or *false*:
  (a) When **if** statements are nested, the last **else** gets associated with the nearest if without an **else**.
  (b) One **if** can have more than one **else** clause.
  (c) A **switch** statement can always be replaced by a series of **if..else** statements.
  (d) A **switch** expression can be of any type.
  (e) A program stops its execution when a **break** statement is encountered.
  (f) Each expression in the **else if** must test the same variable.
  (g) Any expression can be used for the **if** expression.
  (h) Each case label can have only one statement.
  (i) The **default** case is required in the **switch** statement.
  (j) The predicate !((x >= 10) ¦(y = = 5)) is equivalent to (x < 10) && (y !=5).

5.2 Fill in the blanks in the following statements.
  (a) The ______ operator is true only when both the operands are true.
  (b) Multiway selection can be accomplished using an **else if** statement or the ________ statement.
  (c) The ______ statement when executed in a **switch** statement causes immediate exit from the structure.
  (d) The ternary conditional expression using the operator ?: could be easily coded using ______statement.
  (e) The expression ! (x ! = y) can be replaced by the expression ________.

5.3 Find errors, if any, in each of the following segments:
  (a)
```
if (x + y = z && y > 0)
    printf(" ");
```
  (b)
```
if (code > 1);
    a = b + c
else
    a = 0
```
  (c)
```
if (p < 0) || (q < 0)
    printf (" sign is negative");
```

5.4 The following is a segment of a program:
```
x = 1;
y = 1;
if (n > 0)
    x = x + 1;
    y = y - 1;
printf(" %d %d", x, y);
```
What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

5.5 Rewrite each of the following without using compound relations:
  (a)
```
if (grade <= 59 && grade >= 50)
    second = second + 1;
```

  (b)
```
if (number > 100 || number < 0)
    printf(" Out of range");
else
    sum = sum + number;
```
  (c)
```
if ((M1 > 60 && M2 > 60) || T > 200)
    printf(" Admitted\n");
else
    printf(" Not admitted\n");
```

5.6 Assuming x = 10, state whether the following logical expressions are true or false.

(a) x = = 10 && x > 10 && !x	(b) x = = 10 || x > 10 && ! x
(c) x = = 10 && x > 10 || ! x	(d) x = = 10 || x > 10 || !x

5.7 Find errors, if any, in the following switch related statements. Assume that the variables x and y are of int type and x = 1 and y = 2
  (a) `switch (y);`
  (b) `case 10;`
  (c) `switch (x + y)`
  (d) `switch (x) {case 2: y = x + y; break};`

5.8 Simplify the following compound logical expressions

(a) !(x <=10)	(b) !(x = = 10) || ! ((y = = 5) || (z < 0))
(c) ! ((x +y = = z) && !(z > 5)	(d) !((x <=5) && (y = = 10) & & (z < 5))

5.9 Assuming that x = 5, y = 0, and z = 1 initially, what will be their values after executing the following code segments?
  (a)
```
if (x && y)
    x = 10;
else
    y = 10;
```
  (b)
```
if (x || y || z)
    y = 10;
else
    z = 0;
```
  (c)
```
if (x)
  if (y)
    z = 10;
else
    z = 0;
```
  (d)
```
if (x = = 0 || x & & y)
  if (!y)
    z = 0;
else
    y = 1;
```

5.10 Assuming that x = 2, y = 1 and z = 0 initially, what will be their values after executing the following code segments?
  (a) switch (x)

```
    {
      case 2:
          x = 1;
          y = x + 1;
      case 1:
          x = 0;
          break;
      default:
          x = 1;
          y = 0;
    }
(b) switch (y)
    {
      case 0:
          x = 0;
          y = 0;
      case 2:
          x = 2;
          z = 2;
      default:
          x = 1;
          y = 2;
    }
```

5.11 Find the error, if any, in the following statements:

```
(a) if ( x > = 10 ) then
    printf ( "\n") ;
(b) if x > = 10
    printf ( "OK" ) ;
(c) if (x = 10)
    printf ("Good" ) ;
(d) if (x = < 10)
     printf ("Welcome") ;
```

5.12 What is the output of the following program?

```
main (  )
{
        int m = 5 ;
        if (m < 3) printf("%d" , m+1) ;
        else if(m < 5) printf("%d", m+2);
        else if(m < 7) printf("%d", m+3);
        else printf("%d", m+4);

}
```

5.13 What is the output of the following program?

```
main ( )
{
      int m = 1;
      if ( m==1)
      {
            printf ( " Delhi " ) ;
            if (m == 2)
            printf( "Chennai" ) ;
            else
            printf("Bangalore") ;
      }
      else;
      printf(" END");
}
```

5.14 What is the output of the following program?

```
main( )
{
      int m ;
      for (m = 1; m<5; m++)
            printf(%d\n", (m%2) ? m : m*2);
}
```

5.15 What is the output of the following program?

```
main( )
{
      int m, n, p ;
      for ( m = 0; m < 3; m++ )
      for (n = 0; n<3; n++ )
      for ( p = 0; p < 3;; p++ )
      if ( m + n + p == 2 )
      goto print;

      print :
      printf("%d, %d, %d", m, n, p);
}
```

5.16 What will be the value of x when the following segment is executed?

```
      int x = 10, y = 15;
      x = (x<y)? (y+x) : (y-x) ;
```

5.17 What will be the output when the following segment is executed?

```
int x = 0;
if (x >= 0)
if ( x > 0 )
```

```
printf("Number is positive");
else
printf("Number is negative");
```

5.18 What will be the output when the following segment is executed?

```
char ch = 'a' ;
switch (ch)
{
        case 'a' :
        printf( "A" ) ;
        case'b':
        Printf ("B") ;
        default :
        printf(" C ") ;
}
```

5.19 What will be the output of the following segment when executed?

```
int x = 10, y = 20;
if( (x<y) || (x+5) > 10 )
printf("%d", x);
else
printf("%d", y);
```

5.20 What will be output of the following segment when executed?

```
int a = 10, b = 5;
if (a > b)
{
        if(b > 5)
        printf("%d", b);
}
else
        printf("%d", a);
```

Programming Exercises

5.1 Write a program to determine whether a given number is 'odd' or 'even' and print the message
NUMBER IS EVEN
or
NUMBER IS ODD
(a) without using **else** option, and (b) with **else** option.

5.2 Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.

5.3 A set of two linear equations with two unknowns x1 and x2 is given below:

$$ax_1 + bx_2 = m$$
$$cx_1 + dx_2 = n$$

The set has a unique solution

$$x1 = \frac{md - bn}{ad - cb}$$

$$x2 = \frac{na - mc}{ad - cb}$$

provided the denominator ad – cb is not equal to zero.
Write a program that will read the values of constants a, b, c, d, m, and n and compute the values of x_1 and x_2. An appropriate message should be printed if ad – cb = 0.

5.4 Given a list of marks ranging from 0 to 100, write a program to compute and print the number of students:
(a) who have obtained more than 80 marks,
(b) who have obtained more than 60 marks,
(c) who have obtained more than 40 marks,
(d) who have obtained 40 or less marks,
(e) in the range 81 to 100,
(f) in the range 61 to 80,
(g) in the range 41 to 60, and
(h) in the range 0 to 40.
The program should use a minimum number of **if** statements.

5.5 Admission to a professional course is subject to the following conditions:
(a) Marks in Mathematics >= 60
(b) Marks in Physics >= 50
(c) Marks in Chemistry >= 40
(d) Total in all three subjects >= 200
or
Total in Mathematics and Physics >= 150
Given the marks in the three subjects, write a program to process the applications to list the eligible candidates.

5.6 Write a program to print a two-dimensional Square Root Table as shown below, to provide the square root of any number from 0 to 9.9. For example, the value x will give the square root of 3.2 and y the square root of 3.9.

Square Root Table

Number	0.0	0.1	0.2	....	0.9
0.0 1.0 2.0					
3.0			x		y
9.0					

5.7 Shown below is a Floyd's triangle.

```
1
2 3
4 5 6
7 8 9 10
11 .. .. .. 15
.
.
79 .. .. .. .. .. 91
```

(a) **Write a program to print this triangle.**

(b) **Modify the program to produce the following form of Floyd's triangle.**

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

5.8 **A cloth showroom has announced the following seasonal discounts on purchase of items:**

Purchase amount	*Discount*	
	Mill cloth	*Handloom items*
0 – 100	–	5%
101 – 200	5%	7.5%
201 – 300	7.5%	10.0%
Above 300	10.0%	15.0%

Write a program using **switch** and **if** statements to compute the net amount to be paid by a customer.

5.9 Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 & \text{for } x < 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

using

(a) nested **if** statements,

(b) **else if** statements, and

(c) conditional operator ? :

5.10 Write a program to compute the real roots of a quadratic equation

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

The roots are given by the equations

$$x_1 = -b + \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = b - \frac{\sqrt{b^2 - 4ac}}{2a}$$

The program should request for the values of the constants a, b and c and print the values of x_1 and x_2. Use the following rules:

(a) No solution, if both a and b are zero

(b) There is only one root, if a = 0 (x = –c/b)

(c) There are no real roots, if $b^2 - 4ac$ is negative

(d) Otherwise, there are two real roots

Test your program with appropriate data so that all logical paths are working as per your design. Incorporate appropriate output messages.

5.11 Write a program to read three integer values from the keyboard and displays the output stating that they are the sides of right-angled triangle.

5.12 An electricity board charges the following rates for the use of electricity:

For the first 200 units: 80 P per unit

For the next 100 units: 90 P per unit

Beyond 300 units: Rs 1.00 per unit

All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs. 400, then an additional surcharge of 15% of total amount is charged.

Write a program to read the names of users and number of units consumed and print out the charges with names.

5.13 Write a program to compute and display the sum of all integers that are divisible by 6 but not divisible by 4 and lie between 0 and 100. The program should also count and display the number of such values.

5.14 Write an interactive program that could read a positive integer number and decide whether the number is a prime number and display the output accordingly.

Modify the program to count all the prime numbers that lie between 100 and 200.

NOTE: A prime number is a positive integer that is divisible only by 1 or by itself.

5.15 Write a program to read a double-type value x that represents angle in radians and a character-type variable T that represents the type of trigonometric function and display the value of

(a) sin(x), if s or S is assigned to T,

(b) cos (x), if c or C is assigned to T, and

(c) tan (x), if t or T is assigned to T

using (i) **if......else** statement and (ii) **switch** statement.