2.5 CONSTANTS

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

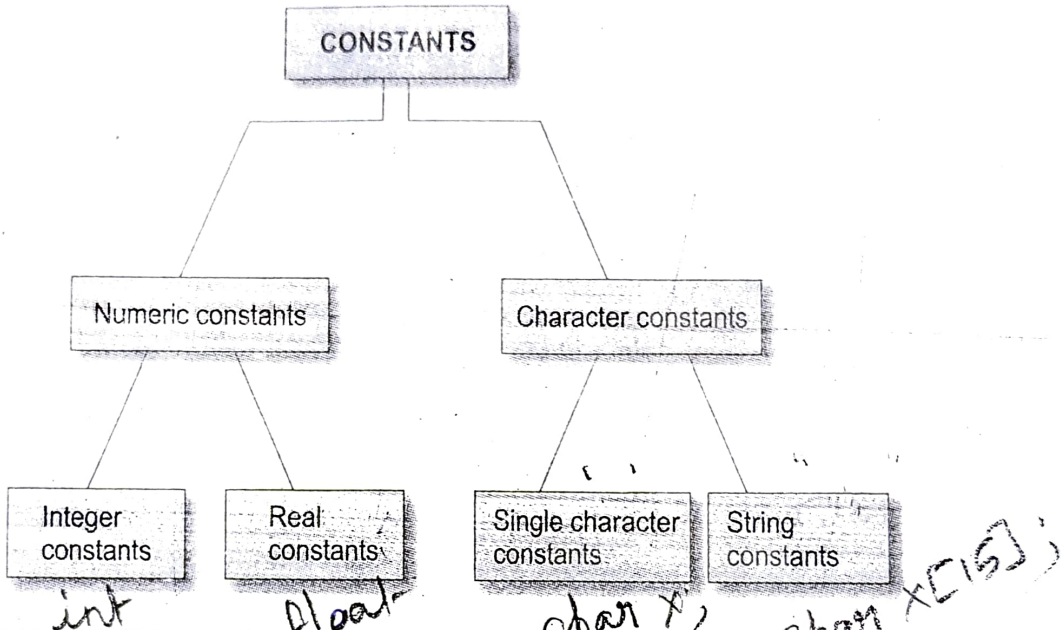


Fig. 2.2 *Basic types of C constants*

Integer Constants

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

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

123 –321 0 654321 +78

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

15 750 20,000 $1000

are illegal numbers.

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

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

037 0 0435 0551

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

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

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

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

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

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

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

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

        32767 -32768 -32759

        Long integer values

        32767 32768 32777
```

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

Real Constants

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

0.0083 –0.75 435.36 +247.0

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

215. .95 –.71 +.5

are all valid real numbers.

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

mantissa e exponent

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

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

Embedded white space is not allowed.

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

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

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

Table 2.4 *Examples of Numeric Constants*

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

Single Character Constants

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

'5' 'X' ';' ' '

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

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

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

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

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

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

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

String Constants

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

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

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

Backslash Character Constants

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

Table 2.5 *Backslash Character Constants*

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

2.6 VARIABLES

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

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

Average
height
Total
Counter_1
class_strength

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

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

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

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

Table 2.6 *Examples of Variable Names*

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

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

average_height
average_weight

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

avg_height and avg_weight

or

ht_average and wt_average

without changing their meanings.

2.7 DATA TYPES

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

ANSI C supports three classes of data types:

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

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

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

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

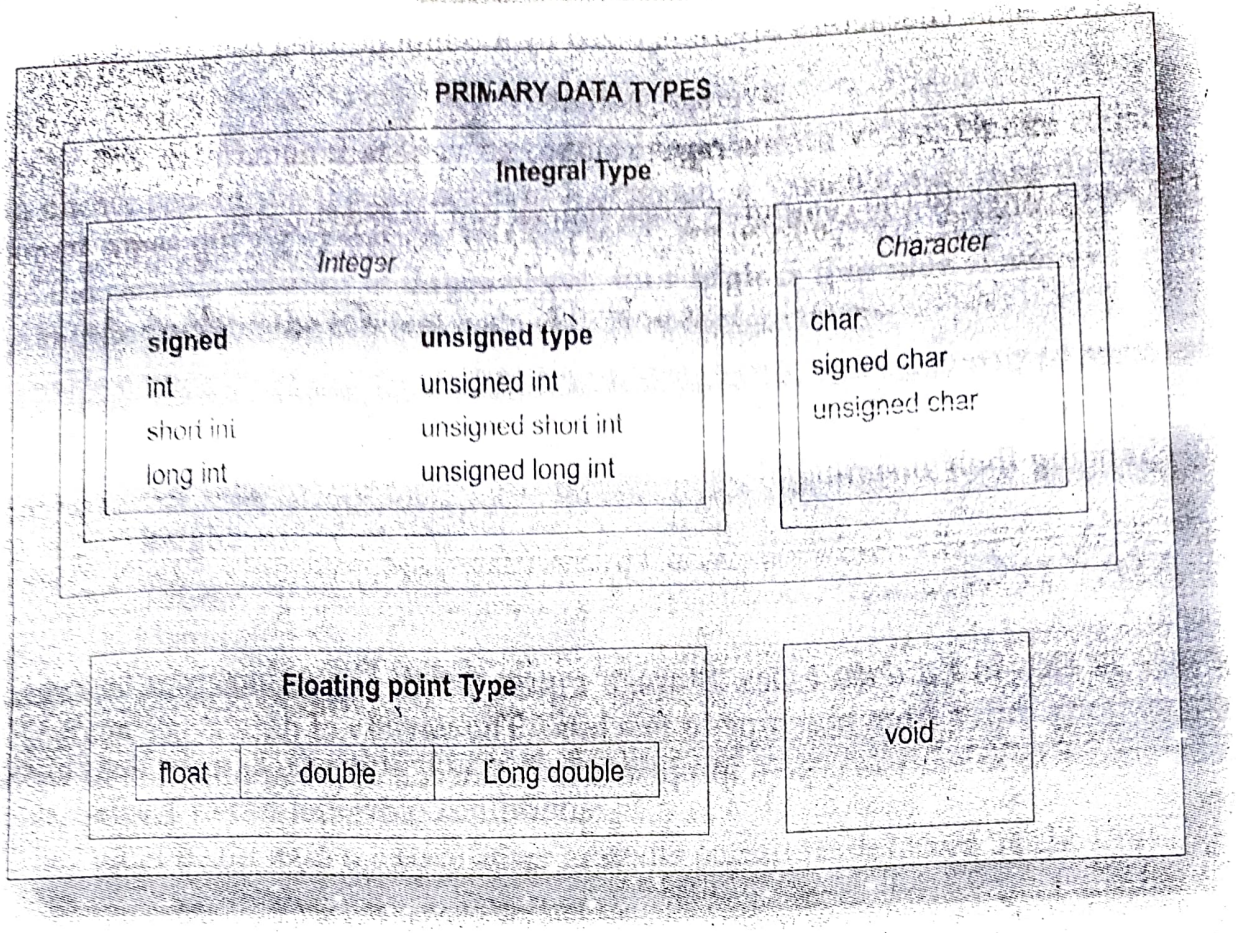


Fig. 2.4 *Primary data types in C*

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

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

Integer Types

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

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

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

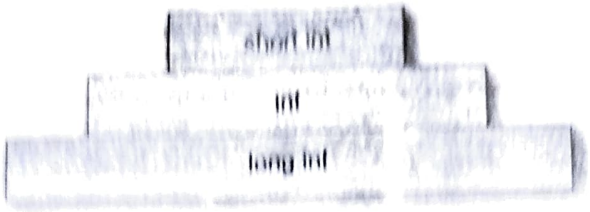


Fig. 2.5 *Integer types*

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

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

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

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

Floating Point Types

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

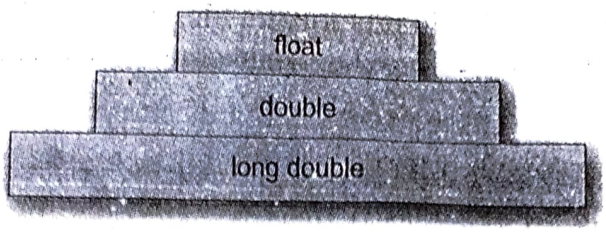


Fig. 2.6 *Floating-point types*

Void Types

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

Character Types

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

2.8 DECLARATION OF VARIABLES

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

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

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

Primary Type Declaration

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

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

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

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

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

Table 2.9 *Data Types and Their Keywords*

Data type	*Keyword equivalent*
Character	char
Unsigned character	unsigned char
Signed character	signed char (or 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.....*/
```

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

Default values of Constants

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

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

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

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

User-Defined Type Declaration

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

typedef *type* identifier;

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

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

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

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

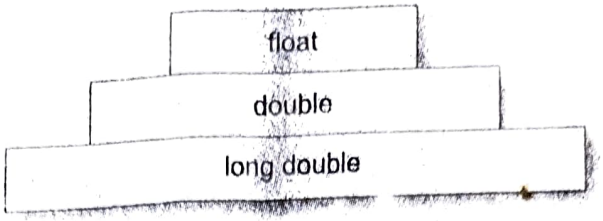


Fig. 2.6 *Floating-point types*

Void Types

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

Character Types

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

2.8 DECLARATION OF VARIABLES

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

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

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

Primary Type Declaration

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

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

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

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

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

Table 2.9 *Data Types and Their Keywords*

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

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

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

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

5

Decision Making and Branching

5.1 INTRODUCTION

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

C language possesses such decision-making capabilities by supporting the following 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 statements '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 shall 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 of execution of statements. It is basically a two-way decision statement and is used in 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 associated with it, one for the *true* condition and the other for the *false* condition as shown in Fig. 5.1.

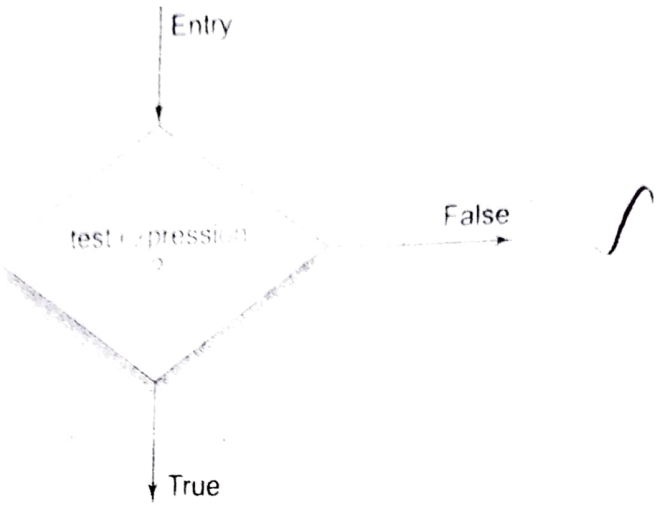


Fig. 5.1 *Two-way branching*

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. [illegible]der.

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 illustrated in Fig. 5.2.

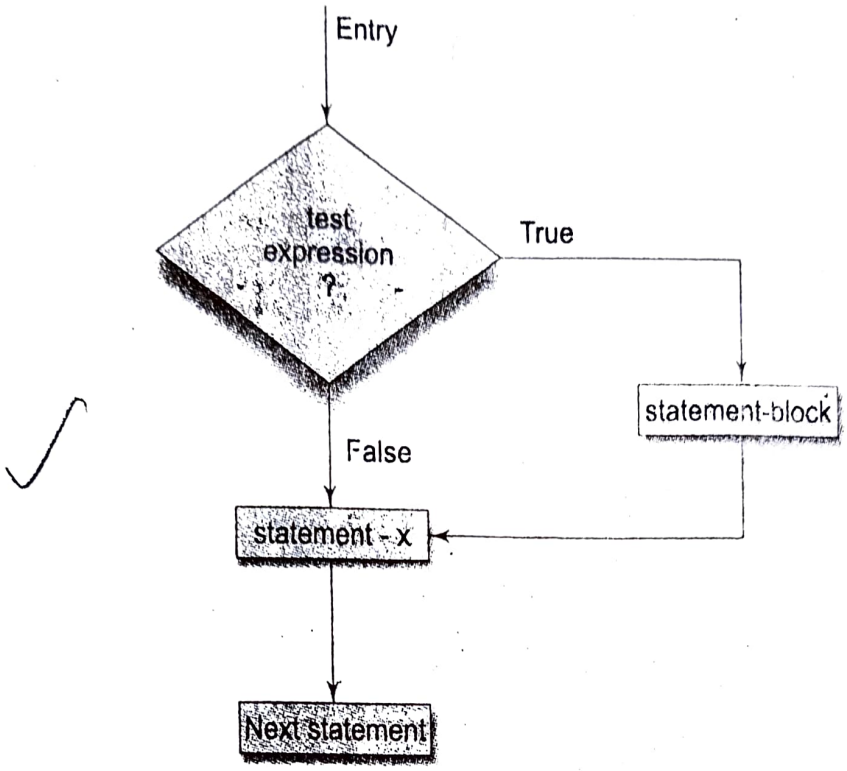


Fig. 5.2 *Flowchart of simple if control*

Consider the following segment of a program that is written for processing of marks obtained 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 terminal and evaluates the ratio of (a+b) to (c-d) and prints the result, if 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 function 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)

This would have 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 in turn is another **if** statement. This **if** statement tests **height** and if the **height** is greater 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 &&

Example:

!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* statement 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.

This would have 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 in turn is another **if** statement. This **if** statement tests **height** and if the **height** is greater 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
```

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.

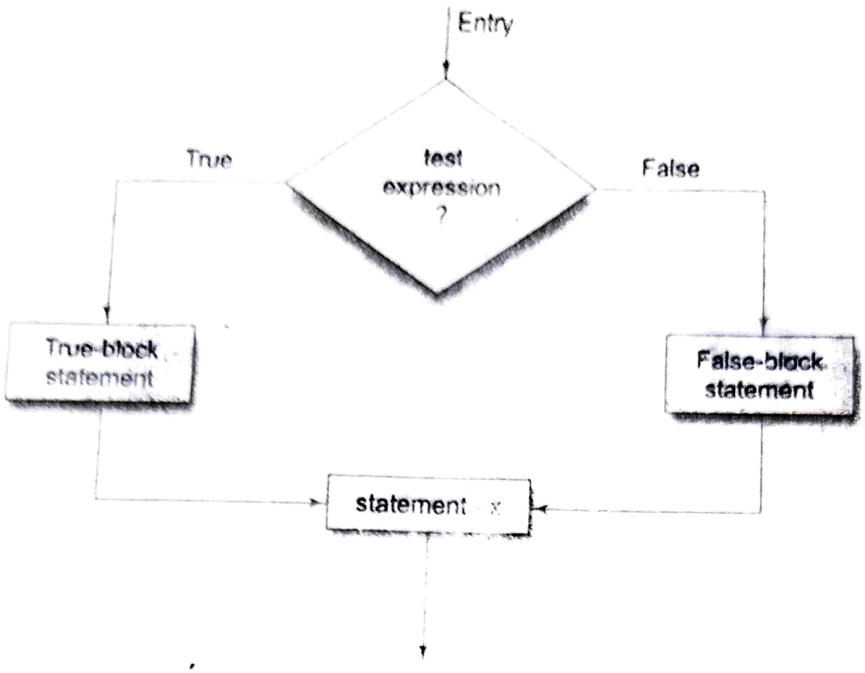


Fig. 5.5 *Flowchart of* ***if.....else*** *control*

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

```
.........
.........
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 is increased by 1 and the program continues to the second test. The second test again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both. 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 when the value of the term is less than 0.0001 (**ACCURACY**). Note that when a term is less than ACCURACY, the value of n is set equal to 999 (a number higher than 100) and therefore the **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...else** 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 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the

```
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.

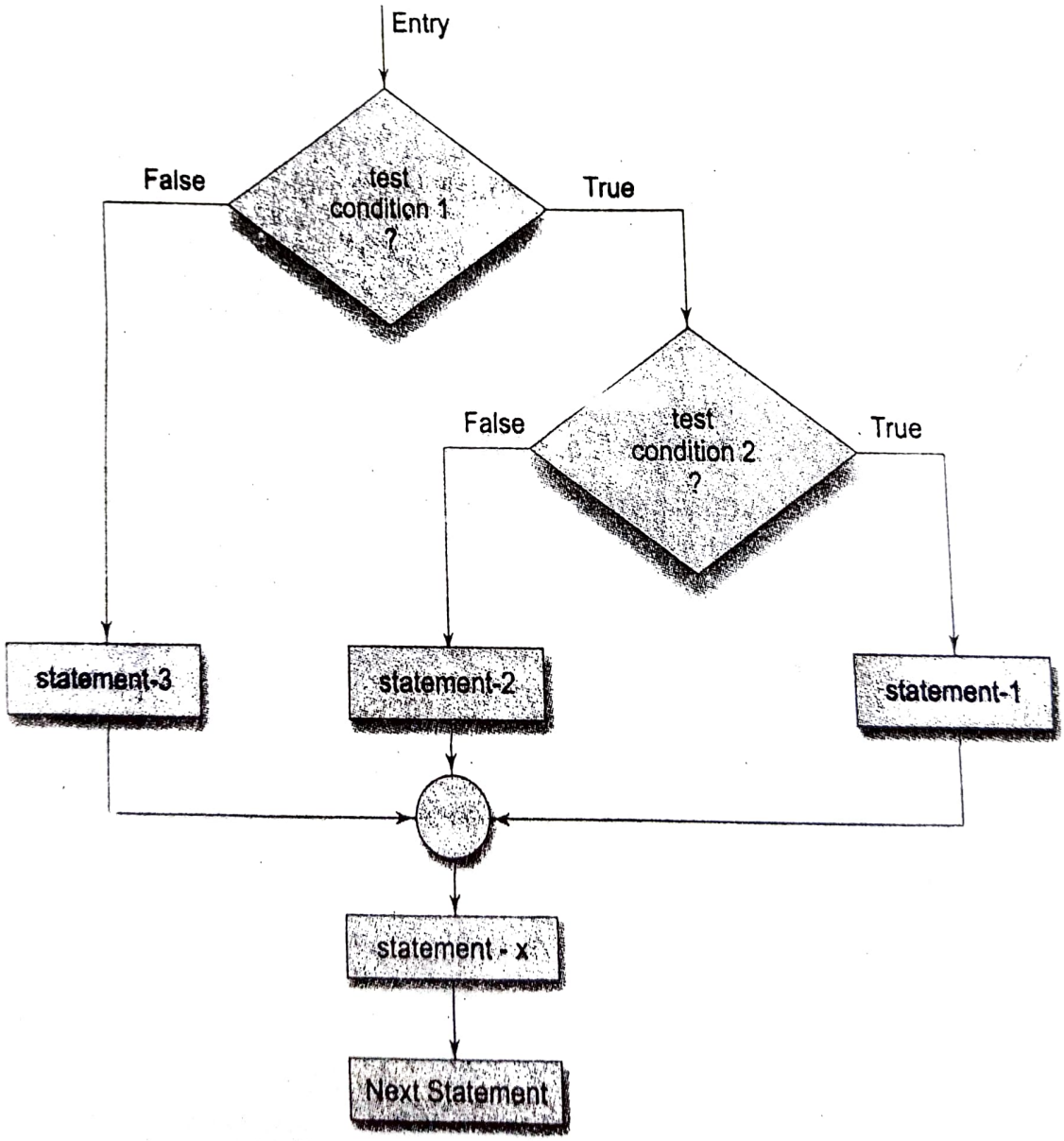


Fig. 5.7 *Flow chart of nested* **if...else** *statements*

```
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 consumers 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 prints 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 Cnarges = 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. 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 or constant expressions (evaluable to an integral constant) and are known as *case labels*. Each of these values should be unique within a **switch** statement. **block-1, block-2** .... are statement lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that **case** labels end with a colon (:).

When the **switch** is executed, the value of the expression is successfully compared against the values *value-1, value-2,...*. If a case is found whose value matches with the value of the 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 and causes an exit from the **switch** statement, transferring the control to the **statement-x** following the **switch.**

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

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

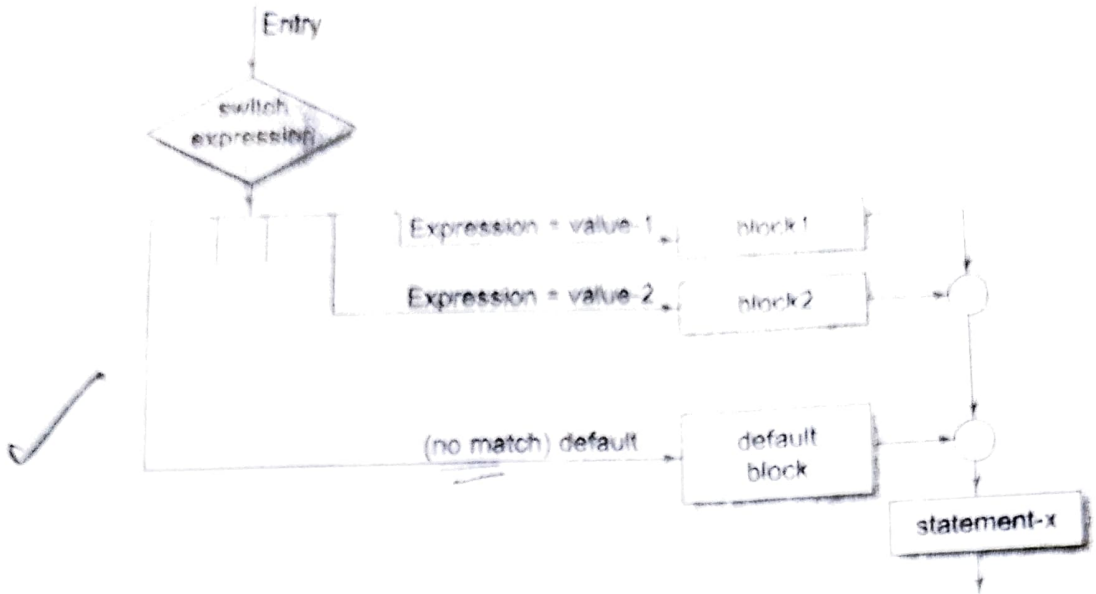


Fig. 5.11 *Selection process of the* **switch** *statement*

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 values.

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 cases. 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 cases 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 st... ment. 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 $x \leq 2$

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

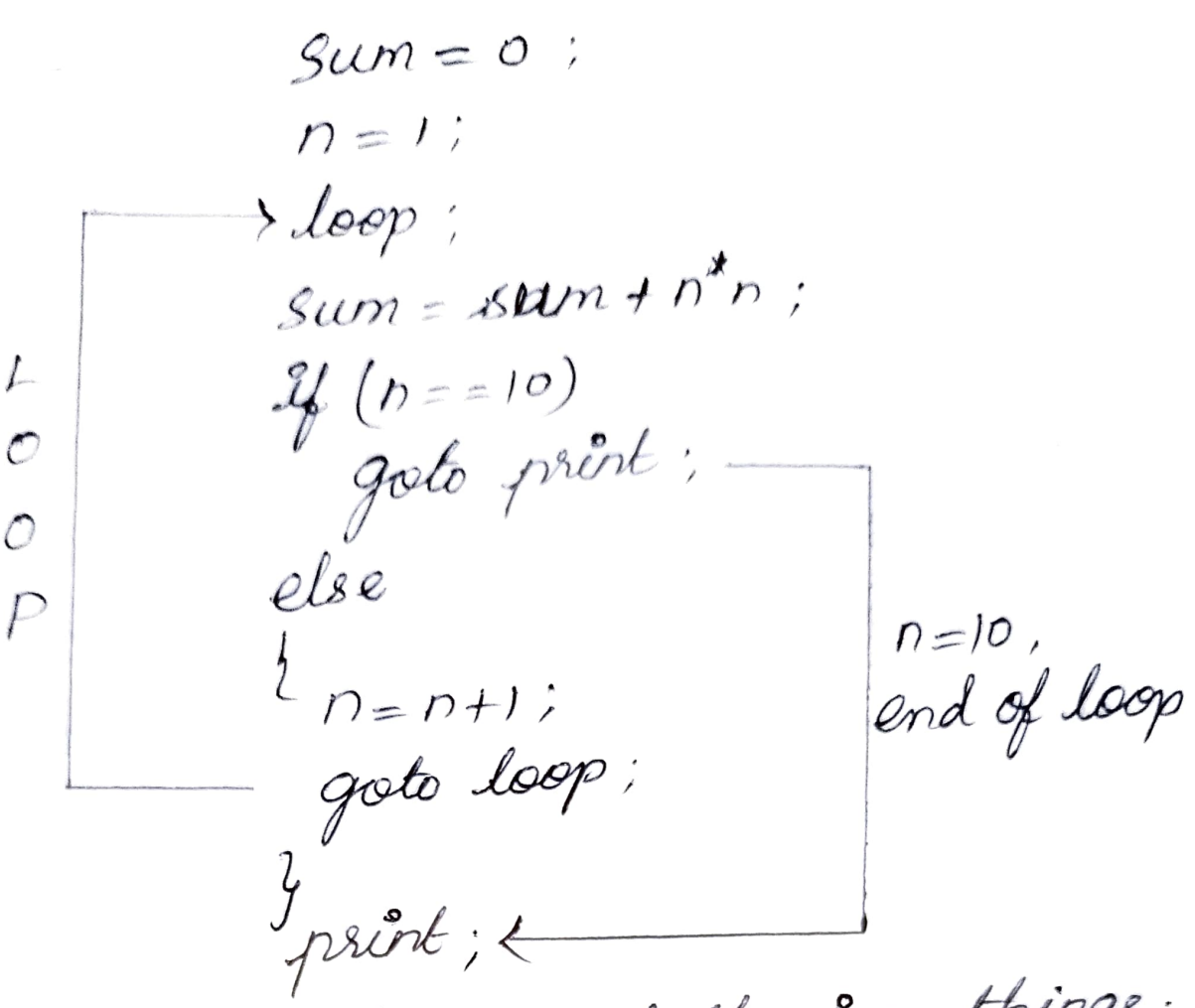
Decision Making and Looping

6.1 Introduction:

We have seen in the previous chapter that it is possible to execute a segment of a progress repeatedly by introducing a counter and later testing it using the if statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop. For example, suppose we want to calculate the sum of squares of all integers between 1 and 10, we can write a program using the If statement as follows:

Balance

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test conditions will eventually transfer the control out of the loop. In case, due to some reasons it does not d loop and the loop body is executed over



This program does the following things;

⇒ Initializes the variable n.

⇒ Computes the square of n and adds it to sum.

⇒ Tests the value of n to see whether it is equal to 10 or not. If it is equal to 10, then the program prints the results.

⇒ If n is less than 10, then it is incremented by one and the control goes back to compute the sum again.

do so, the control sets up an infinite
er and over again

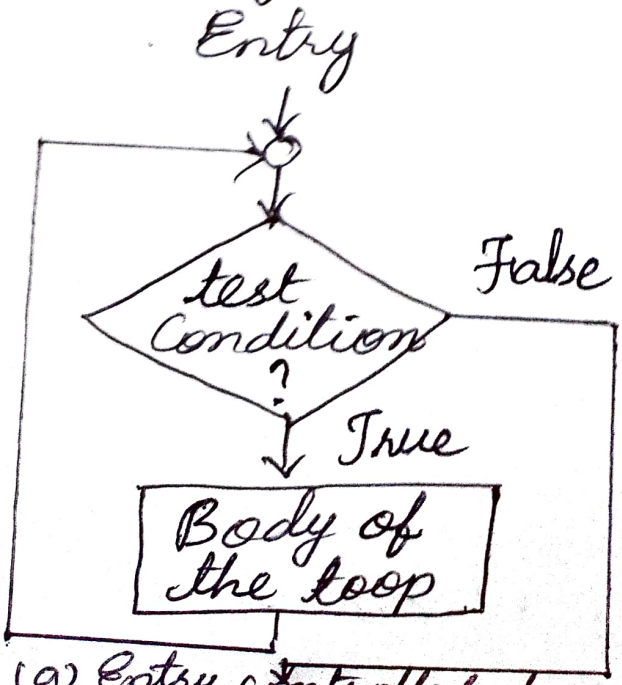
The program evaluates the statement

sum = sum + n * n ;

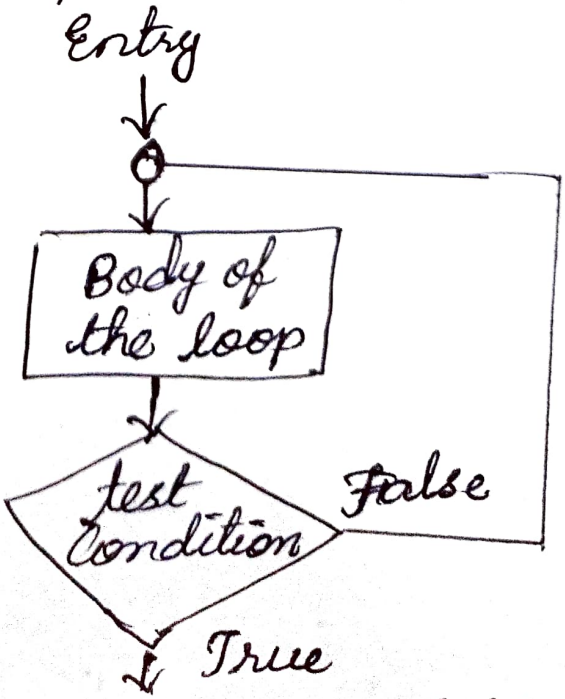
20 times. That is, the loop is executed 10 times. This number can be increased or decreased easily by modifying the relational expression appropriately in the statement if (n==10). On such occasions where the exact number of repetitions are known, there are more convenient methods of looping in C. These looping capabilities enable us to develop concise programs containing repetitive processes without the use of goto statements.

In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the entry-controlled loop or as the exit-controlled loop. The flow charts in illustrate these structures. In the enery-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionly for the first time. The entry-controlled and exit-controlled loops are also known as pre-test and post-test loops respectively.



(a) Entry controlled loop



(b) Exit controlled loop

A looping process, in general, would include the following four steps:

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three *constructs* for performing *loop* operations. They are:

1. The **while** statement.
2. The **do** statement.
3. The **for** statement.

We shall discuss the features and applications of each of these statements in this chapter.

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a *counter-controlled loop*. We use a control variable known as *counter*. The counter must be initialized, tested and updated properly for the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a *sentinel-controlled loop*, a special value called a *sentinel* value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like –1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often called *indefinite repetition loop* because the number of repetitions is not known before the loop begins executing.

6.2 THE WHILE STATEMENT

The simplest of all the looping structures in C is the **while** statement. We have used **while** in many of our earlier programs. The basic format of the **while** statement is

```
while (test_condition)
{
    body of the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition is *true*, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

We can rewrite the program loop discussed in Section 6.1 as follows:

```
========
sum = 0;
n = 1;                          /* Initialization */
while(n <= 10)                  /* Testing */
{                                                    loop
   sum = sum + n
   n = n+1;                     /* Incrementing */
}
printf("sum = %d\n", sum);
========
```

The body of the loop is executed 10 times for n = 1, 2, ..., 10, each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as **n < 11**; the result would be the same. This is a typical example of counter-controlled loops. The variable n is called ***counter*** or ***control variable***.

Another example of **while** statement, which uses the keyboard input is shown below:

```
=========
character = ' ' ;
while (character != 'Y')
       character = getchar();
xxxxxxx;
=========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether **character** is not equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because **character** equals Y, and the loop terminates, thus transferring the control to the statement xxxxxxx. This is a typical example of sentinel-controlled loops. The character constant 'Y' is called *sentinel* value and the variable **character** is the condition variable, which often referred to as the *sentinel variable*.

Example 6.1 A program to evaluate the equation

$$y = x^n$$

when n is a non-negative integer, is given in Fig. 6.2

The variable **y** is initialized to 1 and then multiplied by **x**, **n** times using the **while** loop. The loop control variable **count** is initialized outside the loop and incremented inside the loop. When the value of **count** becomes greater than **n**, the control exists the loop.

Program

```
main()
{
  int count, n;
  float x, y;

  printf("Enter the values of x and n : ");
  scanf("%f %d", &x, &n);
  y = 1.0;
  count = 1;              /* Initialisation */
  /* LOOP BEGINs */
  while ( count <= n)  /* Testing */
  {
    y = y*x;
    count++;              /* Incrementing */
  }
  /* END OF LOOP */
  printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
}
```

Output

```
Enter the values of x and n : 2.5 4
x = 2.500000; n = 4; x to power n = 39.062500
Enter the values of x and n : 0.5 4
x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 6.2 *Program to compute x to the power n using while loop*

6.3 THE DO STATEMENT

The while loop construct that we have discussed in the previous section, makes a test of condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takes the form:

```
do
{
    body of the loop
}
while (test-condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test-condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test-condition* is evaluated at the bottom of the loop, the **do...while** construct provides an *exit-controlled* loop and therefore the body of the loop is *always executed at least once.*

A simple example of a **do...while** loop is:

```
.....
do
{
    printf ("Input a number\n");
    number = getnum ();
}
while (number > 0);
.....
```

(loop)

This segment of a program reads a number from the keyboard until a zero or a negative number is keyed in, and assigned to the *sentinel* variable **number**.

The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

in the above example would cause the loop to be executed as long as the number keyed in lies between 0 and 100.

Consider another example:

```
.....
I = 1;                          /* Initializing */
sum = 0;
do
```

```
        {
            sum = sum + 1;
loop        1 = 1*2;                          * Incrementing *

           while(sum < 40 || 1 < 10);         * Testing *
           printf("%d %d\n", 1, sum);
           --------
```

The loop will be executed as long as one of the two relations is true.

Example 6.2 A program to print the multiplication table from 1 x 1 to 12 x 10 as shown below is given in Fig. 6.3.

1	2	3	4	.....	10
2	4	6	8	.....	20
3	6	9	12	.....	30
4				.....	40
.					
.					
.					
12		.	.	.....	120

This program contains two **do.... while** loops in nested form. The outer loop is controlled by the variable **row** and executed 12 times. The inner loop is controlled by the variable **column** and is executed 10 times, each time the outer loop is executed. That is, the inner loop is executed a total of 120 times, each time printing a value in the table.

```
Program:
    #define COLMAX 10
    #define ROWMAX 12
    main()
    {
        int row,column, y;
        row = 1;
        printf("    MULTIPLICATION TABLE      \n");
        printf("------------------------------\n");
        do /*.....OUTER LOOP BEGINS.....*/
        {
            column = 1;
            do /*.....INNER LOOP BEGINS.....*/
            {
              y = row * column;
              printf("%4d", y);
              column = column + 1;
            }
```

```
            while (column <= COLMAX); /*... INNER LOOP ENDS ...*/
            printf("\n");
            row = row + 1;
        }
        while (row <= ROWMAX);/*..... OUTER LOOP ENDS .....*/
        printf("-------------------------------------------\n");
    }
```

Output

```
            MULTIPLICATION TABLE
-------------------------------------------
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
11  22  33  44  55  66  77  88  99  110
12  24  36  48  60  72  84  96  108 120
-------------------------------------------
```

Fig. 6.3 *Printing of a multiplication table using* ***do...while loop***

Notice that the **printf** of the inner loop does not contain any new line character (\n). This allows the printing of all row values in one line. The empty **printf** in the outer loop initiates a new line to print the next row.

6.4 THE FOR STATEMENT

Simple 'for' Loops

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for ( initialization ; test-condition ; increment)
{
    body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The *test-condition* is a relational expression, such as i < 10 that determines when the loop will exit. If the

condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

NOTE: C99 enhances the **for** loop by allowing declaration of variables in the initialization portion. See the Appendix "C99 Features".

Consider the following segment of a program:

```
loop    for ( x = 0 ; x <= 9 ; x = x+1)
        {
            printf("%d", x);
        }
        printf("\n");
```

This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+1.

The **for** statement allows for *negative increments*. For example, the loop discussed above can be written as follows:

```
for ( x = 9 ; x >= 0 ; x = x-1 )
    printf("%d", x);
printf("\n");
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9; x = x-1)
printf("%d", x);
```

will never be executed because the test condition fails at the very beginning itself.

Let us again consider the problem of sum of squares of integers discussed in Section 6.1. This problem can be coded using the **for** statement as follows:

```
sum = 0;
for (n = 1; n <= 10; n = n+1)
{
    sum = sum+ n*n;
}
printf("sum = %d\n", sum);
```

The body of the loop

```
sum = sum + n*n;
```

is executed 10 times for n = 1, 2, ... 10 each time incrementing the **sum** by the square of the value of n.

One of the important points about the **for** loop is that all the three actions, namely *initialization*, *testing*, and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmers and users, in one place. The **for** statement and its equivalent of **while** and **do** statements are shown in Table 6.1.

Table 6.1 *Comparison of the Three Loops*

for	*while*	*do*

for (n=1; n<=10; ++n)
{
_____
_____
}
 	
	n = 1;
	while (n<=10)
	{
	_____
	_____
	n = n+1;
	}
		n = 1;
		do
		{
		_____
		_____
		n = n+1;
		}
		while(n<=10);

Example 6.3 The program in Fig. 6.4 uses a **for** loop to print the "Powers of 2" table for the power 0 to 20, both positive and negative.

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a ***long int*** and **q** as a **double**.

Additional Features of for Loop

The **for** loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)
```

```
Program
  main()
  {
    long int p;
    int n;
```

Arrays

7.1 INTRODUCTION

So far we have used only the fundamental data types, namely **char, int, float, double** and variations of **int** and **double**. Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as *array* that can be used for such applications.

An array is a *fixed-size* sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day, or a month, or a year.
- List of employees in an organization.
- List of products and their cost sold by a store.
- Test scores of a class of students.
- List of customers and their telephone numbers.
- Table of daily rainfall data.

and so on.

Since an array provides a convenient structure for representing data, it is classified as one of the *data structures* in C. Other data structures include structures, lists, queues and trees. A complete discussion of all data structures is beyond the scope of this text. However, we shall consider structures in Chapter 10 and lists in Chapter 13.

As we mentioned earlier, an array is a sequenced collection of related data items that share a common name. For instance, we can use an array name *salary* to represent a *set of salaries* of a group of employees in an organization. We can refer to the individual salaries by writing a number called *index* or *subscript* in brackets after the array name. For example,

```
salary [10]
```

represents the salary of 10th employee. While the complete set of values is referred to as an array, individual values are called *elements*.

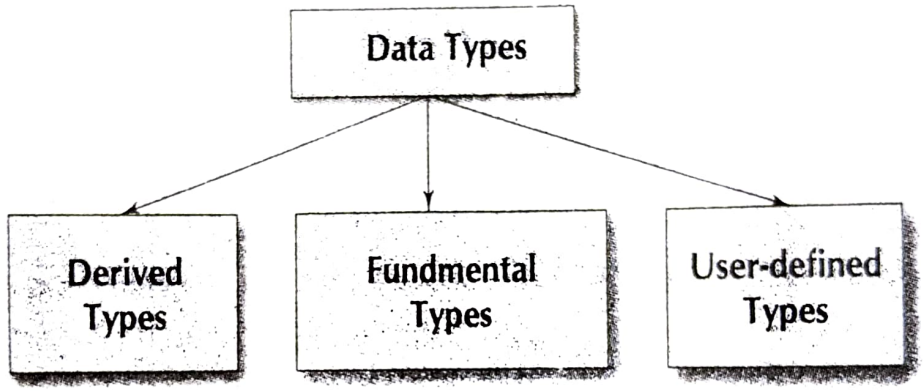
The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs. For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations, and print out the results.

We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions. In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.

- One-dimensional arrays
- Two-dimensional arrays
- Multidimensional arrays

Data Structures

C supports a rich set of derived and user-defined data types in addition to a variety of fundamental types as shown below:



- Arrays	- Integral Types	- Structures
- Functions	- Float Types	- Unions
- Pointers	- Character Types	- Enumerations

Arrays and structures are referred to as *structured data types* because they can be used to represent data values that have a structure of some sort. Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations. In programming parlance, such data types are known as *data structures*.

In addition to arrays and structures, C supports creation and manipulation of the following data structures:

- Linked Lists
- Stacks
- Queues
- Trees

7.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted variable* or a *one-dimensional* array. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^{n} x_i}{n}$$

to calculate the average of n values of x. The subscripted variable x_i refers to the ith element of x. In C, single-subscripted variable x can be expressed as

x[1], x[2], x[3],.....x[n]

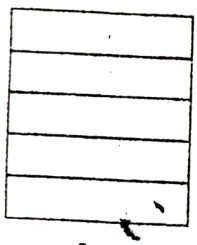
The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable **number**, then we may declare the variable **number** as follows

```
int number[5];
```

and the computer reserves five storage locations as shown below:



number [0]
number [1]
number [2]
number [3]
number [4]

The values to the array elements can be assigned as follows:

```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

This would cause the array **number** to store the values as shown below:

number [0]	35
number [1]	40
number [2]	20
number [3]	57
number [4]	19

These elements may be used in programs just like any other C variable. For example, the following are valid statements:

```
a = number[0] + 10;
number[4] = number[0] + number [2];
number[2] = x[5] + y[10];
value[6] = number[i] * 3;
```

The subscripts of an array can be integer constants, integer variables like i, or expressio that yield integers. *C performs no bounds checking and, therefore, care should be exercised ensure that the array indices are within the declared limits.*

7.3 DECLARATION OF ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

type ***variable-name[size];***

The *type* specifies the type of element that will be contained in the array, such as **int**, **float**, or **char** and the *size* indicates the maximum number of elements that can be stored inside the array. For example,

```
float height[50];
```

declares the **height** to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly,

```
int group[10];
```

declares the **group** as an array to contain a maximum of 10 integer constants. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a numeric constant or a symbolic constant.

The C language treats character strings simply as arrays of characters. The *size* in a character string represents the maximum number of characters that the string can hold. For instance,

```
char name[10];
```

declares the **name** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable **name.**

"WELL DONE"

Each character of the string is treated as an element of the array **name** and is stored in the memory as follows:

'W'
'E'
'L'
'L'
' '
'D'
'O'
'N'
'E'
'\0'

7.4 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain 'garbage'. An array can be initialized at either of the following stages:

* At compile time
* At run time

Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

type array-name[size] = {list of value};

The values in the list are separated by commas. For example, the statement

int number[3] = {0,0,0};

will declare the variable number as an array of size 3 and will assign zero to each element. If the number of values in the list is less than the number of elements then only that many elements will be initialized. The remaining elements will be set to zero automatically. For instance,

float total[5] = {0.0, 15.75, -10};

will initialize the first three elements to 0.0, 15.75, and -10.0 and the remaining two elements to zero.

The size may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

```
int counter [] = {1, 1, 1, 1};
```

Will declare the counter array to contain four elements with initial values 1. This approaches works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char name [] = {'J', 'o', 'h', 'n', '\0'};
```

declares the name to be an array of five characters, initialized with the string "John" ending with the null character. Alternatively, we can assign the string literal directly as under.

```
char name [] = "John";
```

(Character arrays and strings are discussed in detail in chapter 8)

Compile time initialization may be partial. That is, the number of initializers may be less than the

declared size. In such cases the remaining elements are initialized to zero, if the array type is numeric and NULL if the type is character. For example,

int number[5] = {10, 20};

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0. Similarly, the declaration

char city[5] = {'B'};

will initialize the first element to 'B' and the remaining four to NULL. It is a good idea, however, to declare the size explicitly as it allows the compiler to do some error checking. Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement.

int number[3] = {10, 20, 30, 40};

will not work. It is illegal in C.

Run time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
.....
.....
for(i=0 ; i<100 ; i=i+1)
{
  if i<50
     sum[i]=0.0 ; /* assignment
  else                Statement */
     sum[i]=1.0;
}
.....
.....
```

The first 50 elements of the array sum are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

- Bubble sort
- Selection sort
- Insertion sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the *search key*. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search
- Binary search

A detailed discussion on these techniques is beyond the scope of this text. Consult any good book on data structures and algorithms.

7.5 TWO-DIMENSIONAL ARRAYS

So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item1	*Item2*	*Item3*
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four *rows* and three *columns*. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.

In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij}. Here v denotes the entire matrix and v_{ij} refers to the value in the i^{th} row and j^{th} column. For example, in the above table v_{23} refers to the value 325.

C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

```
v[4][3]
```

Two-dimensional arrays are declared as follows:

type *array_name* [row_size][column_size];

Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.

Two-dimensional arrays are stored in memory, as shown in Fig.7.3. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

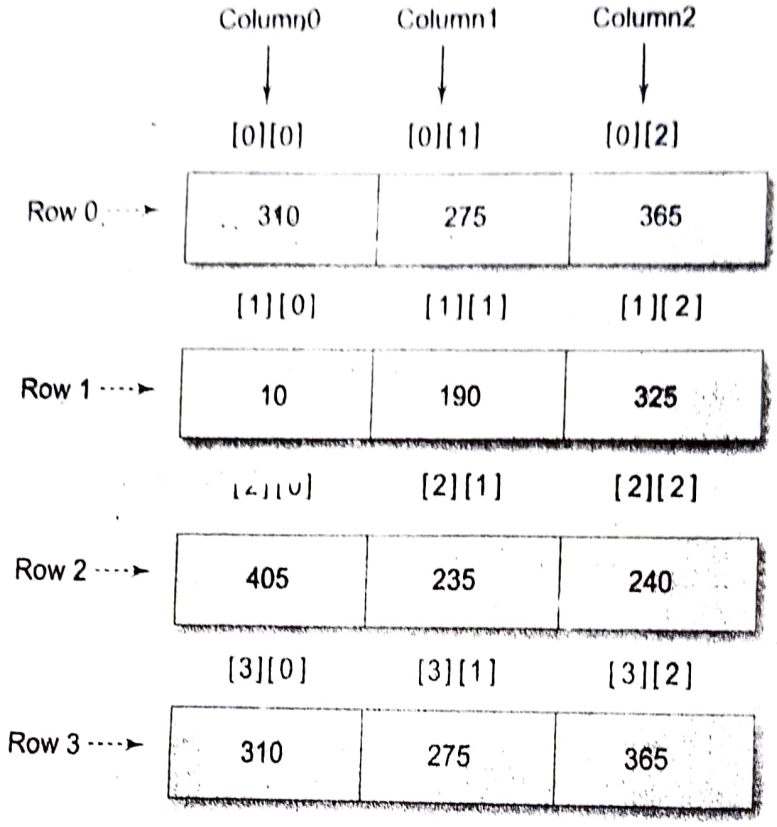


Fig. 7.3 *Representation of a two-dimensional array in memory*

Example 7.3 Write a program using a two-dimensional array to compute and print the following information from the table of data discussed above:
(a) Total value of sales by each girl.
(b) Total value of each item sold.
(c) Grand total of sales of all items by all girls.

The program and its output are shown in Fig. 7.4. The program uses the variable value in two-dimensions with the index i representing girls and j representing items. The following equations are used in computing the results:

(a) Total sales by m^{th} girl = $\sum_{j=0}^{2}$ value [m][j](girl_total[m])

(b) Total value of n^{th} item = $\sum_{i=0}^{3}$ value [i][n](item_total[n])

(c) Grand total = $\sum_{i=0}^{3}\sum_{j=0}^{2}$ value[i][j]

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed. Otherwise the function will never return.

§ 17. PASSING ARRAYS TO FUNCTIONS

One-Dimensional Arrays

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, *without any subscripts*, and the size of the array as arguments. For example, the call

```
largest(a,n)
```

will pass the whole array a to the called function. The called function expecting this call must be appropriately defined. The largest function header might look like:

```
float largest(float array[ ], int size)
```

The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[ ];
```

The pair of brackets informs the compiler that the argument array is an array of numbers. It is not necessary to specify the size of the array here.

Let us consider a problem of finding the largest value in an array of elements. The program is as follows:

```
main( )
{
    float largest(float a[ ], int n);
    float value[4] = {2.5, -4.75, 1.2, 3.67};
    printf("%f\n", largest(value,4));
}
float largest(float a[], int n)
{
    int i;
    float max;
    max = a[0];
    for(i = 1; i < n; ++i)
        if(max < a[i])
```

```
            max = a[i];
        return(max);
    }
```

When the function call **largest**(value,4) is made, the values of all elements of array **value** become the corresponding elements of array **a** in the called function. The **largest** function finds the largest value in the array and returns the result to the **main.**

In C, the name of the array represents the address of its first element. By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing addresses of parameters to the functions is referred to as *pass by address* (or pass by pointers). Note that we cannot pass a whole array by value as we did in the case of ordinary variables.

Example 9.5 Write a program to calculate the standard deviation of an array of values. The array elements are read from the terminal. Use functions to calculate standard deviation and mean.

Standard deviation of a set of n values is given by

$$S.D. = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{x} - x_i)^2}$$

Where $\bar{x}$ is the mean of the values.

Program

```
#include     <math.h>
#define SIZE    5
float std_dev(float a[], int n);
float mean (float a[], int n);

main( )
{
     float value[SIZE];
     int i;

     printf("Enter %d float values\n", SIZE);
     for (i=0 ;i < SIZE ; i++)
         scanf("%f", &value[i]);
     printf("Std.deviation is %f\n", std_dev(value,SIZE));
}

float std_dev(float a[], int n)
```