OVERVIEW OF C

1.1 Input and Output Statements

Data input to the computer is processed in accordance with the instructions in a program and the resulting information is presented in the way that is acceptable to the user. For example, to compute c=a+b, the computer should know the values of *a* and *b*. Giving the values to such variables is done in two ways:

- Assigning the values to variables in the program code
- Reading the data from the key board during run time. That is, asking the user to provide the values for the variables during run time.

The second method is efficient because, the program should be flexible to accept any valid data and provide the result accordingly. The process of feeding the data to the computer is known as input and the process of computer displaying the result/data is called as output.

Input and output can be either through console or file in C programming language. Here we will discuss, the built-in functions provided by C language for doing I/O operations through console.

Type	Function	Operation
Formatted I/O Functions	scanf()	Reads one or more values (data types: int, float, char, string) according to user's format specification
	printf()	Outputs one or more values (data types: int, float, char, string) according to user's format specification
Un-formatted I/O Functions	getch()	Reads single character (un-buffered)
	getche()	Reads single character and echos the same (un-buffered)
	putch()	Outputs single character
	getchar()	Reads single character. But enter key has to be pressed after inputting the character
	putchar()	Outputs single character
	gets()	Reads a string until enter key is pressed
	puts()	Outputs a string

The above functions are discussed in detail here.

1.2 Formatted Console Input Function – scanf()

This function is used to input data through the standard input file viz. keyboard. Basic data types like int, float, char etc. and strings can be read using this function. The general form is –

scanf(control string, list of variables);

Here, the first argument *control string* consists of conversion specifiers. The second argument is list of variables whose values are to be input. The control string and list of variables are separated by a comma.

Example:

```
int a;
float f;
char ch;
scanf("%d%f%c", &a, &f, &ch);
```

The combination of variable name and '&' symbol gives the address of memory location where the corresponding input data is to be stored.

Control string consists of a list of conversion specifiers, wherein each conversion character is preceded by a percent symbol. Data input through the keyboard is actually the text consisting of letters, digits and punctuation symbols and is accepted in the binary form into the computer's memory. That is, if your to input 25 for an integer variable, you will actually key in the text consisting of 2 and 5. The conversion specifiers say %d, %f etc are then used to translate the input information to integer and float data types respectively. The data type order and number of specifiers in the control string should match with those of variable list.

List of Conversion Specifiers

Conversion Specifiers	Data Types
%c	Single character
%d or %i	Integer
%ld	Long int
%u	Unsigned int
%f	Floating point
%e	Floating point value in exponential form
%o	Octal value
%lo	Octal long
%x	Hexadecimal lower case a-f
%X	Hexadecimal upper case A-F
%s	string

1.2.1 Rules for scanf()

There are certain rules to be observed carefully while using scanf() function. Few are listed below.

- scanf() function skips white space characters, new line, tabs etc. before an integer or a float value in the input. Example:

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

  For the above statement, the user input can be either

  ```
          25 32                                  (separated by space)

  Or      25              32                     (separated by tab)

  Or      25
          32                                     (separated by new line)
  ```

- Spaces in the control string consisting of conversion specifiers for numerical data type are ignored. That is :

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

  here, space between %d and %f makes no difference.

- Ordinary characters may be placed in the control string region. This rule allows us to introduce comma between two conversion specifiers:

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

  But, now the input also should contain the data items separated by a comma symbol. That is the input should be either:

  ```
          25,  3.2                (separated by comma and then space)
  Or      25              , 3.2   (separated by comma and then tab)
  Or      25,
          32                      (separated by comma and then new line)
  ```

- But, rules of ignoring white space characters before conversion specifiers for character type of data is different:
  - When a white space character is not included before %c in the control string, the white space character in the input is ***not skipped.***

    ```
    int a;
    char ch;
    scanf("%d%c", &a, &ch);
    ```

    Now, if the input is

    ```
    5x
    ```

    then, *a* will get the value 5 and *ch* will get x.
    But, if the input is

5 x

then, *a* will get the value 5, but *ch* will get the **blank space** as a character.

- A space before %c in the control string leads to ignoring white space characters before the character data in the input.

```
int a;
char ch;
scanf("%d  %c", &a, &ch);
```

Now, if the input is

5x

or

5 x

the variables *a* and *ch* will take the values 5 and x respectively.

- Variable name belonging to string data type (i.e. character array) need not be preceded b an ampersand (&), because name of a string itself is the address of the string.

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

The input can be

Ramu 25 75.4

1.3 The formatted output – printf()

The data can be output to a standard output device using the function printf(). The general form is –

```
printf(control string, Variable List);
```

Here, control string consists of conversion specifiers as in case of scanf(). Along with conversion specifiers, a string of characters and escape sequence characters can also be used for improved presentation of data. Variable list contains the list of variables whose values are to be output. But, these variable names are not preceded by ampersand (&), since they do not specify the memory addresses. The variable list also may contain arithmetic expressions, names of library functions and user defined functions.

1.4 The Escape Sequence

The escape sequences are used for cursor movements and for printing certain characters such as double quote, back-slash etc. Typical escape sequence used for cursor movement is the tab control character (\t), which shifts the current active position of cursor through 8 character positions horizontally. The escape sequences are very useful for displaying data according to the required format.

Escape Sequence character	Meaning
\a	Audio alert
\b	Back space
\f	Form feed
\n	New line
\r	Carriage Return
\t	Horizontal tab
\v	Vertical Tab
\\	Back slash (\)
\’	Single quote (‘)
\”	Double quote (“)

1.5 Space Specifications

For more meaningful input and output, it is essential to specify the number of spaces and starting locations for data items. Format specification is particularly useful for displaying floating point values according to required accuracy. This is done by specifying number of spaces for the fraction part.

The method of space specification for integer, float and string data type items are discussed here.

1.5.1 Integer Data Type

The syntax of space specification for integer data type is

%wd

where, *w* is total number of spaces required for an integer number.
Let us consider an example:

int a=359;

- Now, the statement

  printf(“%3d”, a);

  will generate the output on the monitor as –

3	5	9

  That is, without leaving any space at the top-left corner of the monitor, the number will be displayed.

- The statement

  printf(“%2d”, a);

will also display on the monitor as –

3	5	9

Since width specified is lesser than the actual width needed for the data, compiler will provide enough space.

- The statement

  printf("%5d", a);

  will generate the output on the monitor as –

		3	5	9

Here, two additional spaces are left blank at the top-left corner of the monitor and then the number is displayed.

1.5.2 Floating point Data type

The syntax of space specification for float data type is

%w.df

where, *w* is total number of spaces required which is inclusive of the space required for decimal point and *d* is number of spaces required for fractional part.

Consider an example:

float a=45.765322;

- The statement

  printf("%8.5f", a);

  will generate the output as –

4	5	.	7	6	5	3	2

Here, total spaces used including the dot is 8 and the number of digits after decimal point is 5.

- The statement

  printf("%5.2f", a);

  generates

4	5	.	7	7

Here, note down the value after decimal point. The fractional part 765322 has been rounded off to two places and resulted into 77.

- The statement

  printf("%8.2f",a);

  indicates the total spaces to be used is 8, but the fractional part should be of 2 digits. Hence, in the display, first 3 spaces are left blank as shown below –

			4	5	.	7	7

- The statement

  printf("%3.3f", a);

  indicates total spaces should be only 3. But, for the given value of variable *a,* this is not possible. Because, the integral part itself has 2 spaces (45) and the dot requires one space. Hence, the compiler will ignore this instruction partially, and generate the output as –

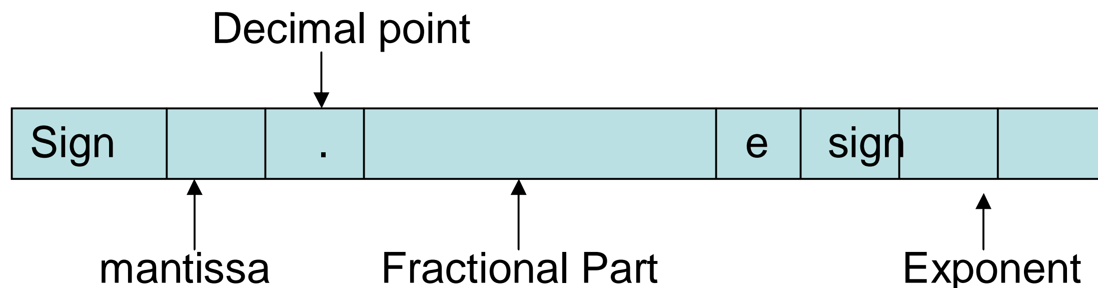
4	5	.	7	6	5

1.5.3 Floating point data with exponential form

The syntax of space specification for exponential form of input/output is

%w.de

where, *w* is total number of spaces required and *d* is number of spaces required for fraction part. The number of spaces provided for the fraction part is according to the required precision. Total number of spaces to be provided for the number will be 7 spaces required for sign of number, mantissa, decimal point, exponent symbol, sign of exponent and exponent itself plus the spaces required for the fraction part.



Let us consider an example:

```
float a=72457631;
printf("%12.4e", a);
```

The output would be –

		7	.	2	4	5	8	e	+	0	1

1.6 Un-formatted I/O Functions

The unformatted I/O functions are used for characters and strings. These functions require less time compared to their formatted counterparts.

1.6.1 getch(), getche() and putch()

The formatted input function scanf() is a buffered function. That is, after inputting the data, one needs to press the enter key. The input data will first be stored in the temporary memory buffer and after pressing the enter key, the data will be pushed to the actual

memory location of the variable. There are two functions viz. getch() and getche() that are un-buffered. That is, these functions do not require the enter key to be pressed after inputting the character.

Example:

```
char opt;
opt=getch();
```

The above set of statements requires a single character to be input from the keyboard. The first character typed will be taken as the input. **But, the character entered by the user is not visible on the monitor.** On the other hand, C provides one more function getche() which ***echoes*** the character typed on to the monitor.

Example:

```
char opt;
opt=getche();
```

Here, the character typed will be displayed on the monitor and without the need of enter key, the typed character will be stored directly on to the physical address of the variable.

The function putch() will display one character.

Example:

```
char opt= 'Y' ;
putch(opt);            // Y will be displayed on to the monitor.
```

1.6.2 getchar() and putchar()

The getchar() function reads one character from the key board. It is a buffered function. That is, the input character is first stored in the buffer. After pressing the enter key, the input is stored in the actual physical memory. Also, the character typed will be displayed on the monitor. This feature of getchar() facilitates the user to make correction in the input (if necessary) before pressing the enter key.

Example:

```
char opt;
opt=getchar();
```

The putchar() function displays one character on the monitor.

Example:

```
char opt= 'Y' ;
putchar(opt);          // Y will be displayed on to the monitor.
```

The putch() function is available in the header file *conio.h* and this header file is not supported by some of the compilers. Whereas the function putchar() is from *stdio.h* and hence available in all compilers.

1.6.3 gets() and puts()

The gets() function is used to read strings (array of characters). The scanf() function for reading a string (with format specifier %s), requires the string input without any white spaces. That is, for example, we can not give the input as “hello, how are you?” using scanf() function. Whereas, gets() function reads the string till the enter key is pressed.

Example:

```
char str[25];
gets(str);      //reads array of characters till enter key is pressed.
```

The puts() function will display the string on to the monitor.

Example:

```
char str[25]=”Hello, How are you?”;
puts(str);             // displays Hello, How are you? On to the monitor
```

1.7 Control Structures

Usually, the statements of a program are executed sequentially. That is, every statement in a program is executed one after the other in the order in which they are written. This kind of linear structure is useful when the values of variables must be generated in a sequential order. But some of the problems will be solved based on decision of a situation. So, in such programs, block of statements must be executed depending on several alternative situations. The selective structures are introduced in C to handle such situations. Thus, *the statements that can alter the sequence of execution of the program are called as control statements.*

Control statements are broadly classified into two categories viz.

i) Selective Control Structure
ii) Repetitive Control Structure (Looping statements)

1.8 Selective Control Structures

Selective control structures are also known as *branching statements.* These statements provide decision making capability to the programmer. C/C++ programming language provides conditional control structures like –

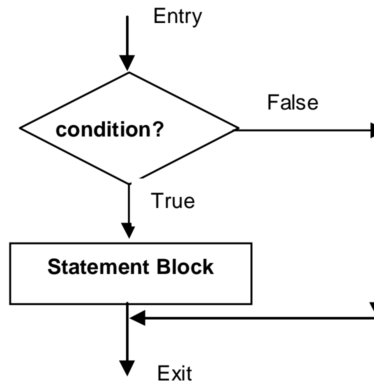
i) ***if*** statement
ii) ***if – else*** statement
iii) ***else – if*** ladder
iv) nested – ***if*** statement
v) ***switch*** statement
vi) ***go to*** statement

1.8.1 The *if* Statement

The *if* statement is one-way conditional branching statement in which a block (or set) of statements can be executed or skipped depending on whether a particular criterion is satisfied or not.

Syntax:

```
if(condition)
{
      Statement block;
}
```



Here,

condition can be a arithmetic/logical/relational expression, a variable, a constant. It can even be a combination of one or more of all these.

Statement block is a set of statements constituting a *true block* of *if* statement. There can be one or more statements. They will be executed when the **condition** is true.

Working Procedure: The execution of **if** statement begins by evaluating the **condition.** It may result in either **true (any non-zero number)** or **false (zero)**. When the value of **condition** is true, the statement – block is executed and then the program control is passed to the statement following the **if**-structure. Otherwise, the statement – block will be skipped without execution and the program control is directly passed to the next statement of **if** structure.

Following is an example showing the usage of **if** statement. Here, a variable is read from the key-board. And then it is checked for negative or positive. If the variable **p** is negative, its absolute value is calculated and printed.

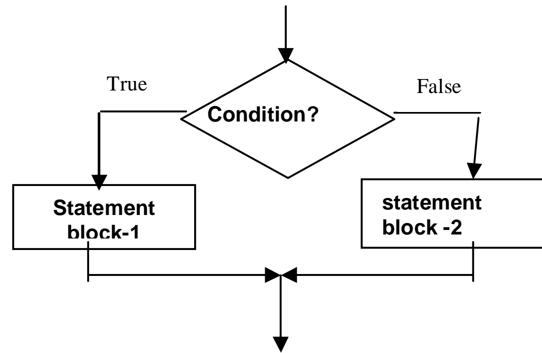
```
int p, q;
scanf("%d", &p);

if(p<0)
{
      printf("p is negative");
      q=abs(p);
      printf("\n The absolute value is ", q);
      exit (0);
}
printf("p is positive");
```

1.8.2 The *if-else* Statement

It is a two-way decision making structure, which executes any one of the two statement blocks based on the evaluated value of expression.

Syntax:

```
if(condition)
{
   statement block -1;
}
else
{
   Statement block -2;
}
```



Here,

condition can be a arithmetic/logical/relational expression, a variable, a constant. It can even be a combination of one or more of all these.

True block is a set of statements constituting a *true block* of *if* statement. There can be one or more statements. They will be executed when the **condition** is true.

Else block is also a set of statements and will be executed when the **condition** is false.

Working Procedure: The execution of **if-else** statement begins by evaluating the **condition.** When the value of **condition** is true, the statement block-1 is executed; otherwise, the statement block-2 is executed. After executing any one of these blocks, the program control is passed to the statement following the **if**-**else** structure.

Example:

```
int p, q;
scanf("%d", &p);
if(p<0)
     printf("p is negative");
else
     printf("p is positive");
```

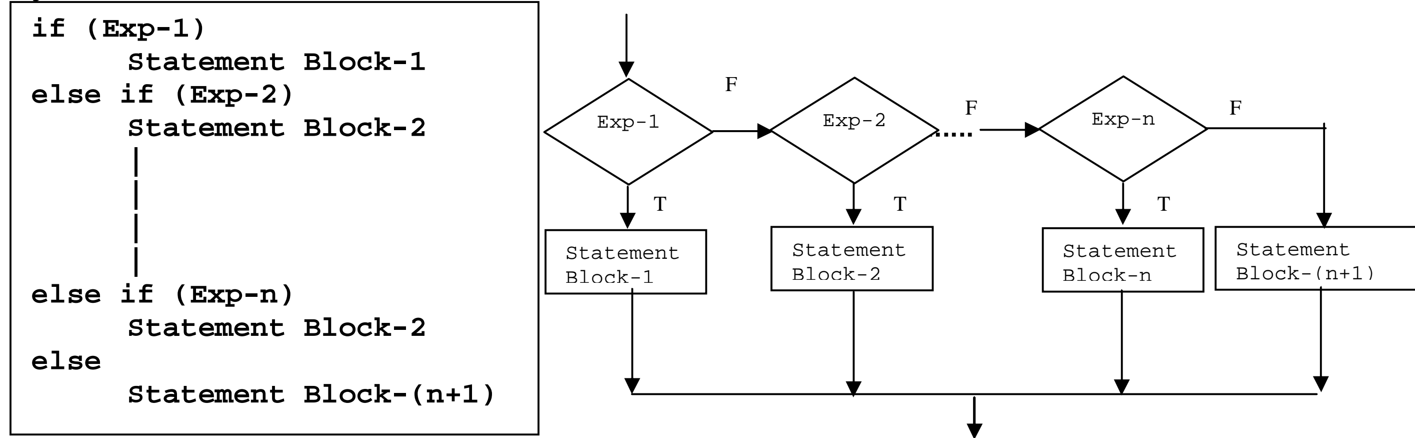
1.8.3 The *else-if* Ladder

Some times, it is necessary to take one out of many alternatives. In such situations, multiple conditions are to be checked and based on one of the decisions, the block of code must be executed. C/C++ provides multiple choices with else-if known as the *else-if ladder*. This is also known as *multi-way else-if structure.*

Working Procedure: The execution of **else-if** ladder begins by evaluating the **Exp-1.** When the value of **Exp-1** is true, the statement block-1 is executed. If it is false then the **Exp-2** is checked. If this value is true, then statement block-1 is executed. Continuing in this way, the expressions are checked until one of them evaluates true. If none of the expressions is true, then the statement block-(n+1) is executed. After executing any one

of these blocks, the program control is passed to the statement following the **else-if** ladder.

Syntax:



Example:

```
int marks;
scanf("%d",&marks);

if (marks >= 80)
     printf("First Class with Distinction");
else if (marks >= 60 && marks < 80)
     printf("First Class");
else if (marks >= 50 && marks < 60)
     printf("Second Class");
else if (marks >= 35 && marks < 50)
     printf("Third Class");
else
     printf("Fail");
```

1.8.4 Nesting of Conditional Statements

Insertion of one or more conditional statements within another conditional statement is known as *Nesting of Conditional Statements*. This method divides the conditions into parts and reduces the complexity of the logic to be implemented. Consider some of the examples:

Example 1:

```
if (marks>=60)
{
     if (marks<70)
          printf("First Class");
     else
          printf("Distinction");
}
```

Here, the program control is sent to inner if statement only when the expression at outer if i.e. marks>=60 is true. When both outer and inner expressions are true, it is printed as first class. But, in case, outer expression is true and inner expression is false, then it is printed as distinction.

Example 2:

```
if (sex == 'M')
{
      if (age >= 21)
            printf("Boy, Eligible for Marriage");
      else
            printf("Boy, Not Eligible for Marriage");
}
else
{
      if (sex == 'F')
      {
            if (age >= 18)
                  printf("Girl, Eligible for Marriage");
            else
                  printf("Girl, Not Eligible for Marriage");
      }
}
```

Note that, the nested conditional structures can be replaced by else-if ladder by using relational and logical operators within expressions.

1.8.5 The *switch-case* Statement

If there are multiple conditions to be checked then the better way is to use else-if ladder. But if there is only one expression which may result into multiple values, then the switch statement is a better alternative.

Syntax:

```
switch (expression)
{
      case value-1: statement-1;
                    break;

      case value-2: statement-2;
                    break;
                |
                |
      case value-n: statement-n;
                    break;

      default:   statement-(n+1);

}
```

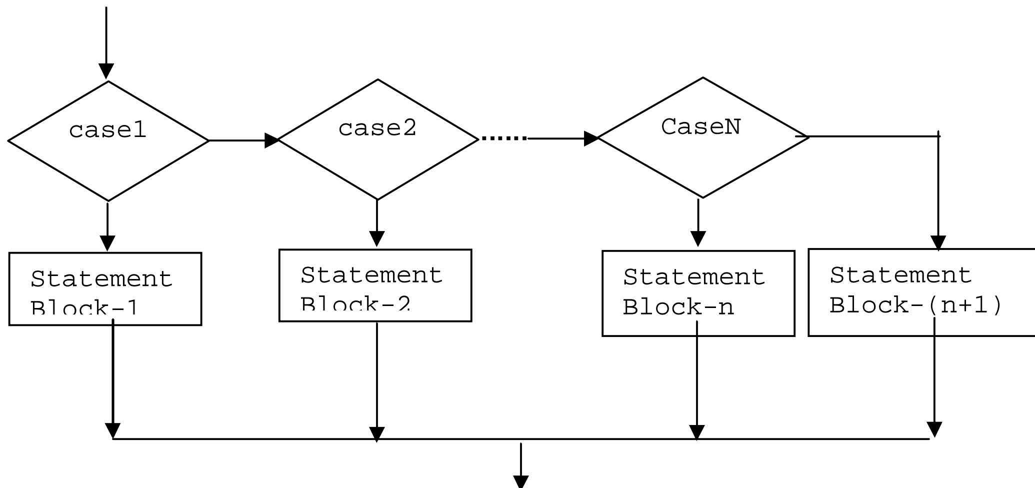
Here

switch, case, default, break are Keywords

expression which is evaluated into either an integer constant or a character constant.

value-1……value-n are the constant values with which the value returned by the expression is compared.

Flowchart:



Working Procedure: Once the control encounters the statement containing the keyword *switch*, expression following the switch is evaluated. The evaluated value is then compared with the value at first case statement. If the value of expression is equal to value-1, then the statements following the first case statement are executed until a break statement is found. Therefore, it is very essential to include a break statement at the end of each statement block. The statement containing break sends the control out of the switch.

Further, if the expression value is not equal to value-1, control directly jumps to the statement containing the next case. Now the value of expression is compared with value-2. If they are equal, the statement-2 gets executed. This procedure continues for all the case values. Thus, the expression value is compared with each case values sequentially until a match is found. Then the statements contained in that block are executed and the control is transferred outside the switch by encountering break statement. If none of the values are matched with expression value, then the statements at default are executed and by encountering closing brace (**}**), the program control comes out of the switch statement.

Note:

1. The case statement must end with a colon (**:**)
2. The default keyword should also end with a colon (**:**), which is an optional block executed only when no case values are equal to the evaluated expression value.
3. The default block should be the last block inside the switch.
4. break is also optional, but it is essential to maintain the distinction between the statement blocks of each cases.
5. There is no need of break in the default block, since default is the last block and control will be passed outside the switch automatically.
6. The values with several cases need not be in a sorted order.
7. More than one statement for a case statement need not be enclosed within braces. As the last statement is break, the program control jumps out the switch statement.

8. Nesting of switch statements is allowed. That is, it is possible to have one switch statement within a case of another switch statement.

Example:

```
int i;
printf("Enter a positive integer (1 to 3) :");
scanf("%d", &i);

switch(i)
{
      case 1:printf("One");
              break;

      case 2: printf("Two");
              break;
      case 3: printf("Three");
              break;
      default: printf("Entered value is not between 1 and 3");
}
```

1.8.6 The *goto* Statement

It is an unconditional statement used to pass the program control from one statement to any other statement in a program. The syntax is –

```
go to label;
:
:
:
label: statement;
```

Here, *label* is used with a statement to which the control is to be transferred. It can be any valid name just like a variable name. But, declaration is not necessary for labels. Consider an example:

```
int i=0, sum=0;

start: i++;
       sum+=i;
      if(i<=10) go to start;
```

In the above example, value of i is incremented and is added to sum until i becomes greater than 10.

Note that as the *goto* statement is unconditional, it may enter into infinite loop, if proper care is not taken. Moreover, it leads to unstructured programming. Hence, the

programmer is advised to not to use it as many of other control structures are available to satisfy the needs of programming.

1.9 Repetitive Control Structures

Some of the programming statements need to be executed repeatedly. These statements may be repeated either for a pre-defined number of times or till a condition becomes false. For doing such repetitive tasks, C/C++ provides three looping structures viz.

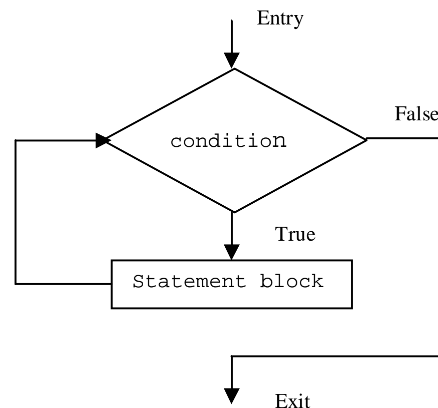
- **while loop**
- **do-while lop**
- **for loop**

1.9.1 The *while* Loop

The *while* loop executes a block of statements as long as a condition is true. When the condition becomes false, the while loop terminates. The syntax and flow-charts are as given below:

Syntax:

```
while(condition)
{
      Statement Block
}
```



Working Procedure: When the program control encounters the *while* loop, it checks the condition. The condition can be any arithmetic/logical/relational expression. If the condition is true, the statement block is executed. Again the condition is checked. If it is still true, the statement block is executed for the second time. This procedure of checking the condition and executing the statement block is continued till the condition becomes false. When the condition becomes false, the control is passed to the next statement following the *while* loop. If the condition is false for the very first time, then the statement block never gets executed. Thus, *while* loop is also known as *entry-check* loop.

Example:

```
int i=1;
while(i<=10)
{
      printf("%d", i);
      i++;
}
```

In the above example, after initializing value of i to 1, the condition within while is checked. Since i<=10 is true, the control enters the loop. Now, value of i is printed and incremented to 2. Again the condition i<=10 is checked. Since it is true, value of i (now, 2) is printed and then incremented. This procedure is continued till i become greater than 10 (i.e. 11). When i become 11, the condition becomes false and hence the control will come-out of the loop. By the time, the values from 1 to 10 would have been printed.

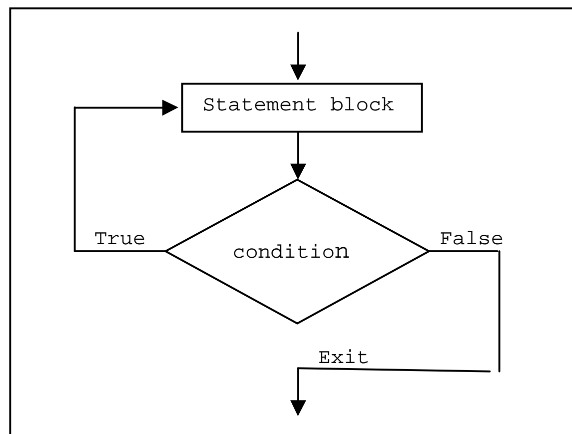
1.9.2 The *do-while* Loop

In this loop, the statement block is executed initially and then the condition is checked. If the condition is true, the statement block is executed for the second time. Continuing in this manner, the statement block is executed several times until the condition becomes false. Following are syntax and flow-charts for do-while loop.

Syntax:

```
do
{
      Statement Block
} while(condition);
```

Flow Chart:



Irrespective of condition, the statement block is executed at least once in this loop as condition is checked after executing the statement block. Thus, the do-while loop is also known as *exit-check* loop.

Example:

```
int i=1;
do
{
      printf("%d", i);
      i++;
} while(i<=10);
```

In this example, after initializing value of i to 1, it gets printed and then incremented. Now, the condition is checked whether i<=10. Since it is true, value of i gets printed for second time and then incremented. This process is continued till i become 11. By the time, values from 1 to 10 will be printed.

Note: The difference between while loop and do-while loop are listed below:

The Difference between while and do-while loops

while Loop	do-while Loop
1. The condition is checked at the beginning. So, it is entry-check loop.	1. The condition is checked after the statement block. Thus, it is exit-check.
2. As it is entry-check, if the condition is false for the first time, the statement block never gets executed.	2. Since it is, exit-check, even if the condition is false for the very first time, the statement block gets executed at least once.

1.9.3 The *for* Loop

The programmer can go for while or do-while loops when the number of times the statement block to be executed is unknown at the beginning. But, if it known well-in-advance, then one can use for loop. The syntax and flow-charts of for loop are:

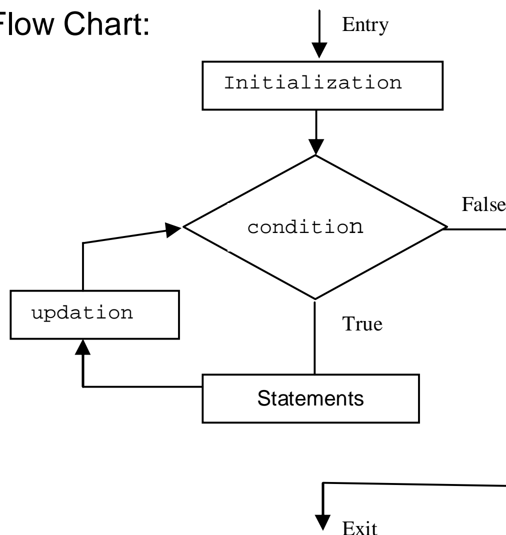
Syntax:

```
for(initialization; test-condition; updation)
{
     Statement block
}
```

Here, *initialization* is initialization of counter variable
test-condition is any expression resulting true/false
updation is an expression that updates the value of counter variable

Note that, these three portions are optional, but the body of the loop should be proper so that, the program should not end-up in infinite number of loops.

Flow Chart:



Working Procedure: When the program control encounters for loop, the initialization part is executed. Then, the test-condition is checked. If it is true, then the statement block is executed. Now, update-assignment is encountered. After updation, the condition is

checked once-again. If it is still true, statement block is executed for the second time. Again updation occurs and condition is checked. Continuing in this manner, the statement block is executed until the condition becomes false. Afterwards, the control will come out of the loop.

Thus, it is observed that *initialization* happens only once. The procedure of condition checking-execution of statement block-updation will happen in a sequence till the condition becomes false.

Example:

```
int i;
for(i=1;i<=10;i=i+1)
{
     printf("%d",i);
}
```

Within the *for* loop, value of i is assigned to 1. Now the condition i<=10 is checked. Since it is true, the control enters the loop and prints the value of i. now, the update statement i=i+1 is executed. Again the condition is checked. It is still true and hence value of i is printed once again. This procedure continued till i becomes 11 and by the time, values from 1 to 10 would have been printed.

Variations in *for* Loop:

Though the general form of *for* loop is as given at the beginning of this section, there may be several variations as per the requirement of the program. Consider following few variations (with examples):

- The updation in counter variable can be in steps of any number. For example, the following code segment will print every 3rd number starting from 1 to 50.

  ```
  for(i=1; i<50; i=i+3)
          printf("%d", i);
  ```

  It will print the numbers: 1 4 7 10 etc

- The update assignment can be skipped from the *for* statement and it can be placed at appropriate position in the body of *for* loop.

  ```
  for(i=1; i<10;)
  {
          sum=sum+i;
          i++;
  }
  ```

- An empty *for* loop with just two semicolons is executed infinite number of times. Then the program has to be terminated by using ***ctrl – break*** keys.

```
for(; ;) ;
```

- More than one value can be initialized and updated in a *for* statement. And they have to be separated by comma.

```
for(i=1, j=10; i<10; i++, j--)
        printf("%d \t %d \n", i, j);
```

Here, the value ***i*** is initialized to 1 and incremented in every iteration. Whereas, the value ***j*** is initialized to 10 and decremented in every iteration. Hence, the output would be

```
1    10
2    9
3    8
4    7
5    6
6    5
7    3
8    2
9    1
```

1.9.4 Nesting of Loops

It is possible to have one loop within another loop. Even it is possible to have several types of loops one within another. This is known as nesting of loops. Consider the following example which prints the ordered pairs from (0,0) to (2,2).

```
int i,j;
for(i=0;i<=2;i++)
{
        j=0;
        while(j<=2)
        {
                printf("(%d, %d)", i, j);
                j++;
        }
}
```

The output of above code segment would be –

```
(0,0)        (0,1)        (0,2)
(1,0)        (1,1)        (1,2)
(2,0)        (2,1)        (2,2)
```

In the above example, the *for* loop is encountered first. After initialization of variable i and checking the condition i<=2, the control enters the loop. Now, j is initialized and the

condition of *while* loop j<=2 is checked. Then the ordered pair (0,0) is printed and j is incremented. Again the condition of *while* is checked and this process continued till j becomes 3. Note that value of i is not yet incremented. Once the program control comes out of *while* loop, value of i is incremented and condition of *for* loop is checked. Now, again j starts with 0 and *while* loop terminates when j reaches 3. This process is continued till i become 3. By the time, all the ordered pairs as shown above would have been printed.

Note that, unless the inner loop terminates, the next iteration of outer loop will not start. And for every iteration of outer loop, there will be several iterations of inner loop.

1.9.5 The *break* Statement

The program control will come out of any looping structure either normally or abnormally.

Normal Exit: Transferring the program control out of a loop when its condition is false, after executing the statement block repeatedly is known as normal exit or normal termination of the loop.
Abnormal Exit: In some situations, it is necessary to pass the control out of the loop based on some condition and not completing the required number of iterations. This is known as abnormal termination of a loop.

Abnormal exit from any loop at any point can be achieved by using *break* statement within a conditional statement. Consider the following code segment as an example:

```
int x, sum=0,i;

for(i=1;i<=10;i++)
{
      printf(”Enter a number:”);
      scanf(“%d”, &x);

      sum+=x;
      printf(“Sum = %d“,sum);
      if(sum>100)
            break;
}
printf(”Over!”);
```

A sample output:

```
Enter a number: 45
Sum = 45
Enter a number: 31
Sum = 76
Enter a number: 34
Sum = 110
Over!
```

In this example, in every iteration, the user is asked to input a number and it is added to sum. Depending on the values entered, the loop will continue for maximum of 10 iterations. But, if the sum becomes more than 100 in any iteration, the loop gets terminated and control will be passed out of the loop. In a sample output shown here, after 3 iterations, the loop has been terminated as the sum crossed 100.

Note that if *break* statement is used in nested loops, only inner loop is terminated.

1.9.6 The *continue* Statement

In some of the programs, few statements within a body of a loop have to be skipped for a particular iteration and the next iteration should be started. When such statements are at the end of the statement block, they can be skipped using *continue* statement followed by a conditional statement. Then the statements after *continue* are bypassed and the control is passed to the beginning of the loop for next iteration. For example:

```
int i,sum=0;
for(i=1;i<=10;i++)
{
      if(i%3 == 0)
            continue;

      printf("%d", i);
}
```

A sample output would be –

    1   2   4   5   7   8   10

The above code segment will print all the integers from 1 to 10 which are not divisible by 3. Inside *for* loop, the condition is checked whether i is divisible by 3 or not. If it is divisible, *continue* statement is used to take back the control to for loop and i get incremented. Note that, when i is divisible by 3, the statement

```
printf("%d", i);
```

is not executed.

1.10 Programming Examples

Various programming examples are given here based on the concepts of topics discussed in this chapter. However, the programs given here are just indicative and students are advised to workout more questions. Also, every problem may have different ways of solving it. And the students are advised to write the programs using their own logic, to get the expected output.

1. Write a C program to find area of a circle given the radius.

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

```
void main()
{
        float rad, area;
        float PI=3.1416;
        clrscr();

        printf("Enter the value of radius:");
        scanf("%f",&rad);
        area=PI*rad*rad;
        printf("\nThe area of a circle with radius %5.2f is %5.2f",rad,area);
        getch();
}
```

2. **Write a C program to compute simple interest and compound interest given the Principal amount (p), Time in years (t) and Rate of interest(r).**

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

void main()
{
        float P,T,R,SI,CI;
        clrscr();

        printf("Enter Principal amount:");
        scanf("%f",&P);
        printf("\nEnter Time duration:");
        scanf("%f",&T);
        printf("\nEnter Rate of interest:");
        scanf("%f",&R);

        SI=(P*T*R)/100;
        printf("\nThe Simple interest is %9.2f",SI);

        CI=P*pow(1+R/100,T)-P;
        printf("\nThe compound interest is %9.2f",CI);
        getch();
    }
```

3. **Write a C program to check whether a given number is even or odd using bitwise operator.**

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

```
#include<conio.h>

void main()
{
        int num,mask,result;
        clrscr();

        mask=1;

        printf("Enter a number :");
        scanf("%d",&num);

        result=num&mask;

        if(result==0)
                printf("\n%d is even",num);
        else
                printf("\n%d is odd",num);

        getch();
}
```

4. Write a C program to find biggest among three numbers using ternary operator.

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

void main()
{
        int a,b,c,big;
        clrscr();

        printf("Enter three numbers:\n");
        scanf("%d%d%d",&a,&b,&c);

        big=(a>b)?((b>c)?b:c):((b>c)?b:c);
        printf("\bBiggest is %d",big);

        getch();
}
```

5. Write a C program to find area of a triangle given three sides.

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

```
#include<math.h>

void main()
{
	float a,b,c,big,small1,small2;
	float s,area;
	clrscr();

	printf("Enter three sides:\n");
	scanf("%f%f%f",&a,&b,&c);

	big=a;
	small1=b;
	small2=c;

	if(b>big)
	{
		big=b;
		small1=c;
		small2=a;
	}
	if(c>big)
	{
		big=c;
		small1=a;
		small2=b;
	}

	if(big>=small1+small2)
	{
		printf("\nTriangle cannot be formed");
		getch();
		exit(0);
	}
	else
	{
		s=(a+b+c)/2;
		area=sqrt(s*(s-a)*(s-b)*(s-c));
		printf("\nArea of triangle is %5.2f",area);
		getch();
	}
}
```

6. Write a C program to find roots of a quadratic equation.

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

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

void main()
{
        float a, b, c, desc,r1,r2,real,img;
        clrscr();

        printf("Enter the values of a,b,c:\n");
        scanf("%f%f%f",&a,&b,&c);

        if(a==0)
        {
                printf("\nEquation is linear and has only one root. ");
                printf("\nAnd the root is %5.2f", -c/b);
                getch();
        }
        else
        {
                desc=b*b-4*a*c;
                if(desc==0)
                {
                        r1= -b/(2*a);
                        r2= -b/(2*a);
                        printf("\nThe roots are real and equal");
                        printf("\n r1 = %5.2f \t r2 = %5.2f",r1,r2);
                        getch();
                }
                else if(desc>0)
                {
                        r1=(-b+sqrt(desc))/(2*a);
                        r2=(-b-sqrt(desc))/(2*a);
                        printf("\nThe roots are real and distinct");
                        printf("\n r1=%5.2f\t r2=%5.2f",r1,r2);
                        getch();
                }
                else
                {
                        real=-b/(2*a);
                        img=sqrt(-desc)/(2*a);
                        printf("\nThe roots are imaginary");
                        printf("\n r1 = %5.2f + %5.2f i  \n",real,img);
                        printf(" r2 = %5.2f - %5.2f i",real,img);
                        getch();
                }
        }
```

```
}
```

7. Write a C program using *switch* statement to simulate a simple calculator that performs arithmetic operations.

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

void main()
{
        float a, b, result;
        char op;
        clrscr();

        printf("Enter two numbers:");
        scanf("%f%f",&a,&b);
        fflush(stdin);
        printf("\nEnter the operator(+,-,*,/):");
        scanf("%c",&op);

        switch(op)
        {
                case '+': result=a+b;
                          break;
                case '-': result=a-b;
                          break;
                case '*': result=a*b;
                          break;
                case '/': if(b==0)
                          {
                                printf("Division by Zero!");
                                getch();
                          }
                          else
                                result=a/b;

                          break;
                default : printf("Invalid Operator!");
                          getch();
        }
        printf("The result is %5.2f",result);
        getch();
}
```

8. Write a C program to find sum and average of first *n* natural numbers.

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

void main()
{
        int n, sum=0,i;
        float avg;
        clrscr();

        printf("Enter the value of n:");
        scanf("%d",&n);

        for(i=1;i<=n;i++)
                sum=sum+i;

        avg=(float)sum/n;
        printf("The sum of first %d natural numbers = %d",n,sum);
        printf("\nAnd the average is %5.2f", avg);
        getch();
    }
```

9. Write a C program to find the factorial of a given number.

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

void main()
{
        int n,fact=1,i;
        clrscr();

        printf("Enter any positive number:");
        scanf("%d",&n);

        if(n<0)
                printf("Factorial of a negative number can't be generated!");
        else
        {
                for(i=1;i<=n;i++)
                        fact=fact*i;

                printf("%d!=%d",n,fact);
        }
        getch();
}
```

10. Write a C program to generate Fibonacci sequence up to *n.*

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

void main()
{
	int n,i,fib1,fib2,fib3;
	clrscr();

	fib1=0;
	fib2=1;

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

	if(n<=0)
		printf("\nEnter a positive number");
	else
	{
		printf("\nThe sequence is:\n");

		if(n==1)
			printf("%d",fib1);
		else if(n==2)
			printf("%d\t%d",fib1,fib2);
		else
		{
			printf("%d\t%d",fib1,fib2);
			for(i=3;i<=n;i++)
			{
				fib3=fib1+fib2;
				printf("\t%d",fib3);
				fib1=fib2;
				fib2=fib3;
			}
		}
	}
	getch();
}
```

11. Write a C program to find GCD and LCM of two numbers.

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

```
void main()
{
        int a,b,m,n,gcd,lcm;
        clrscr();

        printf("Enter two numbers:");
        scanf("%d%d",&a,&b);

        if(a==0||b==0)
                printf("\nInvalid input!!");
        else
        {
                m=a;
                n=b;
                while(m!=n)
                {
                        if(n>m)
                                n=n-m;
                        else
                                m=m-n;
                }
                gcd=m;
                lcm=(a*b)/gcd;
                printf("\nGCD of %d and %d is %d",a,b,gcd);
                printf("\nLCM of %d and %d is %d",a,b,lcm);
        }
        getch();
}
```

12. Write a C program to generate prime number up to the given number *n*.

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

void main()
{
        int i, n,j,flag,limit;
        clrscr();
        printf("Enter n:");
        scanf("%d",&n);

        for(i=1;i<=n;i++)
        {
                flag=1;
```

```
            for(j=2;j<i;j++)
            {
                if(i%j==0)
                {
                    flag=0;
                    break;
                }
            }
            if(flag==1)
                printf("\n%d",i);
        }
        getch();
}
```

13. Write a C program to reverse a given number and to check whether it is a palindrome.

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

void main()
{
    int num,revnum=0,n,dig;
    clrscr();

    printf("Enter a number:");
    scanf("%d",&num);
    n=num;
    while(n!=0)
    {
        dig=n%10;
        revnum=revnum*10+dig;
        n=n/10;
    }

    printf("\nThe reverse of %d is %d",num,revnum);

    if(num==revnum)
        printf("\nThe number is palindrome");
    else
        printf("\nThe number is not a palindrome");

    getch();
}
```

14. Write a C program to find the sum of all the digits of a given number.

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

void main()
{
        int num,sum=0,n,dig;
        clrscr();

        printf("Enter a number:");
        scanf("%d",&num);
        n=num;
        while(n!=0)
        {
                dig=n%10;
                sum=sum+dig;
                n=n/10;
        }

        printf("\nThe sum of digits of %d is %d",num,sum);

        getch();
}
```

15. Write a C program find the value of *sin(x)* using the series –

$$sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

up to n terms. Also print the value of *sin(x)* using standard library function.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define PI 3.1416

void main()
{
        int n,i;
        float x, theta,sum=0,term;
        clrscr();

        printf("Enter x:");
        scanf("%f",&x);
        printf("\nEnter number of terms in series:");
        scanf("%d",&n);

        theta = x * PI/180;
```

```
        term=theta;

        for(i=1;i<=n;i++)
        {
                sum=sum+ term;
                term=-term* theta*theta/(2*i*(2*i+1));
        }

        printf("\nsin(%f)=%f",x,sum);
        printf("\nUsing library function, sin(%f)=%f",x,sin(theta));
        getch();
}
```

16. Write a C program find the value of *cos(x)* using the series –
$cos(x)= 1 - x^2/2! + x^4/4! - x^6/6! + \ldots$
up to n terms. Also print the value of *cos(x)* using standard library function.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define PI 3.1416

void main()
{
        int n,i;
        float x, theta,sum=0,term;
        clrscr();

        printf("Enter x:");
        scanf("%f",&x);
        printf("\nEnter number of terms in series:");
        scanf("%d",&n);

        theta = x * PI/180;
        term=1;

        for(i=1;i<=n;i++)
        {
                sum=sum+ term;
                term=-term* theta*theta/(2*i*(2*i-1));
        }

        printf("\ncos(%f)=%f",x,sum);
        printf("\nUsing library function, cos(%f)=%f",x,cos(theta));
        getch();
}
```

User-defined Functions

1.11 INTRODUCTION

Transforming a larger application into a single problem leads t o several complications such as inefficient memory management, difficulty in the development of source code, debugging etc. Modularization of a program will avoid these complications. Here, the original problem is divided into several sub-problems and solved. Then the solutions of all of these sub-problems are combined to get the solution for original problem.

A sub-program is a set of instructions that can be executed under the direction of another program to get a solution of a sub-problem. Usually, subprograms are written for smaller and independent tasks that appear frequently in larger problems. In C/C++, the programmer has a concept of ***functions*** for modularization. In fact, main() is also a function from where the execution starts in every program and it is a compulsory function in any C/C++ program. A subprogram contains type declarations, statements etc. and it can not be executed by itself. It can be executed only through another subprogram. A subprogram or main() that calls another subprogram is known as ***calling function*** and the activated subprogram is known as ***called function***.

A function may be in-built or user-defined. An in-built or library function is a subprogram stored in the library of compiler and is readily available to all the programs. For example, *pow(), sqrt(), clrscr()* etc. User-defined functions are the functions developed by the programmer to solve a particular task. A calling function may supply some values known as ***arguments/actual parameters*** to a called function and the called function receives these values through a set of variables known as ***formal parameters***.

1.11.1 Advantages of Functions

The usage of functions in a program has following advantages:

- **Reduced Code Size:** Some times, the programmer has to perform a particular task repeatedly in a program. For example, one has to find factorial of a number, square root of a number, etc. at several stages in a program. Then, instead of writing statements repeatedly for these tasks, it is better to write a function. Whenever a task has to be performed, then the function may be called and thus reducing size of the code.
- **Modular Programming Approach:** Each function performing one independent task can be considered as a module. Such type of modular programming will increase readability of the program and helps in solving relatively bigger problems.
- **Easier Debugging:** A program divided into modules is always easier to debug as each function is consisting of only few set of statements.
- **Reduced Memory Size:** Every executable statement in a program requires certain amount of space in a computer memory. So, the reduction in code-size of a program will lead to reduction in memory usage too.

- **Division of Work:** A big task may need to be handled by a group of people. Then, with the help of modular programming technique, the problem may be divided and each sub-problem may be given to each individual. Thus, every person in a group will have only a small task to solve. These solutions can then be combined to get the solution of original problem. This technique will save the time.
- **Reusability of Code:** The functions written for general purpose tasks like sorting, searching, finding square root etc. can be used by other programs, whenever required. Thus, the code written once can be re-used in several applications.

1.11.2 General form of Functions

Functions that are the building blocks of C/C++ can be thought of as a user-defined operation. The general form of a function is:

```
ret_type fname(para_list)
{
      //local-variable declarations
      // statements
      return expr;
}
```

Here, ret_type	is the type of the data returned by the function
fname	is any valid name given to the function
para_list	is a list of variable names and their associated data types that receive the values of the arguments when the function is called. There may be zero or more parameters.
return	is a keyword used to return any value to the calling function.

When a function call occurs, the execution of currently active function is suspended temporarily and the program control is passed to the called function. When the execution and/or evaluations of the called function are completed, the program control returns to the calling function and the calling function resumes the execution at the point immediately following the function call.

While discussing with functions, three important concepts to be understood are: prototype of the function, argument passing technique and return value of a function. These concepts are discussed here.

1.11.3 Function Prototype

A function must be declared to the program before it is called. This function declaration consists of a single line

return_type fname(para-list);

This is known as a ***function prototype***. However, the function prototype along with function body, which is known as ***function definition***, can also serve as its declaration.

The return_type of a function can be a basic data type like int, float etc. or a compound type such as float* etc, or it may be a user defined type like enumeration, structure, union etc. A function return type may be void indicating that the function is returning nothing. The parameter list may contain zero or more parameters of different/same data types.

1.11.4 Argument Passing

Functions are allocated storage on the program's runtime stack for their use. That storage remains with the function till the function terminates. After the termination of the function, this storage memory will be available for reuse. This storage area of the function is known as ***activation record.*** Each function parameter is given memory from this activation records according to its data type.

Argument passing is the process of initializing storage of function parameters with the values of function call arguments. There are three different ways of passing arguments to a function viz.

- **Call-by-value**
- **Call-by-address (or call by reference)**

Each of these methods is discussed later in this chapter.

1.11.5 Returning a value

The return statement is placed within the function body. It has two important uses:

- It causes immediate exit from the function that it is in. That is, it causes the program control to return to the calling function.
- It can be used to return a value to the calling function from the called function.

There are two different forms of return statement:

- return; This will simply return to the calling function. Whenever it is not necessary to return any value from the called function to the calling function, the return statement is optional. Because after the last executable statement in the program, by encountering the closing bracket (the flower-bracket **}**), it will go back to the calling function. But, it is a good programming practice to use the return statement at the end of the function.
- return expr; This will return the value of expr to the calling function. In this case the called function is used at the right-hand-side of the assignment operator in the calling function, so that the returned value of expr is stored in the variable that is at the left-hand-side of the assignment operator.

Note

1. The value of the returning *expression* may be of any basic data type like int, float, double, char etc. Moreover it can be an address i.e. a pointer to any of these data types or any derived data types like arrays, structures, unions or any of the user defined data types.
2. To indicate that the function is not returning anything, void can be used as the return type for function declaration.

main() is known to be is the compulsory function in any C program. This function returns an integer to the calling process i.e., the operating system. Returning a value from the main() is equivalent to calling the exit() function with the same value.

1.12 CATEGORIES OF FUNCTIONS

Though the general syntax of a function includes return type, parameter list and return statement, these are optional. That is a function may or may not contain these parts. Based on this aspect, one can have following four formats of functions:

- **Function without parameters and returning nothing**
- **Function with parameters and returning nothing**
- **Function without parameters but returning a value**
- **Function with parameters and returning a value**

These formats are discussed as here-under:

- **Function without parameters and returning nothing:** A function may not receive any parameter from and also it may not return any value to calling function. Consider the following example:

```
//A Function without parameters returning nothing
#include<stdio.h>

void sum()
{
        int a, b, c;
        printf("Enter Two values:");
        scanf("%d%d",&a, &b);
        c=a+b;
        printf("\nSum is : %d", c);
}

void main()
{
        sum();
}
```

In this example, the main() function is used just to call the function sum().

- **Function with parameters and returning nothing:** A function may take parameters from the calling function but may not return any value to it. For example:

//A Function with parameters returning nothing

```
#include<stdio.h>
void sum(int a, int b)
{
        int c;
        c=a+b;
        printf(”\nSum is : %d”, c);
}

void main()
{
        int x, y;
        printf(”Enter Two values:”);
        scanf("%d%d" , &x, &y);
        sum(x, y);
}
```

- **Function without parameters but returning a value:** Sometimes, a function may not receive any parameter from the calling function. But it may send some value. For example:

//A Function without parameters returning a value

```
#include<stdio.h>
int sum()
{
        int a,b,c;
        printf(“Enter Two values:”);
        scanf(“%d%d”, &a, &b);
        c=a+b;
        return c;
}
void main()
{
        int x;
        x=sum();
          printf(“Sum is : %d”, x);
}
```

- **Function with parameters and returning a value:** A function receiving parameters from the calling function and returning a value to it is used for general-purpose tasks.

//A Function with parameters returning a value

```
#include<stdio.h>
int sum(int a, int b)
{
        return (a+b);
}
void main()
{
        int x, y, z;
        printf(”Enter Two values:”);
        scanf("%d%d", &x, &y);
        z=sum(x,y);
        printf(“Sum is %d”, z);
}
```

1.12.1 Call-by-value Method

The formal parameters behave like local variables of that function. Thus, in the default initialization method of argument passing, where the values of the arguments are just copied into formal parameters, the change in formal parameters does not reflect the arguments. This is known as ***call-by-value.*** To illustrate this method, consider a program:

Program for addition of two numbers

```
#include<stdio.h>
int sum(int, int);      //function prototype or declaration
void main()
{
        int a, b, c;

        printf(“Enter two values:”);
        scanf(“%d%d”, &a, &b);
        c=sum(a, b);            //function call
        printf(”\nSum is %d “, c);
}

int sum(int x, int y)           //function definition
{
        int z ;
        z= x+y ;
        return z ;                      //returning the result
}
```

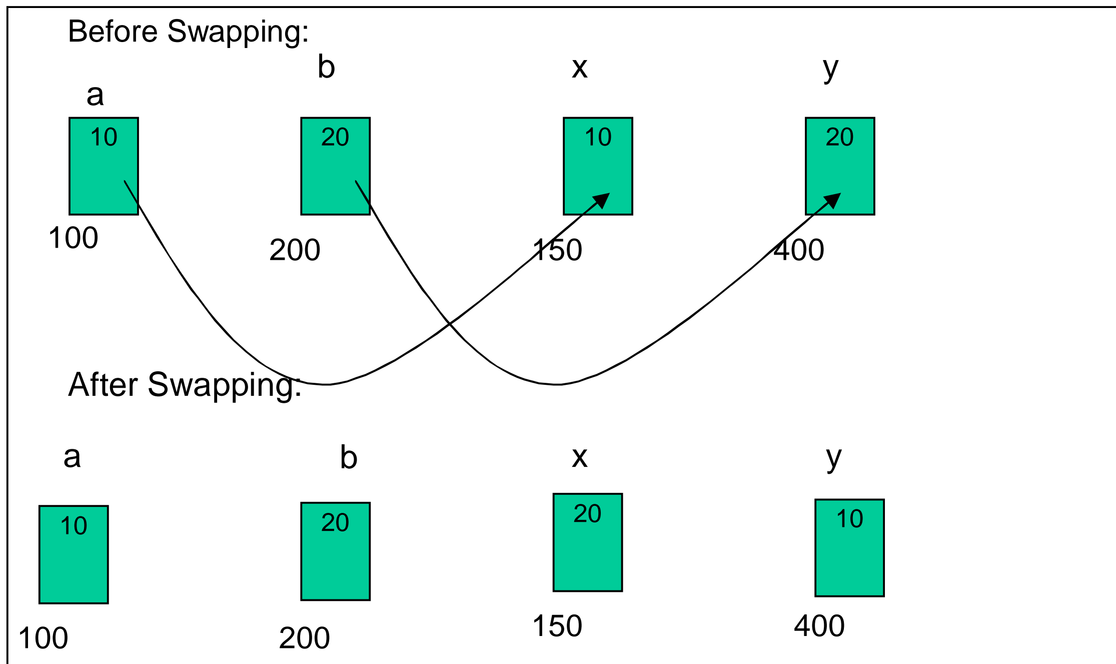
The output may be:

```
Enter two values: 4  8
Sum is 12
```

Since the values of the arguments are copied into formal parameters, they will act as a local copy of the function. As the formal parameters use run-time-stack memory and are destroyed upon the exit of the function, the changes or the manipulations in formal parameters will not affect the actual arguments. Thus when we want to modify the arguments, the call-by-value method is not suitable. Moreover, when a large list of variables must be passed to a function, the time and space costs more to allocate them a memory space in the stack. So, it is not suitable for real world application. To understand this concept, consider following example:

```
//Program that uses call-by-value method
#include<stdio.h>
void swapv(int, int);

void main()
{
	int a = 10, b=25;
	printf("Before Swapping:\n");
	printf("%d \t %d", a, b);
	swapv (a,b);
	printf("\n After swappv() function call :");
	printf("%d \t %d", a, b);
}
swapv(int x, int y)
{
	int temp;
	temp=x;
	x= y;
	y= temp;
	printf("Within function:");
	printf("%d \t %d", x, y);
}
```

In the preceding example, the function **main()** declared and initialized two integers **a** and **b**, and then invoked the function **swapv()** by passing **a** and **b** as arguments to the function **swapv()**. The function **swapv()** receives the arguments **a** and **b** into its parameters **x** and **y**. In fact, the function **swapv()** receives only a copy of the values of **a** and **b** into its parameters. Now, the values of **x** and **y** are swapped. But, the original values at **a** and **b** remains same. This can better be understood using the memory maps of variables as shown in the following figure.



Thus, the modification made to formal parameters within the function will not affect the original arguments, in case of *call-by-value* method. If we need the modification to get affected, we should use *call-by-address/reference* which is discussed in next section.

1.12.2 Call-by-Address (or Call by Reference) Method

We have discussed the *call-by-value* method of passing parameters to functions in Section 4.4 and the inconveniences are discussed. To overcome these problems, ***call-by-address*** technique is preferred. Here, instead of the values of actual arguments, their addresses are passed to formal parameter pointers. That is, in this case, the formal parameters must be pointers of specified type, but not the ordinary variables. In this case, the manipulation made inside the function will be stored in the memory affecting the actual arguments.

Program that uses call by address method to swap two variables

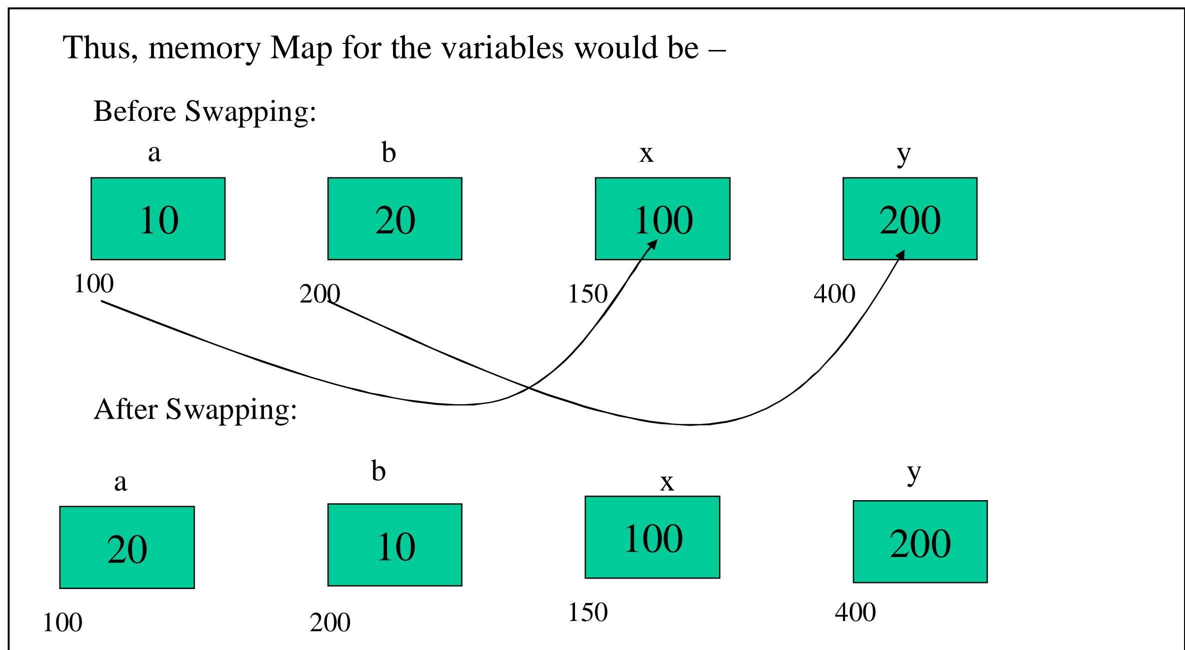
```
#include<stdio.h>

void swapr(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}

void main()
{
        int a =10, b=20;
```

```
        printf("Before swapping: a=%d   b=%d", a, b);
        swapr(&a, &b);        //addresses of a and b are passed
        printf("\nAfter swapping:  a=%d   b=%d", a, b);
}
```

The output would be:

```
a=10  b=20
a=20  b=10
```

In the above example, the function **main()** declared and initialized two integers **a** and **b**, and then invoked the function **swapr()** by passing the addresses of **a** and **b** as arguments to the function **swapr()**. The function **swapr()** receives these addresses into its pointer parameters **x** and **y**. Now, the values stored at the addresses **x** and **y** are swapped. Thus, the actual arguments have been modified. This can better be understood using the memory maps of variables as shown in following figure.



ARRAYS and STRINGS

1.13 INTRODUCTION

When there is a need for single data, one can declare a variable for that element and solve the problem. Suppose the marks of a student in six subjects needs to be stored. Still one can make use of six different variables. Consider a situation of storing the average marks of all the students in a class, say 60 students. Now, making use of 60 different variables is absurd. The best way to solve such kind of problems is to make use of arrays.

As observed in the above requirements, we may need to have a single variable name that may take multiple values and all are related to each other. These values will be of same data type. Therefore, ***array can be defined as the collection of related data elements stored at the contiguous memory locations.***

Depending on the arrangement of data elements, an array can be classified as one-dimensional array, two-dimensional array etc.

1.13.1 One Dimensional Arrays

An array in which the elements are arranged in the form of a row is known as one dimensional array or ***vector.*** For example, marks obtained by a student in 6 subjects is one dimensional array, which may be represented by a subscripted variable *marks*. Syntax for declaring one dimensional array is –

```
data_type array_name[Size];
```

Here,

data_type	is any inbuilt/user-defined data type
array_name	is any valid variable name.
Size	is an integer constant representing total number of elements in an array.

Examples:

1. int marks[6];
   marks is an array that can hold *6 elements* of *integer* type.

2. float avg_score[60];
   avg_score is an array that is capable of storing *60 elements* of *float* type.

All the elements of an array will be stored in contiguous memory locations. The total memory allocated for an array is

Size_of_array * size_of_data_type

For example, assuming that size of integer variable is 2 bytes and that of float is 4 bytes, the total memory allocated for the array *marks* in the above example is

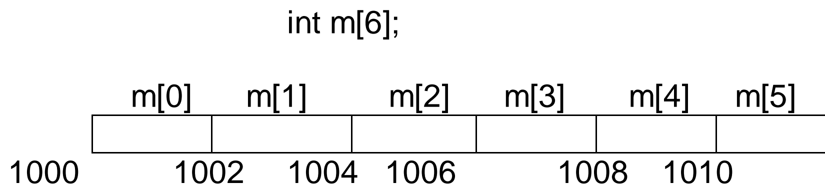
6*2 =12 bytes

The size of *avg_score* is

60*4 = 240 bytes

Each element can be accessed using an integer variable known as *array index*. The ***value of array index ranges from 0 to size-1***. Consider the memory representation for the array declared as –

int m[6];



Thus, the first element of the array is *m[0],*
second element of the array is *m[1], ……*
last element of the array is *m[5]* (i.e. size-1)

One can observe that all the elements of the array have been allocated memory in contiguous locations. The address of first element of the array is known as ***base address*** of the array. In the above example, base address of the array m[] is 1000.

Note that the address of the element arr[i] in an array arr can be obtaining using the formula –

Address of ith element = base address + i * size_of_data_type

In the above example, the array m[] is of integer type. The size of an integer is 2 bytes and the base address of the array is assumed to be 1000.

Thus, the address of the element, m[3] = 1000 + 3 * 2
= 1006

1.13.2 Initialization of One Dimensional Array

When an array is declared, it will contain garbage values for all its elements. One can initialize the elements of the array at the time of its declaration itself. The syntax for array initialization is –

data_type arr_name[size]={v_1, v_2, …, v_n};

Here, v_1, v_2,…,v_n are constants or expressions yielding constant values of specified data_type.

Consider the following examples:

- int m[6] = { 80, 58, 45, 64, 68, 98};
  Now, m[0] is 80,
  m[1] is 58 and so on

  The memory map would be –

m[0]	m[1]	m[2]	m[3]	m[4]	m[5]
80	58	45	64	68	98
1000	1002	1004	1006	1008	1010

- int m[]={75, 89, 41, 54, 90};

  When the array is initialized at the time of declaration, the size of the array need not be specified. In this example, the size of the array is 5 as the value-list contains 5 values.

- int m[5]={75, 89, 41};
  When sufficient numbers of elements are not provided as per the specified size of the array, then the last elements will take the value as 0. In this example,
  m[0]=75 m[1]=89 m[2]=41 m[3]=0 m[4]=0

1.13.3 Assigning and Accessing 1-D Array Elements

When an array is not initialized at the time of declaration, one can give the values later using the index of an array. For example,

```
int marks[6];          // Declaration of the array

marks [0] = 80;
marks [1] = 58;                 and so on.
```

If the values are to be read from the keyboard, then a loop can be used to read array elements as shown –

```
int marks[6], i;       // Declaration of the array

for(i=0;i<6;i++)                    //i ranges from 0 to 5 (size-1)
        scanf("%d",&marks[i]);      //read marks[i] for each i
```

The elements of an array can be accessed in the similar manner using array index as shown –

```
for(i=0;i<6;i++)                    //i ranges from 0 to 5 (size-1)
        printf("%d", marks[i]);     //print marks[i] for each i
```

1.13.4 Programming Examples on 1-D Array

In this section let us see some of the examples on one-dimensional arrays.

1. Write a program to read any *n* numbers and to compute average and standard deviation of those numbers.

If a[0],[1],……,a[n-1] are the elements of an array with size n, then the average is –

$$Avg = \frac{1}{n}\sum_{i=0}^{n-1} a[i]$$

The standard deviation is –

$$SD = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(a[i] * a[i]) - Avg * Avg}$$

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

void main()
{
        int a[10],i,n;
        float sum=0,sumsq=0,avg,sd;
        clrscr();

        printf("Enter size of the array:");
        scanf("%d",&n);
        printf("\nEnter array elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        for(i=0;i<n;i++)
        {
                sum=sum+a[i];
                sumsq=sumsq+ a[i]*a[i];
        }

        avg=sum/n;
        sd=sqrt(sumsq/n-avg*avg);

        printf("\nAverage=%5.2f",avg);
        printf("\nStandard Deviation= %5.2f",sd);
}
```

2. Write a program to display Fibonacci series up to n.

The fibonacci sequence is defined as –

F(0) = 0
F(1) = 1
F(n) = F(n-1)+F(n-2), for all n>=2

Thus, the sequence is 0, 1, 1, 2, 3, 5, 8,.........

```
#include<iostream.h>
#include<conio.h>
void main()
{
	int fibo[20], i,n;
	clrscr();

	printf("Enter number of elements in a series:");
	scanf("%d", &n);
	fibo[0]=0;
	fibo[1]=1;
	for(i=2;i<n;i++)
		fibo[i]=fibo[i-1]+fibo[i-2];

	printf("\n The Fibonacci series:\n");
	for(i=0;i<n;i++)
		printf("%d \t",fibo[i]);
}
```

A sample output would be –

```
	Enter number of elements in a series: 10
	The Fibonacci series:
    0 1    1    2    3    5    8    13    21    34
```

3. Write a C program to sort a list of elements using bubble sort technique.

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

void main()
{
	int a[10],i,n,j,temp;
	clrscr();

	printf("Enter array size:");
	scanf("%d",&n);
	printf("\nEnter array elements:\n");
```

```
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        for(i=1;i<n;i++)
        {
                for(j=0;j<n-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
        }

        printf("\nSorted List:\n");
        for(i=0;i<n;i++)
                printf("%d\n",a[i]);
}
```

4. Write a C program to search for a key element in a given list using linear (sequential) search.

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

void main()
{
        int a[10],i,n,flag=0,key;
        clrscr();

        printf("Enter array size:");
        scanf("%d",&n);
        printf("\nEnter array elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);

        printf("\nEnter key element to be searched:");
        scanf("%d",&key);

        for(i=0;i<n;i++)
        {
                if(a[i]==key)
                {
```

```
                    flag=1;
                    break;
            }
    }

    if(flag==1)
            printf("Key is found at the position %d",i+1);
    else
            printf("Key is not found");
}
```

5. **Write a C program to search for a key element in a given list using binary search.**

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

void main()
{
    int a[10],n,i,mid,low,high,key,flag=0;
    clrscr();

    printf("Enter array size:");
    scanf("%d",&n);
    printf("\nEnter array elemnts in ascending order:\n");
    for(i=0;i<n;i++)
            scanf("%d",&a[i]);

    printf("\nEnter key to be searched:");
    scanf("%d",&key);

    low=0;
    high=n-1;

    while(low<=high)
    {
            for(i=0;i<n;i++)
            {
                    mid=(low+high)/2;
                    if(a[mid]==key)
                    {
                            flag=1;
                            break;
                    }
                    else if(a[mid]>key)
```

```
                high=mid-1;
            else
                low=mid+1;
        }
        break;
    }

    if(flag==0)
        printf("\nKey not found");
    else
        printf("\nKey found at %d",mid+1);

    getch();
}
```

1.14 Multidimensional Arrays

In the previous section, we have discussed one dimensional array. Based on the arrangement of data, arrays can be classified as two dimensional, three dimensional etc. These are known as multidimensional arrays.

1.14.1 Two dimensional Arrays

Elements arranged in a row and column order is known as two dimensional array. For example, marks obtained by 3 students in 3 different subjects is a two dimensional array and it can be written in a row-column format as –

	Sub-0	**Sub-1**
Stud-0	75	85
Stud-1	54	62
Stud-2	92	88

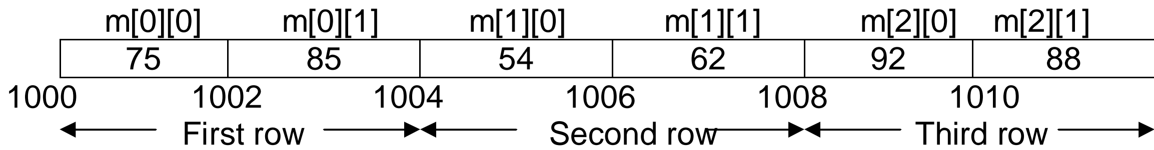
Since two dimensional array is spread into two directions, we need to specify two indices, one to represent row and the other for column. For example-

```
int m[3][2];
```

Here, the first index 3 indicates the number of rows and second index is to indicate column. The first index (say, i) ranges from 0 to 2 and second index (say, j) takes the values 0 and 1 for each value of first index. Thus, the array elements may be given as –

i	j	Array Elements
0	0	m[0][0]
	1	m[0][1]
1	0	m[1][0]
	1	m[1][1]
2	0	m[2][0]
	1	m[2][1]

The memory allocation for above 2-D array would look like –



NOTE:

1. The total memory allocated for any 2-D array can be given as –

Size_of_first_index * Size_of_second_index * Size_of_data_type

For example, the total size of the array,

float marks[5][3];

is

5*3*4 =60 bytes (size of float being 4 bytes)

2. Let the size of a 2-D array be m X n. That is, an array is containing m rows and n columns. Then the address of an element at i^{th} row and j^{th} column of 2-D array can be computed using the formula -

base address + (i * n + j) * size_of_data_type

For example, assume that the base address of the following array is 1000.

float marks[5][3];

Then, the address of the element marks[4][2] is

1000 + (4 * 3 + 2)*4 =1056

(Here, the array dimensions, m = 5 and n=3. Size of float being 4 bytes)

1.14.2 Three dimensional Arrays

Consider an example of marks obtained by 5 students in 2 tests in each of 3 subjects. This type of data can be given in a tabular format as –

	Subject-0		**Subject-1**		**Subject-2**	
	Test-0	**Test-1**	**Test-0**	**Test-1**	**Test-0**	**Test-1**
Stud-0	75	58	76	90	77	75
Stud-1	54	62	52	64	55	61
Stud-2	92	88	90	88	85	82
Stud-3	75	85	77	75	73	77
Stud-4	45	48	55	52	41	44

A 3-D array for the above example can be declared as –

int m[5][2][3];

The first index represents number of students
second index represents number of tests
third index represents number of subjects

The marks obtained by 3rd student in 2nd test in 2nd subject can be accessed as –
m[2][1][1]

Discussion of 3-D and higher dimensional arrays is out of the scope of this book. In the following sections, 2-D array is discussed more in detail.

1.14.3 Initialization of 2-D Array

At the time of declaration, a 2-D array can be initialized just like 1-D array. Consider the following examples that illustrate various ways and their meaning of initializing 2-D array.

- int m[3][2] = {{56,78},{67,61},{90,85}};
  Now, m[0][0]=56, m[0][1]=78, m[1][0]=67 and so on

- int m[3][2] = {56,78,67,61,90,85};
  Now also, m[0][0]=56, m[0][1]=78, m[1][0]=67 and so on
  That is, when sufficient numbers of elements are given at the time of initialization, the inner braces are not necessary.

- If sufficient numbers of elements are not given and inner braces are removed, then the elements at the end will take the value as zero. That is,

  ```
  int m[3][2] = {56,78,67,61};
  ```

  Now, m[0][0], m[0][1], m[1][0], m[1][1] will take the values 56,78, 67 and 61 respectively. But, m[2][0] and m[2][1] will be zero.

- If sufficient numbers of elements are not given within inner braces, then the corresponding elements will be zero. For example –

  ```
  int m[3][4] = {{56,78},{67,61,85},{75}};
  ```

  is equivalent to –

  ```
  int m[3][4] = {{56, 78, 0, 0},
                 {67, 61, 85, 0},
                 {75, 0, 0, 0}
                };
  ```

- If sufficient numbers of elements are provided along with inner braces, then the first index of the array can be omitted by only specifying the second index. for example –

  ```
  int m[][2] = {{56,78},{67,61},{90,85}};
  ```

  Now, it is understood by the compiler that the number of rows is 3.

1.14.4 Assigning and Accessing 2-D Array Elements

Assigning values to the elements of 2-D are just similar to that of 1-D array. Reading the values from the keyboard for 2-D array and then printing those values is illustrated here under:

```
int m[5][4],i,j;            //declaring 2-D array

for(i=0;i<5;i++)            //first index ranges from 0 to 5-1=4
    for(j=0;j<4;j++)            //second index ranges from 0 to 4-1=3
        scanf("%d", &m[i][j]);        //read elements

for(i=0;i<5;i++)
    for(j=0;j<4;j++)
        printf("%d", m[i][j]);   //print elements
```

1.14.5 Programming Examples on 2-D Array

In this section, we will discuss some useful programs that requires 2-D array.

Write a program to find trace and norm of a given matrix.

Trace of a matrix is calculated only for a square matrix. It is a sum of all the elements on a principal diagonal of a square matrix. Norm of a matrix is the square root of the sum of squares of elements of a matrix. Consider the following matrix of order 3x3 –

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Now, trace of this matrix is = $a_{00} + a_{11} + a_{22}$

Norm of a matrix with the order m x n is = $\sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(a_{ij} * a_{ij})}$

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

void main()
{
	int a[5][5],m,n,i,j,trace=0;
	float norm, sumsq=0;
	clrscr();

	printf("Enter Number of rows and columns:");
	scanf("%d %d", &m, &n);
```

```
	if(m!=n)
	{
		printf("\nNot a sqare matrix!!!");
		printf("\nCant find trace. But you can try Norm.");
	}

	printf("\nEnter elements:\n");
	for(i=0;i<m;i++)
		for(j=0;j<n;j++)
			scanf(“%d”, &a[i][j]);

	if(m==n)
	{
		for(i=0;i<m;i++)
		{
			for(j=0;j<n;j++)
			{
				if(i==j)
					trace+=a[i][j];
			}
		}
		printf("\nTrace = ", trace);
	}

	for(i=0;i<m;i++)
		for(j=0;j<n;j++)
			sumsq=sumsq + a[i][j]*a[i][j];

	norm=sqrt(sumsq);
	printf("\nNorm = ", norm);

	getch();

}
```

Write a program to add two matrices.

```
#include<iostream.h>
#include<conio.h>

void main()
{
	int a[5][5],b[5][5],sum[5][5];
	int m,n,p,q,i,j;
```

```
        clrscr();

        printf("Enter order of first matrix:\n");
        scanf("%d%d", &m, &n);
        printf("Enter order of second matrix:\n");
        scanf("%d%d", &p, &q);

        if(m!=p || n!=q)
        {
                printf("Orders must be same. Addition not possible");
                getch();
                exit(0);
        }
        else
        {
                printf("\nEnter 1st matrix:\n");
                for(i=0;i<m;i++)
                        for(j=0;j<n;j++)
                                scanf("%d", &a[i][j]);

                printf("\nEnter 2nd matrix:\n");
                for(i=0;i<p;i++)
                        for(j=0;j<q;j++)
                                scanf("%d", &b[i][j]);

                for(i=0;i<m;i++)
                        for(j=0;j<n;j++)
                                sum[i][j]=a[i][j]+b[i][j];

                printf("\nThe addition of two matrices:\n");

                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                                printf("%d\t",sum[i][j]);
                        printf("\n");
                }
        }
        getch();
}
```

Write a program to find transpose of a given matrix.

Transpose of a matrix is changing of its rows into columns and columns into rows. For example,

$$A = \begin{pmatrix} 4 & 5 \\ -2 & 0 \\ 7 & 3 \end{pmatrix} \qquad \text{Then, } A' = \begin{pmatrix} 4 & -2 & 7 \\ 5 & 0 & 3 \end{pmatrix}$$

Thus, if a matrix is of order m x n, then its transpose will be of order n x m.

```
#include<iostream.h>
#include<conio.h>

void main()
{
	int a[5][5],trans[5][5],m,n,i,j;
	clrscr();

	printf("Enter the order of matrix:\n");
	scanf("%d%d", &m, &n);
	printf("\nEnter elements:\n");
	for(i=0;i<m;i++)
		for(j=0;j<n;j++)
			scanf("%d", &a[i][j]);

	for(i=0;i<n;i++)
		for(j=0;j<m;j++)
			trans[i][j]=a[j][i];

	printf("\nGiven matrix:\n");
	for(i=0;i<m;i++)
	{
		for(j=0;j<n;j++)
			printf("%d\t", a[i][j]);
		printf("\n");
	}

	printf("\nTranspose is:\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
			printf("%d\t", trans[i][j]);
		printf("\n");
	}
}
```

1.15 INTRODUCTION TO STRINGS

C does not provide a separate data type *string* unlike many other languages. But, an array of characters (char) can be treated as a string. The concepts of handling arrays such as array declaration, initialization and manipulation may be adopted to deal with character arrays/strings.

C provides a rich set of library functions for processing strings. A string contains one or more than one characters enclosed in double quotes. The C compiler automatically provides a *null character '\0'* at the end of a string, when it is stored as an array of characters. Though null character appears as a combination of two characters, it is actually a single character whose ASCII value is 0. This indicates the end of a string and hence very useful in tracing the characters of a string. Each character in an array occupies *one byte* of memory. So, a string with *n* characters has to be provided with *n+1* bytes of memory so as to accommodate the null character at the end.

Example:

A string "Hello" is stored in the memory as –

H	e	l	l	o	\0

1.15.1 Rules for constructing Strings

- A string constant is one or more characters enclosed in double quotes. The double quotation marks serving as *delimiters* are not the part of strings.
  Examples:
  "Problem Solving Using C"
  "35"
  "Hello" etc.

- Non-graphic characters such as \n, \t etc can be part of a character array. For example:
  "\n \n Student List \n \n"
  "Name \t USN"

- A string may be continued in the next row by placing a back slash at the end of the portion of the string in the present row. For example:
  "Bangalore is \
  the capital of Karnataka"
  is same as
  "Bangalore is the capital of Karnataka"

- If the characters such as double quotes, back slash, single quote etc. are to be part of a string, they should be preceded by a back slash. For example:
  " \"What a beautiful flower!!\" "
  will yield the output as
  "What a beautiful flower!!

“I Said \’Hello\’ “
will give the output as:
I said ‘Hello’

“Division \\ Multiplication”
will be resulting as
Division \ Multiplication

1.15.2 Declaration and Initialization of Strings

A string can be declared as –

char var_name[size];

An un-initialized string contains garbage values. A string can be initialized in different ways as shown –

- char s[10]=”Hello”;
  Now, the memory allocation for s may look like –

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	e	l	l	o	‘\0’				
100	101	102	103	104	105	106	107	108	109

  Though the value of the string (“Hello”) is only 5 characters long, the string s requires 6 bytes of memory. The last byte is used for null character, which indicates end of the string. If the allocated memory for the string is more than required, then the remaining bytes will be containing garbage values. In this example, the locations s[6],s[7],s[8] and s[9] are containing garbage values.

- String can be initialized through single characters using flower brackets as we do for numeric array. In such a situation, the null character must be specified explicitly. For example –

  char s[10]={‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

  The memory allocation is same as shown in the above example.

- If string variable is initialized with sufficient number of characters, then the size of the string is optional. For example –
  char s[]=”Hello”; **OR**
  char s[]={‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};

  In both the situations, the string size will be 6 bytes.

1.15.3 String Handling Functions

C provides several library functions for performing various operations on strings. These functions are included in the header file string.h. The following table lists out few of such functions.

String Handling functions

Function	Operation
strlen(str)	Returns number of characters in a string str (excluding null character)
strcpy(s1,s2)	Copies the content of s2 into s1
strcat(s1,s2)	Concatenates two strings s1 and s2. That is, contents of s2 are appended at the end of s1.
strcmp(s1,s2)	Compares two strings s1 and s2 character by character, starting from first position. The comparison is carried out till a mismatch is found or one of the strings gets exhausted, whichever is earlier. This function returns the difference between the ASCII values of first non-matching characters. If the returned value is 0, then the strings are equal. If the value is positive, then s1 is greater than s2. Otherwise, s2 is greater than s1.
strcmpi(s1,s2)	Compares two strings ignoring the case of the characters. That is, this function is case-insensitive.
strncpy(s1,s2,n)	Copies first n characters of s2 into s1, where n is an integer.
strncmp(s1,s2,n)	Compares at the most n characters in s1 and s2.

Consider the following examples –

- int len;
  char str[20]=”Hello”;
  len=strlen(str);	//len gets value as 5

- char s1[20],s2[]=”World”;
  strcpy(s1,s2);	//”World” is copied into s1

- char s1[20]=”Hello”, s2[]=”World;
  strcat(s1,s2);	//s1 becomes “HelloWorld”

- char s1[]=”Ramu”, s2[]=”Raju”;
  int n=strcmp(s1,s2);

  Now, the first non-matching characters are m and j. Difference between ASCII values of m and j is 109-106=3. Since a positive number is returned, s1 is greater than s2.

  strcmp(“cup”,”cap”);	//returns 20
  strcmp(“cap”,”cup”);	//returns -20

strcmp(“cap”,”cap”);   //returns 0

- strcmpi(“Hello”,”hello”); //returns 0

- char s1[20], s2=”Programming”;
  strncpy(s1,s2,4);   //s1 gets value “Prog”

To find the types of characters in a given string, there are several functions provided by C through a header file ctype.h. Following table lists out few of such functions.

Functions available in ctype.h

Function	Operation
isalpha(ch)	Finds whether the character ch is an alphabetic character
isalnum(ch)	Checks whether alphanumeric (either alphabet or numeric) character or not
isdigit(ch)	Checks whether ch is a digit or not
islower(ch)	Checks whether ch is a lower case alphabet
isupper(ch)	Checks whether ch is an upper case alphabet
isspace(ch)	Finds whether ch is a white space character
isprint(ch)	Checks whether printable character or not
isxdigit(ch)	Finds whether hexadecimal digit
tolower(ch)	Converts ch into lower case
toupper(ch)	Converts ch into upper case
toascii(ch)	Converts ch into equivalent ASCII character

1.15.4 Programming Examples on Strings

The important programs involve solving the following issues without using inbuilt string function –

- Finding length of the string
- Concatenate two strings
- Copy one string to other
- Reversing a string and checking whether it is palindrome or not
- Comparing two strings.

Also, some problems on single characters involve:

- Counting number of vowels and consonants in a sentence
- Changing case of a letter in a sentence (upper case to lower and vice-versa)

Write a C program to check whether a string is palindrome or not.

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

```
void main()
{
        char str[15],revstr[15];
        int i,len=0,flag=0;
        clrscr();

        printf("Enter a string:\n");
        scanf("%s",str);

        for(i=0;str[i]!='\0';i++)
                len++;
        i=len;
        while(i>=0)
        {
          revstr[len-i]=str[i-1];
          i--;
        }
        revstr[len+1]='\0';
        printf("\nThe Reverse of %s is %s",str,revstr);

        for(i=0;i<len;i++)
        {
                if(str[i]!=revstr[i])
                {
                        flag=1;
                        break;
                }
        }
        if(flag==1)
                printf("\nIt is not a Palindrome");
        else
                printf("\nIt is a Palindrome");

        getch();

}
```

Write a C program to concatenate two strings without using string functions.

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

void main()
{
        char str1[10],str2[10],str3[20];
```

```
        int i,j;
        clrscr();

        printf("Enter the first string:\n");
        scanf("%s",str1);

        printf("Enter the second string:\n");
        scanf("%s",str2);

        for(i=0;str1[i]!='\0';i++)
                str3[i]=str1[i];

        for(j=0;str2[j]!='\0';j++)
                str3[i+j]=str2[j];

        str3[i+j]='\0';

        printf("The concatinated string is: %s",str3);
        getch();

    }
```

Write a C program to read a sentence and to count the number of vowels and consonants in that.

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

void main()
{
        char ch;
        int vowel=0,cons=0;
        clrscr();

        printf("Enter a sentence:\n");
        while((ch=getchar())!='\n')
        {
                switch(tolower(ch))
                {
                        case 'a':
                        case 'e':
                        case 'i':
                        case 'o':
                        case 'u': vowel++;
                                        break;
```

```
                    default : if((ch>='a' && ch<='z')||(ch>='A' && ch<='Z'))
                                        cons++;
                                    break;
            }
        }

        printf("\nNumber of vowels=%d",vowel);
        printf("\nNumber of consonents=%d",cons);
        getch();
    }
```

Write a C program to read a sentence and to replace the uppercase character by lowercase and vice-versa.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>

void main()
{
        char ch;
        clrscr();

        printf("Enter a sentence:\n");

        while((ch=getchar())!='\n')
        {
                if(islower(ch))
                        ch=toupper(ch);
                else if(isupper(ch))
                        ch=tolower(ch);
                putchar(ch);
        }
        getch();
    }
```

Write a C program to compare two strings without using string functions.

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

void main()
{
```

```
        char str1[10],str2[10];
        int i,len=0,flag=1;
        clrscr();

        printf("Enter first string:");
        scanf("%s",str1);
        printf("\nEnter second string:");
        scanf("%s",str2);

        i=0;
        while(str1[i]!='\0')
        {
                len++;
                i++;
        }

        for(i=0;i<len;i++)
        {
                if(str1[i]!=str2[i])
                {
                        flag=0;
                        break;
                }
        }

        if(flag==0)
                printf("\nString are not equal");
        else
                printf("\nStrings are equal");

        getch();
}
```

1.16 PASSING ARRAYS TO FUNCTIONS

Passing an array to a function as parameters requires pointer to an array. So, we will discuss, how a pointer is created for an array.

Illustration of pointer to array

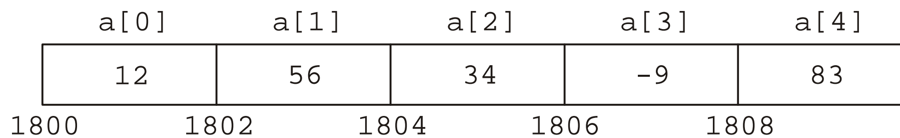
```
        #include<stdio.h>

        void main()
        {
            int a[5]={12, 56, 34, -9, 83};
            int i=0, *pa;
```

```
        /* address of the first element of the array a, that is the address of a[0] is
        stored in pa.  This is known as base address of the array a[].
        */

        pa=a;               //or pa=&a[0];

        for(i=0;i<5;i++)
             printf("%d\n", *(pa+i));
    }
```

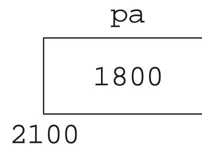
The output would be:

12 56 34 –9 83

In this program, when we declare an array a[5] and initialize it, the memory is allocated as:



And the memory allocation for the pointer pa will be:



Now, inside the *for* loop, when the statement

printf("%d\n", **(pa+i)**);

gets executed for the first time, value of *i* is 0. Hence, the compiler de-references the address **(pa+i)** using the *value at the address(*)* operator. So, the value stored at the address 1800 i.e. 12 is printed.

In the next iteration of the loop, *i* will be 1. But, as per pointer arithmetic, **pa+1** is 1800+2bytes. Thus, the pointer points towards the next location with the address 1802, but not 1801 as normal variable. So, the array element at the address 1802 (i.e., 56) gets printed. The process continues till all the array elements get printed.

Note

1. It can be observed that all the following notations are same:

   a[i] , *(a+i), *(i+a), i[a]

   All these expressions means i^{th} element of an array a.

2. Consider the declarations:

```
int  a[3] = {12, 56, 78};
int *p ;
```

```
p = a;
```

Now, there is a difference between the statements:

printf("%d", ***(p+1)**); and
printf("%d", **(*p+1)**);

The first expression will increment the pointer variable and so, forces the pointer to point towards next array element. Then the pointer is de-referenced and the value 56 gets printed. But the second expression will de-reference the pointer to get the value 12 and then 1 is added to it. So, the output will be 13.

So, the programmer should take care while incrementing the pointer variable and de-referencing it.

Arrays can be passed to the functions using base address of the array. Consider the following example for illustration:

Program for finding average of n numbers

```
#include<stdio.h>
float Average(int a[], int n)
{
        int  i, sum=0;
        for(i=0;i<n;i++)
                sum=sum+a[i];

        return (float)sum/n;
}

void main()
{
        int x[10], n, i;
        float avg;

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

        printf("\nEnter array elements:");
        for(i=0;i<n;i++)
                scanf("%d", &a[i]);

        avg =Average(a, n); //passing base address of the array and value of n
        printf("Average = %f", avg);
}
```

In the above example, you may feel that, the pointers are not being used. But, actually, the name of the array which we are passing to the function ***Average()*** itself is the base

address of the array or pointer. If you want pure pointer notation, the above program can also be written as –

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

float Average(int *p, int n)
{
        int i, sum=0;
        for(i=0;i<n;i++, p++)
                sum=sum+(*p);

        return (float)sum/n;
}

void main()
{
        int a[20], *pa, n, i;
        float avg;
        clrscr();

        printf("Enter n:");
        scanf("%d", &n);
        pa=a;
        printf("\nEnter values:\n");
        for(i=0;i<n;i++,pa++)          //pa will be pointing last element in last iteration
                scanf("%d", pa);

        pa=a;          //reset pa to base address of array
        avg=Average(pa, n);
        printf("\nAverage=%f", avg);
}
```

We can pass base address of two dimensional (or any multidimensional) array to the function. For example:

Program for reading and displaying a matrix:

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

void read(int x[5][5],int m,int n)          //address of 2-d array is received
{
        int i,j;

        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
```

```
                    scanf("%d",&x[i][j]);
}

void display(int x[5][5],int m,int n)
{
        int i,j;

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        printf("%d\t",x[i][j]);
                printf("\n");
        }
}

void main()
{
        int a[5][5], m,n;
        clrscr();

        printf("Enter order of matrix:");
        scanf("%d%d",&m,&n);

        printf("\nEnter the elements of first matrix:\n");
        read(a,m,n);

        printf("\nThe matrix is:\n");
        display(a,m,n);         //base address of 2-D array is being passed
        getch();

}
```

1.17 PASSING STRINGS TO FUNCTIONS

String is an array of characters. So, it is possible to have pointer to a string almost in same manner as with arrays. Consider the following example as an illustration:

```
#include<stdio.h>
void main()
{
   char str[20]="Programming", *p;
   p=str;                  //assign base address

   /* till the character stored at p becomes null character ('\0')  print the character
  and increment pointer */
```

```
    while(*p!='\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

The output is: Programming

With the help of pointers, it is possible to perform standard operations like finding length of a string, copying the strings, comparing two strings etc. Consider the following example –

Finding length of a string and copying the strings

```
#include<stdio.h>
void main()
{
    char str1[20]="Programming", str2[20],*p1,*p2;
    int len=0;

    p1=str1;
    p2=str2;

    for(len=0; *p1!='\0'; p1++);

    printf("Length = %d", len);

    p1=str1;

    //string copy
    while(*p1!='\0')
    {
      *p2=*p1;
      p1++;
      p2++;
    }
    p2='\0';           //assign null at the end of second string
    printf("\nString2 is : %s", str2);
}
```

The output is:

Length of first string is: 11

String2 is: Programming

In the above example, usage of both *for* loop and *while* loop is given. The student can choose any of the loops to do the program.

1.17.1 Passing strings to functions

With the help of pointers, we can pass the strings to functions, as string is nothing but a character array. Consider the following example:

```
#include<stdio.h>
void fun(char *s)
{
    printf("%s", s);
}

void main()
{
    char str[20]="hello";
    fun(str);
}
```

The output: hello

Here, in the main() function, actually, we are passing base address of the array *str* and which is being received by the parameter *s* in the function *fun().*

Passing a string to the function can be achieved in any of the two ways:

void fun(char s[ ]) or
void fun(char *s)