

# **‘C’ Programming Language**

## **(BSIT 21)**

*Contributing Author*

**Dr. G. Raghavendra Rao**  
Prof. & H.O.D.  
NIE, Mysore

## **Syllabus**

1. Introduction to the concept of flowcharts, algorithms and programming simple flowcharts.
2. Fundamentals of C-Variables, data types, arithmetic expressions, their priorities etc.. Library functions.
3. Control Structures of C:- for, while and do loops, if then else statements, switch statements.
4. Arrays: Their creation and manipulation, multi-dimensional arrays.
5. Functions: Definition, parameter passing mechanisms. Use of functions with arrays, structures etc..
6. Structures: Definition, creation and manipulation.
7. Pointers: Definition and operation pointers & structures, linked lists, pointers & functions
8. Bitwise operators, preprocessors.

## **CONTENTS**

**Unit – I**

- I Flow Charts and algorithms
- II C Fundamentals
- III Control Structures
- IV Arrays

**Unit – II**

- I Functions
- II Structures
- III Files

**Unit – III**

- I Pointers
- II Bitwise Operators
- III Preprocessor

**COURSE INTRODUCTION**

In this course, you learn the programming language called “**C**”. You learn about basic concepts of programming and the language itself in detail. Just like any other human

language, the programming languages also have a format in which they are to be written. But the constraints are more stringent. Human beings are capable of understanding sentences that are not fully grammatical, but a compiler simply rejects a program even if a single comma or semicolon is not in place. That way we need to be extra careful when writing programs.

The course itself is divided into 3 units - The first unit talks of the basics of the language, data formats, control structures and the concept of arrays. In each case, a brief explanation of what it means is given, followed by the general format of the operation. This is followed by a large number of sample programs. In fact any study of a programming language is complete only when one is able to analyze & solve a variety of problems. These programs are designed in the increasing order of their complexities and many of them also come with sample outputs. Also the explanation of the programs themselves are given, wherever deemed necessary.

The second unit gives an insight into concepts like functions, structures and files.

The third unit deals with advanced features like pointers, bitwise operators, preprocessors, etc..

## **UNIT – I**

### **UNIT INTRODUCTION**

This unit teaches you about the fundamentals of C language. You may notice that this unit is bulkier than other units. It is because, to start with you have to know many of the important features - all at once.

We start with the fundamentals of programming itself. What is programming ? How do we convert a given real life problem into a programming problem? How do we analyse it and how do we decide on the methodology to be adopted for the same? These aspects and also the method of converting the proposed program to a flowchart and/ or an algorithm so that they become ready to be coded into a program are being discussed in the first block.

Then afterwards, the basics of the language - the variables, constraints, simple formats for a typical C program are discussed in the second block. You also get introduced to the concept of library functions, header files etc..

The third block deals with the control structures in C. They control the flow of program. Though normally the control flows sequentially, i.e., the first line gets executed first.

Then the second line, etc., it is possible to alter it by various methods. You get introduced to concepts of decision making using If statements, Switch Statements, the concept of loops, the different types of loops etc..

The fourth block discusses the concept of arrays. When you have a large number of data items to be handled under a common name, arrays are used. They help remember and manipulate data easily.

By now, you will also have picked up the bare fundamentals of C. Hence, we also try to solve a very large number of programs. Wherever the algorithm is new or needs some explanation, you see comments being liberally provided. Otherwise, you will straightaway be able to study and understand them.

## **BLOCK - I**

### **FLOW CHART**

Block introduction: In this block, you are introduced to the concept of flowcharts and algorithms. Whenever you are asked to solve a problem using a computer, you should first convert it to a sequence of steps which can be solved by a programming language. While in the case of very simple programs this can be done mentally, in real time problems, these are to be done in a systematic manner.

The flowcharts provides a schematic way of doing the same by connecting a sequence of fundamental blocks to indicate the flow of control. You are introduced to the blocks and also the methods of interconnecting them.

You are also introduced to the concept of algorithms, which describe the problem in an English-like language, so that they can be converted into programs.

## FLOW CHARTS AND ALGORITHMS

In this block, we learn about the most fundamental aspect of problem solving through computers - Preparing the problem to be solved by the computer. A computer cannot understand the problem nor it solves it - it can only perform a set of predefined operations. It is the programmer's job to convert the given problem into a set of instructions so that those actions, when performed would solve the problem.

Let us take an example. Suppose you want to pick up a book from a table. For a human being, it is enough to say, "Please pick up the book from the table". He will go to the table and pick up the book. On the other hand, imagine a totally dumb servant, who cannot reason out any thing by himself but can faithfully follow your instructions. For him you have to describe the operation in steps like

Go to your left by 5 steps

Go forward by 10 Steps

Align yourself in front of the table

Stretch your hand to hold the book

Lift the book and hold it in your hand.

A computer can be likened to the servant, who is highly competent and faithful as long as following the instructions are concerned, but cannot reason out by himself. So, you as a programmer will have to break the problem into a sequence steps, similar to those described above for the computer to work with. For this you should know the computer's language. One of them is "C", which you are going to learn in full detail in this course. But before that, you should be able to break the problem into the simple steps so that you can describe these steps in the language.

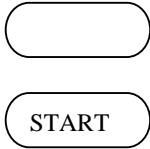
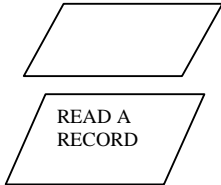
In simple cases, you would be able to do it straightaway. With some practice, most of the simple or even medium-sized problems can be directly coded into the computer language. But when you have really complicated problems on hand, you need to analyze it step by step.

Further, such an analysis will also help you remember what you have done and modify, if necessary, at a much later date. Also, if some other person wants to know your sequence of reasoning, obviously it should be possible for you to describe your logic in simple, unambiguous steps. Because of all these reasons, it is highly essential to develop some form of intermediary mechanism to describe the problem on hand as a sequence of steps, which can be later converted to a program. There are two such commonly used and universally accepted methods of describing the solution process

- i) The Flow Charts, and
- ii) The Algorithms

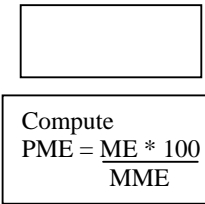
**The flow Chart:** As the name suggests, it is a chart to describe the flow of the solution process. You break up the solution into simple steps and connect them to describe the flow so that the last of these blocks leads you to the solution.

We briefly describe the commonly used symbols to represent these blocks and how to combine them to form the solution.

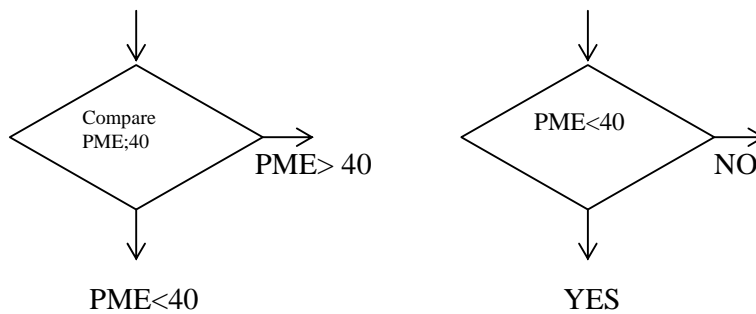
<p>1. <b>Terminal:</b> This symbol is used in the beginning and at the end of the flow chart. Here the beginning and the end refer to the logical beginning and the logical end of the sequence of operations. Start, Stop, End, or Exit is Written inside the symbol to indicate the type of terminal. For example the beginning of the flow chart is represented as given here.</p>	
<p>2. <b>Input/Output:</b> The symbol parallelogram is used for reading or writing of data. This symbol is used for all types of input and output, such as floppy reading, printing, magnetic tape reading or writing, or disk reading or writing, etc. The brief description of the operation is written inside the symbol. For example, reading a record can be represented as given here.</p>	



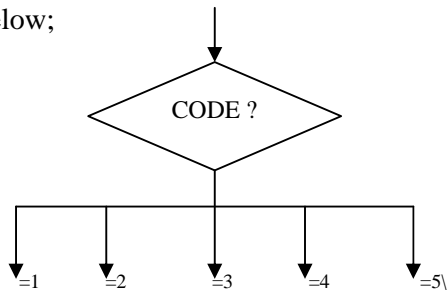
3. **Processing:** The symbol rectangle is used for processing operations, i.e. for arithmetic computations (addition, subtraction, multiplication, division) or moving the data from one location or core storage to another location. For example, the computations of PME (percentage of marks in English) can be represented as given here.

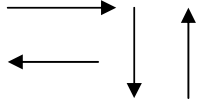
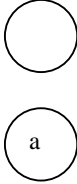






4. **Decision :** The diamond shaped symbol is used for representing a decision point. It indicates that a condition is to be tested and one of the alternative paths is to be followed. For example, PME can be compared 40 (for deciding whether it is more or less or equal to 40) and can be represented as follows:

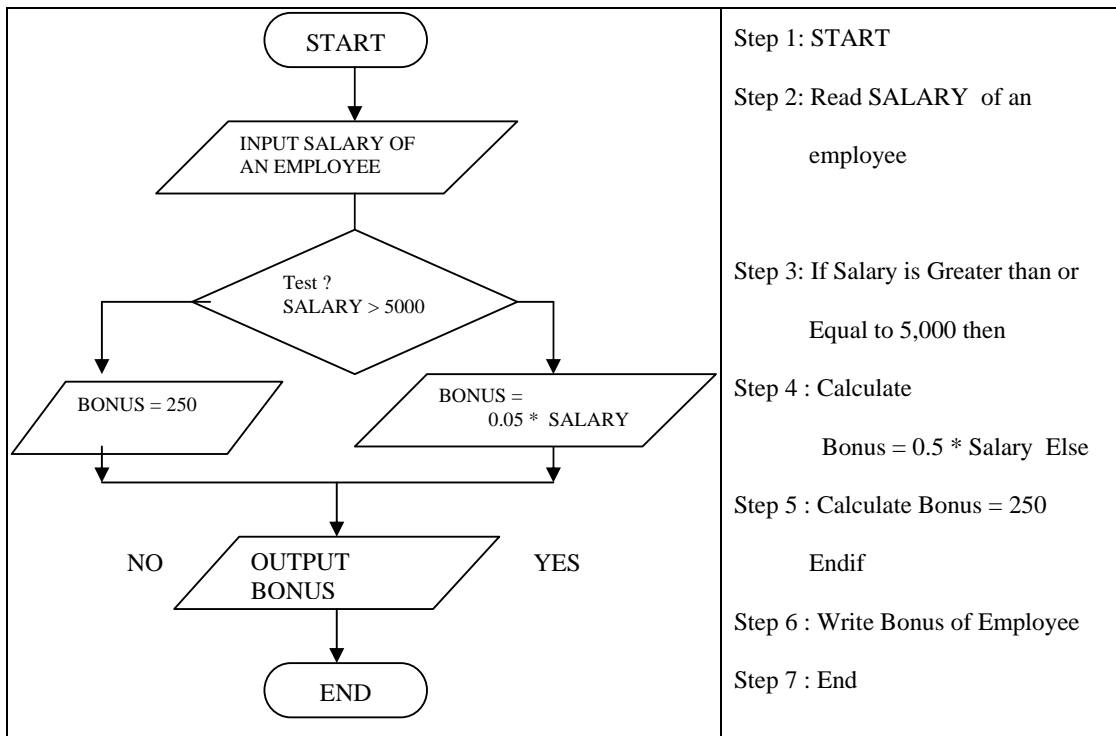


Sometimes, the number of alternative paths after the decision point are three or even more then the symbol used is as given below;

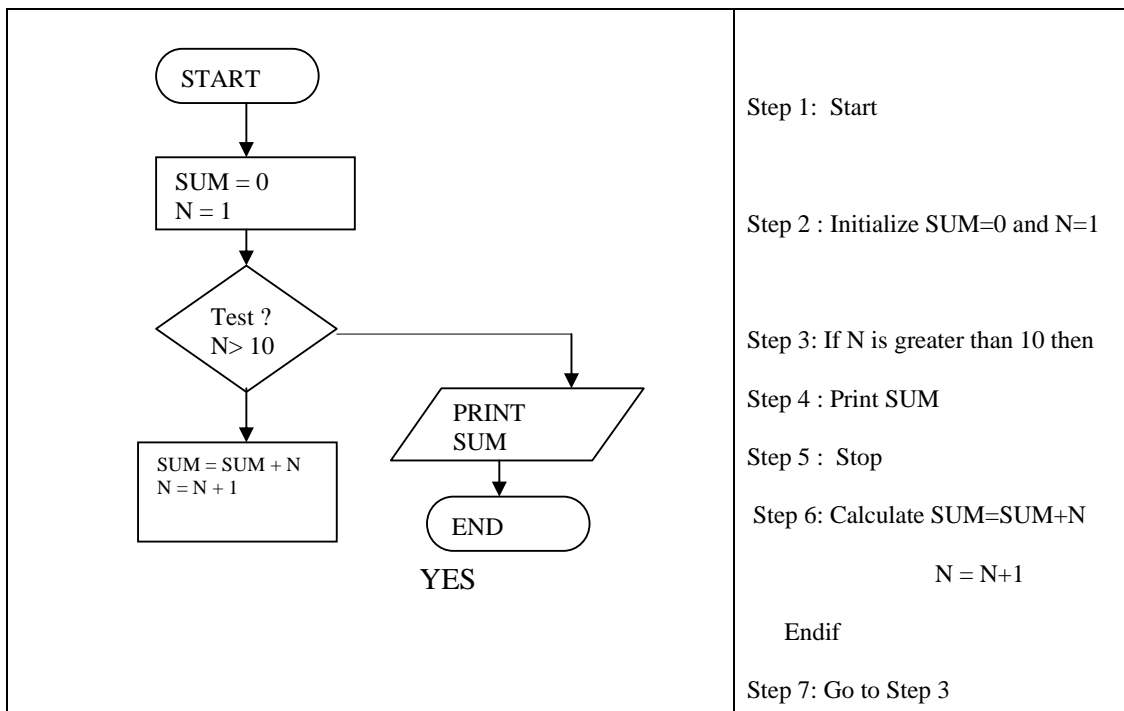


<p>5. <b>Flowline:</b> The straight lines with arrowheads show the path of flow. Arrows on the line indicate the direction of flow. The flowline connects the various symbols</p>	
<p>6. <b>Connector:</b> A small circle is called the connector, and is used to show the entry to or exit from the logical path of a flow char. In fact, this symbol replaces the long flowlines connecting the different parts of a flowchart on the same page. One symbol shows where the branching of flow is done whereas the other symbol shows the entry of the flow. Inside the symbol the name of the entry or exit point is written.</p>	
<p>7. <b>Preparation:</b> This symbol is used for the house-keeping operations. All those operations, which are done before starting the actual processing of data, come in this category. For example, setting the number of students to zero before processing of data starts.</p>	
<p>8. <b>Predefined:</b> This symbol indicates that a routine or process, which is shown in the flowchart separately, is executed at this position. Normally when a specific process is used at different logical paths then the use of this symbol provides the simplicity in the flowchart.</p>	
<p>9. <b>Offpage Connector:</b> This symbol is used in place of connector when the exit and the corresponding entry point are shown on different pages in the flowchart.</p>	
<p>10. <b>Annotation, Comments:</b> This symbol provides the facility of writing more description of an operation. When the space inside a symbol is not sufficient to write the details, then this symbol is used. The dotted line between the symbol and flow chart shows the point to which it relates.</p>	

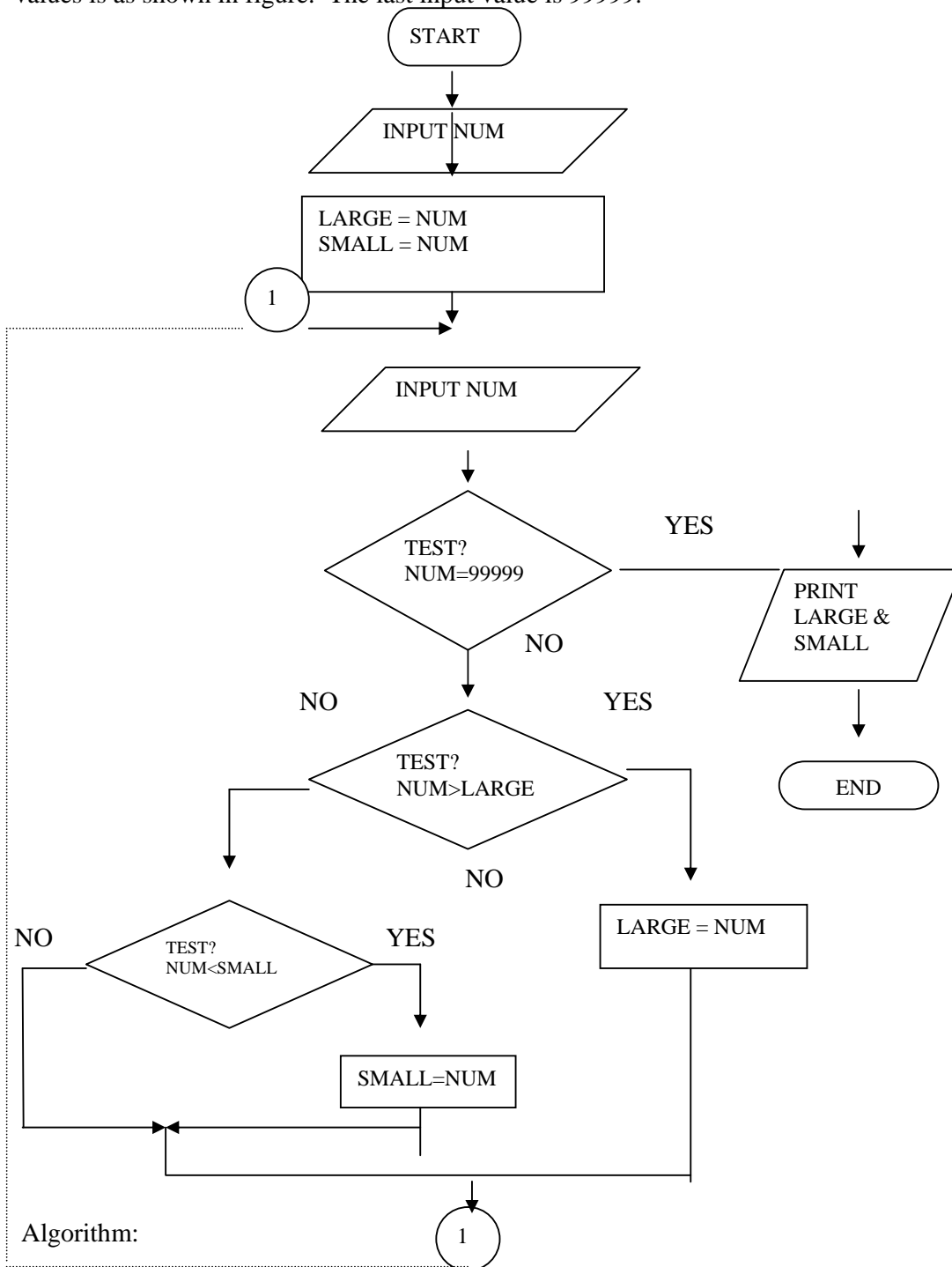




Example c) The flowchart and algorithm for adding the integers from 1 to 10



Example d) A Flowchart to calculate the largest and the smallest numbers in a list of input values is as shown in figure. The last input value is 99999.



1. Read the first number
2. Take that number as the largest number
3. Take that number as the smallest number also
4. Input the next number

If this num = 99999, you want to end the session,

So print the largest and smallest numbers and STOP.

Else

5. if the number is  $>$  the present large number, make the new number the largest.

Else

6. If the number  $<$  the present small number, make the new number as the smallest.

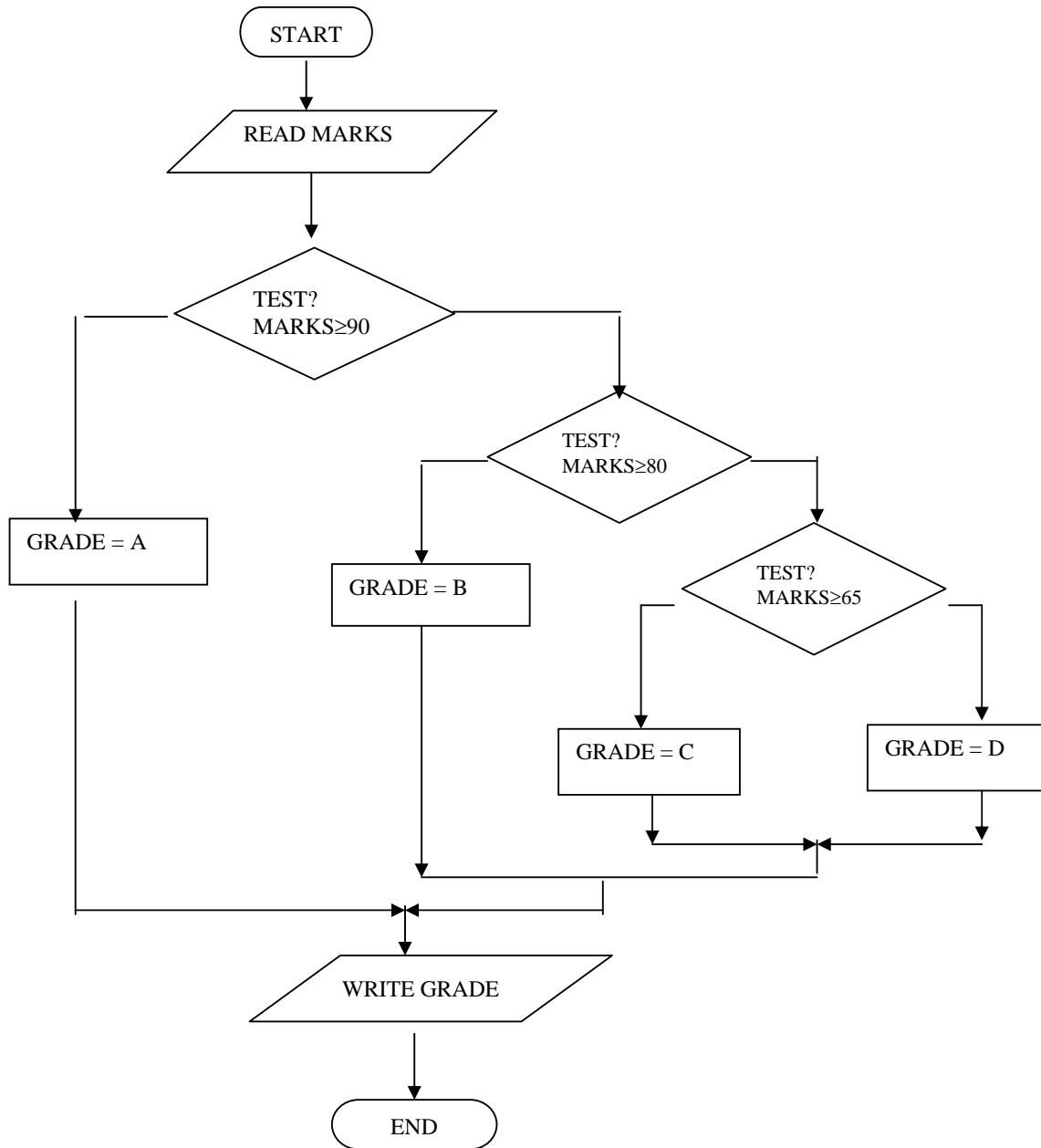
7. Goto step 4.

Example C: A flow chart to read the marks of a student and classify him into different grades.

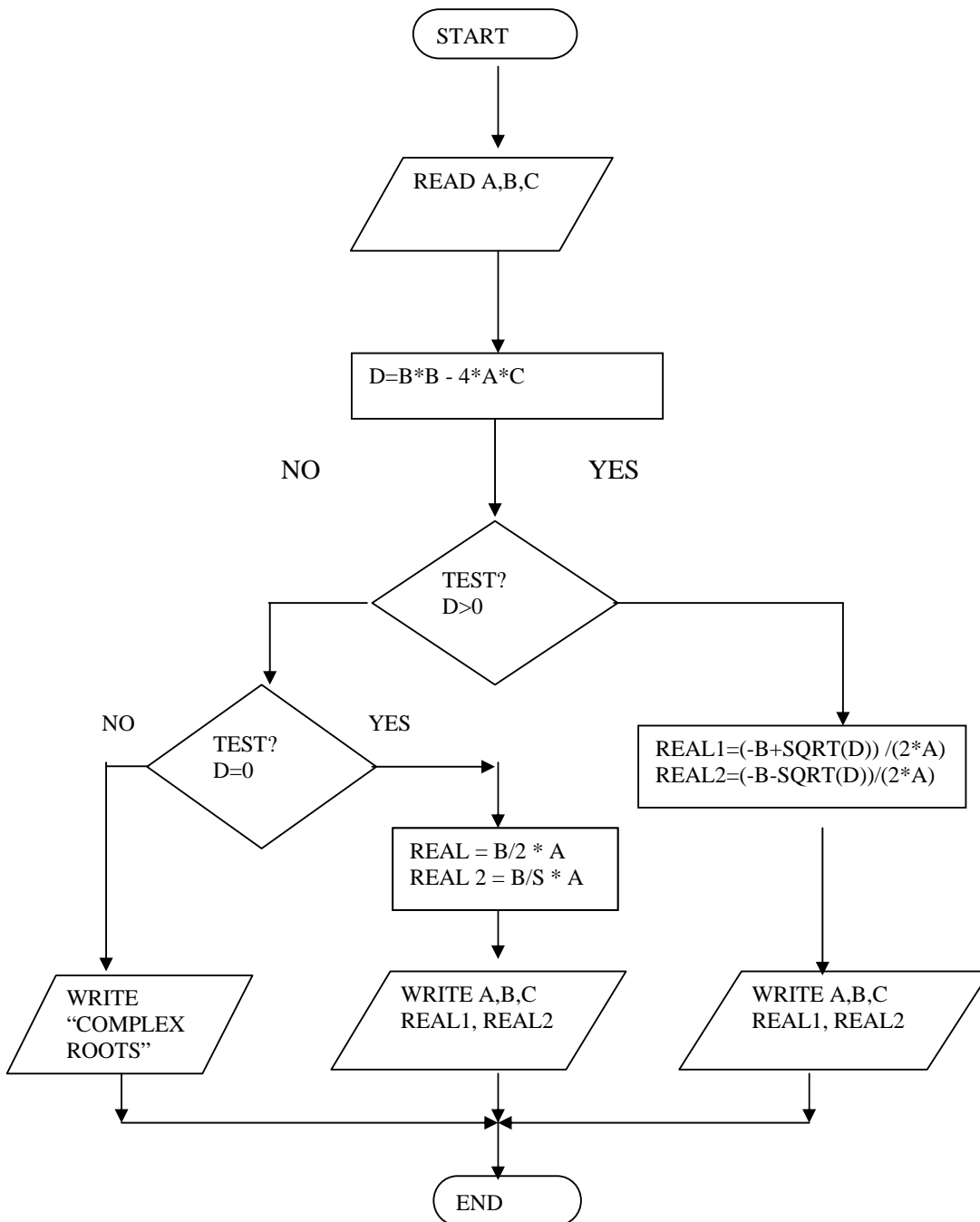
If marks greater than or equal to 90 Grade A, greater than or equal to 80 but less than 90 Grade B, Greater than or equal to 65 but less than 80 Grade C otherwise Grade D.

Algorithm:

1. Read marks
2. If marks  $\geq 90$   
     Declare Grade = A
3. Else If marks  $\geq 80$  declare grade = B
4. Else if marks  $\geq 65$  declare Grade = C
5. Else declare Grade = D
6. Write Grade
7. End



Example f: A flow chart to find the roots of a quadratic equation.





1. Read the parameters of A,B,C
2. Evaluate  $D = b^2 - 4 * a * c$
3. If  $D > 0$ ,
  4. Evaluate the first root as  $-b + \text{sqrt}(D) / (2 * a)$   
Second Root as  $-b - \text{sqrt}(D) / (2 * a)$   
Else
5. If  $D = 0$ 
  6. Evaluate First root as  $-b / 2 * a$   
Else
7. If  $D < 0$ , Simply write complex roots.
8. End.

## **BLOCK - II**

### **C-FUNDAMENTALS**

#### **BLOCK INTRODUCTION**

In this block you get introduced to the C languages. The basic method of storing data-using variables and constraints is discussed. The rules for framing such variables, the various arithmetic and logical operators, simple input/output operations and you also learn to write your first C programs - very simple - but nevertheless complete programs.

## C – FUNDAMENTALS

We will now start the study of the programming language – ‘C’. C evolved from a succession of programming of programming languages developed by Bell Laboratories in early 1970s. The increasing popularity of the **unix** operating system, which has C as it’s “Standard” programming language, further enhanced the usefulness of C making it arguably the most popular of the programming languages.

We now start with the fundamentals of the language. Any language, as you know is made of sentences, which are made up of words, which in turn are made up of characters i.e. we start learning the characters, then learn how to combine them into words, combine the words into sentences and so on.

### Characters of C

C makes use of the normal characters of English – a to z, 0 – 9 and several special characters like + - \* / , . % # \$ & “ = ; ( ) { } \ and so on. Most of the compilers are also case sensitive. i.e. they do differentiate between capital and small letters so you should not go about mixing them. It is a good practice and also pleasing to type all program with small letters.

### Variables of C

Just as a language is made up of names, verbs, adjectives, adverbs etc., C programs are made up of variables, constants, arithmetic operators etc. We learn them one after another in the following sections.

A variable in a program is a name to which you can associate a value. For ex. with the name A you can associate, say, a number 10, so that whenever the number A is called/used, you get the value 10.

**Rules for framing variables.**

They begin with a letter or underscore(\_), and may be followed by any number of letters, underscores or digits (0-9). But you cannot have special characters or blankspaces in a variable name.

The following are valid variables

Prod

sum\_total

I

J

\_sysflag.

The following are invalid variables

Sum \$ total

part total

5load

int

Now based on the previous definition, we leave it to you to decide why they are invalid. But the last word int appears to be valid as per the definition. It is a invalid variable, because it is a “Reserved name”. The C compiler, uses a reserved name with a special name and you can not use it with your own definition. As we go about, we pickup a large number of such reserved words and their meanings.

One sound advise for choosing the names of variables. You can practically use any name to store any value. For example A or M can be used to store sum, product, quotient any thing as long as you know what is it that you have stored. However, it is a good practice to use meaningful names to store data. You can store the result of addition in sum, product of two numbers in prod, etc.. When your are writing long programs, such practices help you a lot in checking the correctness (technically called debugging) and understand the programs.

## Data types and constants of C

The concept of variables that we have just introduced to is just to initiate you to the various ways in which data can be stored in C. We now seek a more formal and complete description of the various ways in which data can be manipulated in C.

A variable name is one which the stored data can be changed during the execution of the program. For example if you have store 10 in A initially, you can add or subtract something to it or even store a totally different value. i.e the data A is “variable”. On the other hand you can also have certain constants, whose values do not change during execution.

Variables themselves come in different types. They can be integers (whole numbers), floats (fractional numbers) doubles and characters. Characters as the name suggests are used to store characters. A number of characters in a string are sometimes called the string (Str) variables. The data type doubles need some introduction. Normally a C compiler can store numbers of certain size only. The actual size depends on the computer you are using and the compiler in use. If you suspect that you are likely to use numbers bigger than this, you should use double declaration, which will give you twice the number of places. i.e. if your original limit was 5 places, you may now get to use up to 10 places which will suffice in most cases.

C expects you to list the variables you are using in your program at the beginning itself and also their types. For ex.

```
Int a,b,c
```

```
Float no,n1,n2;
```

```
Char name, title;
```

makes a,b,c available for storing integer numbers (like 10,20,30), no,n1,n2 store fractional numbers (like 10.42, 13.56, 15.7 etc) and name and title store character strings (like rama, student etc.).

The semicolon (;) at the end is used extensively in C to demarcate one statement from another (Just as . is used in English).

**Arithmetic Expressions:** Once you have learnt about integers, float etc, the next stage is to combine them. They can be combined with the usual addition, subtraction symbols.

For ex.  $A + B$ ,  $a - c$ ,  $a * b$  etc. can be used. However, mere such operations are not enough. The data that you get after the operation is to be stored in a variables name. Hence, you can write

```
a = a + 10;
```

```
Sum = a+b;
```

```
X=y*z; etc.
```

In the first case, 10 is added to the original value of a (what ever it was) and is stored as the new value of a. (the original value is lost). In the second case a and b are added and the result is stored in sum. In this case, since a and b are not being overwritten, their values continue to be available to you. In the third case, the product of y and z is stored in x.

Before you can start writing small programs, we also conclude about precedence of operators. In a long expression with a number of such operators, which one is evaluated first?

The answer is first all multiplication and division operators are completed. Then all addition and subtraction are taken up.

For ex. If we write  $A = b + 2 * c$ ;

$2 * c$  is evaluated first (multiplication) and this is then added to b.

If there are more than one multiplication, addition, etc. the computations are done from left to right.

Ex:  $a = b + c / d + e * f$

$c / d$  is evaluated first, then  $e * f$  then b is added to the result of  $c / d$  and finally the result of  $e * f$  is added. The entire value is stored in a.

Try yourself: Suppose b is 4, c is 8, d is 2, e is 4 and f is 2, what value is store in a?

Sometimes, we may like to override the precedence of operators of C. Suppose in the above example, I want  $b + c$  to be added first and then divided by  $d$ . I can still do it, simply enclosing  $b + c$  within parentheses  $(b+c)$ . The rule is whatever is within the parentheses is evaluated first. So  $(b+c)$  is evaluated first and then is divided by  $d$ .

Try evaluating  $a = (b + c)/d + e*f$  with the above given values for  $a, b, c, d, e$  and  $f$ . You notice that, by enclosing them within parentheses, we have changed the result. So, whenever you are evaluating long expressions, it is always desirable to be careful about the precedence of operators. However, one golden rule is that whenever in doubt, use parentheses. For ex; in our original example, you want  $c/d$  to be evaluated first, there is no need for parentheses to be used. However, if you are not confident of your judgement, simply enclose them in parentheses as in  $a = b + (c / d) + (e * f)$ .

You should note that the expression within brackets need not be as simple as we have illustrated. They can be as long as you like, there can be parentheses within parentheses and so on. Again, within these parentheses the same rules of precedence apply, including the fact that those inside the parentheses are evaluated first. So, in effect the evaluation starts from the innermost parentheses and proceeds outwards.

For ex, I can write  $a = b + c / ((d + e) * f)$

wherein the innermost parentheses  $d + e$  is evaluated first, this is multiplied by  $f$ ,  $c$  is divided by the whole value and finally it is added to  $b$ . (What will be the result if you take the above mentioned values? )

Again remember the golden rule:

Whenever in doubt, use parentheses. Use of extra parentheses will not affect the results, but not using them when needed changes the final results. Now, we are in a position to illustrate these with a sample C program. However, before we can write a full program, we introduce you to one more concept of getting the output from the program. There is a function `printf()` which allows you to do this.

```
/* Illustrate the use of various arithmetic operators */
```

```

#include<stdio.h>

main( )
{
    int a = 100;

    int b = 2;

    int c = 25;

    int d = 4;

    int result;

    result = a-b;          /*subtraction */

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

    result = b * c          /* multiplication */

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

    result = a / c;          /* division */

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

    result = a + b * c;      /* predence */

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

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

}

```

output :

a - b = 98

b\* c = 50

a / c = 4

a + b + c = 150



$$a * b + c * d = 300$$

Now a description of what we did in the program.

Whatever we write in between `/*` and `*/` is a comment. i.e. the C compiler will not process it. You can write your comments and explanations so that it will be easier for you to understand the program, when you go through it at a later date.

`#include <stdio.h>` is used to include certain input / output functions. We discuss about them later. As of now we simply presume that it should be there.

`Main()` indicates that it is a main program, we study more about it later. Notice that the entire program is enclosed between the brackets `{` and `}`. Later, we see that there can be many more such segments enclosing portions of the program. The declaration `int a = 100;` declares `a` as an integer variable and stores 100 in it. Similarly stores 2 in `b`, 25 in `c` and 4 in `d`. The variable `result` is also declared as an integer, but no value is stored in it as of now. As and when the calculations are done, the result of the calculation are stored in `result`.

(Note that after `a,b,c,d` we could have declared an `e`. But since we are starting the result, we have called it `result`, so that we immediately know what to expect in it).

Now look at the statement : `printf("a - b = %d \n",result);`

`Printf`, as we have already described is used to print the output. Now whatever is between the inverted commas "`"` and `"`" is printed as it is, except `%d` and `\n`. `%d` indicates that an integer value is to be printed at that place and `\n` indicates that after printing, the cursor should go to the next line. Now, outside the inverted comma, we have the `result`, which is an integer. So the value of integer stored in variable `result` is printed where `%d` is appearing.

In effect the output looks like this :

`a - b =` is printed as it is

In place of `%d`, the value of `result` is printed (What ever is its value)

Because `\n` is there, the control goes to the next line i.e the next `printf` comes in the next line.

Now analyse the entire program and the output of the same given above. To make things more familiar, we write one more very similar program. This will evaluate the expressions we encountered during the discussion on precedence of operators.

```
/* Illustrate the use of precedence of operators */

#include <stdio.h>

main()
{
    int a;

    int b = 4;

    int c = 8;

    int d = 2;

    int e = 4;

    int f = 2;

    a = b + c / d + e * f      /* result without parentheses */

    printf("The value of a is = %d \n", a);

    a = (b + c) / d + e * f    /* result with parentheses */

    printf("The value of a is = %d \n", a);

    a = b + c / ((d + e) * f)  /* another result with parentheses */

    printf("The value of a is = %d \n", a);

}
```

output :

The value of a is = 16

The value of a is = 14

The value of a is = 1

```
/* More arithmetic expressions */
#include<stdio.h>
main( )
{
    int a = 25;
    int b = 2;
    int result;
    float c = 25.0;
    float d = 2.0;

    printf("6 + a / 5 * b = %d \n", 6 + a / 5 * b);
    printf("a / b * b = %d\n", a / b * b);
    printf("c / d * d = %f\n", c / d * d);
    printf("-a = %d\n",-a);
}
```

output:

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

Note the difference between this and the previous two programs. When evaluate  $6 + a / 5 * b$ , we have not stored it's value in any result, but it is evaluated in the printf statement itself and printed straight away.

```

/* program to multiply two numbers */

#include <stdio.h>

main( )
{
    float num1,num2,product;

    num1=300.0;

    num2=0.6;

    product = num1 * num2;

    printf("\n %f times %f is %f\n",num1,num2,product);
}

```

output:

300.000000 times 0.600000 is 180.000000

```

/* program to computer average of three numbers */

#include<stdio.h>

main( )
{
    int a = 8;

    int b = 10;

    int c = 4;

    int sum,remainder;

    float average;

    /* Calculate the average and display the result */

    sum = a + b + c;

```

```

average = sum / 3;

remainder = sum % 3;

printf(The average of %d, %d, %d is %d and %d/3 \n",a,b,c,average,remainder);

}

```

Output:

The average of 8,10,4 is is 7.000000 and 1/3

There are some special types of arithmetic statements in C. Consider the statement  $i = i + 1$ ; it states add one to  $i$  and store the new value as  $i$ . Such statements are very frequently used in what are called “increment” operation. Suppose you want to perform an operation 10 times. Then all that you do is to perform the operation once, count  $i$  as one, perform it again, add 1 to  $i$ . Perform again. Add one more to  $i$  and so on.

C provides a special method of writing such counting operation. Instead of  $i = i + 1$ , you can write  $i++$ . Both mean the same. In this example, you perform the operation and then increment  $i$ . In some cases, you may want to first increment and then perform the operation. For such situations, we use  $++i$ . Whereas  $i++$  is called post increment (increment after the operation)  $++i$  is called preincrement. Of course, initially if you feel a bit uncomfortable about these statements, you can as well use  $i=i+1$  type of statements to begin with.

Similarly there are decrement statements, for situations when you want to count backwards - instead for say 1 to 10 suppose you want to count from 10 to 1. Then initially you put  $i=10$  then keep subtractive  $i=i-1$ . For such situation we have  $i--$  and  $--i$  post decrement where subtraction is done after the operation and subtraction is done before the operation respectively.

C also provides a list of arithmetic and logical operator. These will be useful, basically for the control structures operation (see next block). We give you the table of such operators, which can be used to compare two variables or statements.

## Scanf() function:

One thing you might have noticed in all the above programs is that we are not giving any inputs at all. The values needed by the program are being included in the program itself - only the output is being made available. Obviously, they cannot happen always. We may not know the input before hand always nor can we go on changing the programs, whenever the input data changes. So, there should an input function, which asks for input during the execution time and accepts the values from the keyboard. That function, obviously, is similar to printf( ) - it is called scanf( );

Whenever a scanf( ) is encountered, the computer waits for the user to give the value for that particular input from the keyboard and takes that as the value for the variable.

For ex, it in a program we have

```
Scanf("%d",&a);
```

When this statement is executed on the computer, the program waits. If you type in, say 10, the value of a is taken as 10.

There is a difference though. Instead of simple 'a', we write &a, which is a pointer to the location where a is stored. Any way, we come back to this topic again, For now, we can say all scanf( ) parameter come with the ampersand &.

## Library functions:

The C compiler wants to make the life of the user easier. It provides small program modules called library functions which are programs that perform functions that are needed often by programmers. The user will have to simply call them by their name & use it - he need not write them again and again. Some of the more frequently used library functions are listed below:

Cosine of the value	:	cos( )
Sine of the value	:	sin( )

Tangent of value	:	<code>tan( )</code>
Absolute value	:	<code>abs( )</code>
(-a is taken as a)		
Logarithm to base e	:	<code>log( )</code>
Square root	:	<code>sqrt( )</code>
Raising to a power	:	<code>pow( )</code>

For that matter even `scanf()` & `printf()` with which you are familiar are library functions.

These functions are available in special files - called header files. For ex: `scanf`, `printf` etc are available in a file called `stdio.h`, whereas `cos`, `sine` etc - are in a file called `math.h`. If you want to use these functions, you should include these files using the `#include` directive at the beginning of the program.

We have so far used only data type `%d`, a decimal value. Atleast three more type are frequently used `%f` for indicating the floating point(real) numbers, `%e` to indicate double length numbers and `%c` to store characters.

With these fundamentals we now write a large number of fairly simple programs.

### **Write a program to convert days to months and days**

Algorithm:

1. Start
2. Enter days
3. Calculate months  $\leftarrow \text{Days}/30$   
 $\text{Days} \leftarrow \text{Days} \bmod 30$
4. Output months, Days
5. Stop

**/\* PROGRAM TO CONVERT DAYS TO MONTHS AND DAYS\*/**

```
#include<stdio.h>

main ()
{
    int m,d;

    printf("Enter days");

    scanf("%d",&d);

    m = d/30;

    d = d/%30;

    printf("Months = %d Days = %d",m,d);

}
```

**Typical Output:**

Enter days: 305

Months = 10 Days = 5.

**/\* PROGRAM TO EVALUATE EXPRESSION \*/**

```
#include<stdio.h>

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

    a=2; b=10; c=3;

    x=a*c+b;
```



```

y=a*x*x+b*x+c;

printf("x = %f\n",x);

printf("y = %f\n",y);

}

/* END OF PROGRAM */

```

**Output:**

X=16.000000

Y=675.000000

```

/* PROGRAM TO READ A NAME AND DISPLAY THE SAME */

```

```

#include <stdio.h>

main ()

{

    char str[20];

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

    scanf("%s",str);

    printf("\nYour Name is . . . . . %s",str);

}

/* END OF PROGRAM */

```

**Output:**

Enter your name

SUDARSHAN

Your name is . . . . . SUDARSHAN

```

/* PROGRAM TO READ A STRING */

```

```
#include<stdio.h>

main ()
{
    char str[20];

    printf (“\n HI, WHAT IS YOUR NAME ? ”);

    scanf(“%s”,str);

    printf(“\n\n WELCOME %s, \n LET’S BE FRIENDS.”,str);

}

/* END OF PROGRAM */
```

**output:**

HI WHAT IS YOUR NAME ? DINESH

WELCOME DINESH

LET’S BE FRIENDS

**/\* TO ROUNDOFF A REAL NUMBER TO NEAREST INTEGER VALUE \*/**

```
#include<stdio.h>

main ()
{
    int d;

    float r,t;

    printf(“\nEnter a Real no.:”);

    scanf(“%f”,&r);

    t = r + 0.5;

    d = ((int)t);
```

```
printf("The value rounded off to the nearest integer is: %d",d);
}
```

```
/* END OF PROGRAM */
```

**output:**

Enter a Real no : 62.768

The value rounded off to the nearest integer is : 63.

**Write a program to find the area and perimeter of a circle given its radius**

Solution:

Algorithm:

1. Start
2. Assign  $Pi \leftarrow 3.1415$
3. Input radius
4. Calculate  $area \leftarrow pi * r^2$   
 $peri \leftarrow 2 * pi * r$
5. Output Area, Perimeter
6. Stop

```
/* CALCULATION OF AREA AND PERIMETER OF A CIRCLE */
```

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

```
main ()
```

```
{
```

```
float r,pi=3.1415, area,peri;
```

```
printf("\n Enter radius of circle:");
```

```
scanf("%f", &r);
```

```

    area = pi*r*r;

    peri = 2 * pi * r;

    printf("\n Area = %5.2f",area);

    printf("\n Perimeter = %5.2f",peri);

}

/* END OF PROGRAM */

```

**output:**

Enter radius of circle: 2.56

Area = 20.59

Perimeter = 16.08.

**Write a program to find the area and perimeter of a rectangle having length, l and breadth b.**

**Solution:**

Algorithm:

1. Start
2. Input length & breadth
3. Calculate area  $\leftarrow$  length \* breadth  

$$\text{peri} \leftarrow 2 * (\text{length} + \text{breadth})$$
4. Output area, peri
5. Stop.

**/\*CALCULATION OF PERIMETER AND AREA OF A RECTANGLE \*/**

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

```
main ()
```

```

{
    float l,b,area,peri;

    printf("\nEnter length of rectangle:");

    scanf("%f",&l);

    printf("\nEnter breadth of rectangle:");

    scanf("%f",&b);

    area=l*b;

    peri= 2*(l+b);

    printf("\n Area=%10.2f",area);

    printf("\n Perimeter=%10.2f",peri);

}

/*END OF PROGRAM*/

```

**Output:**

Enter length of rectangle: 2.5

Enter breadth of rectangle: 3.4

Area = 8.50

Perimeter = 11.80

**Write a program to accept temperature in Fahrenheit and convert it to degree celsius and viceversa. [ Hint:  $C = \frac{5}{9} * (F - 32)$ ]**

**Solution:**

**Algorithm:**

1. Start
2. Input temperature in Fahrenheit(F)
3. Calculate Celsius  $\leftarrow 5.0/9.0*(F-32.0)$
4. Output temperature in Celsius (C)
5. Input temperature in Celsius(C)
6. Calculate Fahrenheit  $\leftarrow (C*9.0/5.0) + 32.0$
7. Output temperature in Fahrenheit
8. Stop

```
/* CONVERSION OF TEMPERATURE IN DEGREE TO FAHRENHEIT AND VICE-
VERSA*/
```

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

```
main()
```

```
{
```

```
    float f,c,faren, cel;
```

```
    printf("\n Enter temperature in Fahrenheit:");
```

```
    scanf("%f",&f);
```

```
    cel=5.0/9.0*(f-32.0);
```

```
    printf("\nTemperature in Celsius =%10.2f",cel);
```

```
    printf("\n Enter temperature in Celsius:");
```

```
    scanf("%f",&c);
```

```
    faren=(c*9.0/5.0)+32.0;
```

```
    printf("\n Temperature in fahrenheit=%10.2f",faren);
```

```
}
```

```
/* END OF PROGRAM */
```

Output:

Enter temperature in Fahrenheit : 68

Temperature in Celsius = 20.00

Enter temperature in Celsius:20

Temperature in Fahrenheit = 68.00

**/\*WRITE A C PROGRAM TO INTERCHANGE THE VALUES OF TWO VARIABLES WITH AND WITHOUT USING TEMPORARY VARIABLE.\*/**

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

```
main()
```

```
{
```

```
    int a,b temp;
```

```
    printf("input the values for a & b\n");
```

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

```
    printf("Interchanging with using the temporary variable\n");
```

```
    temp=a;
```

```
    a=b;
```

```
    b=temp;
```

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

```
    printf("Interchanging without using the temporary variable\n");
```

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

```
    a=b - a;
```

```
        b = b -a;  
  
        printf("A =%d B=%d\n",a,b);  
  
    }  
  
/* END OF PROGRAM*/
```

Output:

Input values for a & b:

A=6 B = 4

## **BLOCK - III**



## **CONTROL STRUCTURES**

### **BLOCK INTRODUCTION**

Here we learn about the control structures of C. The compiler normally takes the program line by line and executes them in a sequence ( one after another ). But this may not always be the case. Based on certain conditions existing in the data, we may want to change the data - Then we use an If statements. In a limiting case, you may need to choose one out several possible options - by using a Switch statements.

Sometimes you also need to repeat the same set of statements repeatedly. Such statements are called loops. The number of times you do this may be or may not be known at the time of writing a program. This will lead us to for, while and DO While loops.

## CONTROL STRUCTURES IN 'C'

So far, we have only seen a program as a sequence of instructions the program beginning at the first line & ends in the last. However, such simple structures do not always exist in practice. Depending on the situation, we may have to skip certain instructions, repeat certain instructions etc.. Such facilities are provided by the control structures. Basically there are two types of very popular structures. One will allow us to make decisions while the other will make repeated executions of a set of instructions possible. We shall see them one after another.

The if statement :

The basic structure of the statement is

If ( expression)

Program statement

Next statement

i.e if the expression inside the parenthesis is satisfied, then the program statement is executed & then next statement is executed. If it is false, the program statement is skipped, but the next statement is executed. A non programming example could be like this.

If ( the weather cold)

I shall wear woolen clothes

I go for a walk.

i.e if the weather is cold ( expression is true) I shall wear woolen clothes ( program statement). (then) I go for a walk ( next statement).

If the weather is not cold(expression false), I go for a walk ( skip the program statement, go to a next statement) we shall see a more academic example in the following program.

```
/* CALCULATE THE ABSOLUTE VALUE OF AN INTEGER */
```

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

```
main ()
```

```
{
```

```
    int number;
```

```
    printf (" Type in your number: ");
```

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

```
    if ( number <0)
```

```
        number = -number;
```

```
    printf(" The absolute value is %d\n",number);
```

```
}
```

**Output:** Type in your number: -100

The absolute value is 100

**Output(Re-run)**

Type in your number: 2000

The absolute value is 2000

A more frequently used version of decision is

If (condition) then { statement 1 }

Else

{ statement 2 ) type of statements

i.e if the condition is satisfied statement 1 is executed if it false statement 2 is executed.

In either case the statement next to statement 2 is executed.

( Note that these are direct implementations of decision boxes in the chapter on flow charts)

One other point to not is that statement1 and statement2 need not be single statements, but they can be a group of statements included between { } )

We shall see some programs that use such control statements.

### **Write a program to calculate tax**

Solution:

**Algorithm:**

1. Enter pay and status
2. Check for status, if it results true compute tax with 20%
3. Print the tax.

**/\* PROGRAM TO TEST IF...ELSE STATEMENT\*/**

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

```
main()
```

```
{
    char status;
    float tax,pay;
    printf("Enter the payment:\n");
    scanf("%f",&pay);
    printf("Enter status\n");
    scanf("%c",&status);
    if (status == 's')
```

```

        tax=0.20 * pay;

    else

        tax=0.14*pay;

    printf("Tax=%f",tax);

}

/* PROGRAM TO DETERMINE IF A NUMBER IS EVEN OR ODD*/

#include<stdio.h>

main ()

{

    int number_to_test, reminder;

    printf ("Enter your number to be tested.: ");

    scanf ("%d", &number_to_test);

    reminder = number_to_test %2;

    if ( reminder==0)

        printf (" The number is even.\n");

    if (reminder !=0)

        printf (" The number is off.\n");

}

```

Output:

Enter your number to be tested: 2455

The number is odd.

**Output (Re-run):**

Enter your number to be tested: 1210

The number is even.

**/\* THIS PROGRAM DETERMINES IF A YEAR IS A LEAP YEAR\*/**

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

```
main ()
```

```
{
```

```
    int year, rem_4,rem_100,rem_400;
```

```
    printf("Enter the year to be tested:");
```

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

```
    rem_4 = year % 4;
```

```
    rem_100 = year % 100;
```

```
    rem_400 = year % 400;
```

```
    if ( ( rem_4 ==0 && rem_100 !=0) || rem_400 == 0 )
```

```
        printf (" It's a leap year.\n");
```

```
    else
```

```
        printf ("Nope, it's not a leap year.\n");
```

```
}
```

**Output:**

Enter the year to be tested: 1955

Nope, it's not a leap year.

**Output ( Re-run)**

Enter the year to be tested: 2000

It's a leap year.

**Output(Re-run)**

Enter the year to be tested: 1800

Nope, it's not a leap year

**/\* PROGRAM TO EVALUATE SIMPLE EXPRESSION OF THE FORM NUMBER  
OPERATOR NUMBER \*/**

**#include<stdio.h>**

**main ()**

**{**

float value1, value2;

char operator;

printf ("Type in your expression.\n");

scanf ("%f %c %f",&value1, & operator, & value2);

if ( operator == '+')

printf("%.2f\n",value1 + value2);

else if ( operator == '-')

printf("%.2f\n",value1 – value2);

else if ( operator == '\*')

printf("%.2f\n",value1 \* value2);

else if (operator == '/')

```
printf("%.2f\n",value1/value2);

}
```

Output:

Type in your expression.

123.5 + 59.3

182.80

**Output (Re-run)**

198.7 / 26

7.64

**Output ( Re-run)**

89.3 \* 2.5

223.25

**/\* THIS PROGRAM FINDS THE LARGEST OF THE 3 GIVEN NUMBERS USING A NESTED IF CONSTRUCT.\*/**

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

```
main ()
```

```
{
    int num1, num2, num3, max;
    printf("Enter 3 integer number:");
    scanf("%d %d %d",&num1, &num2, &num3);
    max = num3;
    if ( num1 > num2)
    {
```



```

    if (num1 > num3)
        max = num1;
    }
    else if (num2 > num3)
        max = num2;

    printf("The given number are %3d, %3d,and %3d\n",num1,num2,num3)

    printf("The largest number = %3d\n",max);

}

```

Output:

Enter 3 integer numbers: 5 87 12

The given numbers are 5, 87, and 12

The largest number = 87

### **The switch statement:**

When there are a number of else alternatives as above, way of representing is by the switch statement.

The general format of a switch statement is

```

    Switch (expression)
    {
case value1:
    program statement
    program statement
    .....
    break;

```

```
case value2:
```

```
    program statement
```

```
    program statement
```

```
    .....
```

```
    break;
```

```
    .....
```

```
case value'n':
```

```
    program statement
```

```
    program statement
```

```
    .....
```

```
    break;
```

```
default:
```

```
    program statement
```

```
    program statement
```

```
    .....
```

```
    break;
```

```
}
```

```
/* program to evaluate simple expression of the form value operator value */
```

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

```
main()
```

```
{
```

```
    float value1, value2;
```

```
    char operator;
```

```
    printf("Type in your expression. \n");
```

```
    scanf ("%f %c %f",&value1,&operator,&value2);
```

```
    switch(operator)
```

```
{  
    case '+':  
        printf("%.2f \n", value1 + value2);  
        break;  
    case '-':  
        printf("%.2f \n", value1 - value2);  
        break;  
    case '*':  
        printf("%.2f \n", value1 * value2);  
        break;  
    case '/':  
        if(value2 == 0)  
            printf("division by zero. \n");  
        else  
            printf("%.2f \n", value1 / value2);  
        break;  
    default:  
        printf("Unknown Operator \n");  
        break;  
}  
}
```

## LOOPS

The other type of control structures that we need are loops. Quite often, a set of instructions will have to be repeated again & again. For example, if you are calculating salary of 1000 employees, the portion of the program pertaining to the salary of employees will have to be repeated 1000 times, each time with a different set of data. The easiest way to do it is to start some sort of a counter, say *i*, to zero; each time one set of instructions are completed, the counter is increased by one and when it reaches 1000, we have to stop the repetitions. This can be done by the programmer also, but C provides special construct to do this.

In certain other cases, we will not be sure as to how many times the repetitions are to be done, but we have to continue till some conditions are satisfied - like all records getting exhausted or as long as some employees remain etc.. C provides facilities for such type of loops also.

We shall see them one after another.

**The for loop:** This is the simplest form of loops, where you know before hand how many repetitions ( " iterations" in computer terminology ) are to be done. Like the case of 1000 employees or 100 students of a class etc..

The format is

```
for (variable = initial ; variable = how long value to continue; amount of increment )
{
    Lines to be repeated
}
```

It is obvious that the initial value of variable need not always be 0 or 1; it can be anything. Similarly after each operation, you need not increment by 1. It can be 2,3 ... anything, even negative you want to count backwards). Note that you are only specifying the method of incrementing, the actual incrementing is done by the C control. It sets the variable

to the initial value, after each iteration, increments suitably, checks whether the terminating condition is met with, if not repeats the operations. Of course, it is the duty of the programmer to ensure that at some stage, the loop terminates - if the terminating condition never emerges out of the successive increments the program will go into an infinite loop.

Let us see a few examples for the **for loop**

Program:

Suppose we want to find the average of N given numbers. Obviously we input the numbers one after the other, add them into a sum and then divide by N. Let us also presume that N is also given at run time.

```
#include <stdio.h>

#include<math.h>

main()
{
    int n,i,num,sum=0;

    float avg;

    /* N is the number of numbers, num is the number ( read one at a time) &
    sum is the total*/

    printf("input how many numbers are there\n");
    scanf("%d",&n);
    for (i=1; i<n; i=i+1)
    {
        printf("input the next number\n");
        scanf("%d",&num);
```

```

        sum=sum+num;
    }

    avg=sum/n;

    printf("the average of the numbers is:\n %f",avg);
}

```

The steps are fairly simple to understand.

- a) The computer prints on the terminal, Input how many numbers are there.
- b) If, say, 10 numbers are to be added, the number 10 is given as the input.
- c) Then, inside the loop, the system keeps asking ( 10 times in this case) input the next number.
- d) Every time, the next number of the set of numbers for which average is to be calculated is inputted.
- e) In the end, the sum & average are calculated.

The point to be noted is that the loop keeps incrementing itself as long as  $i < \text{or} = n$  automatically and stops once  $n$  become greater than  $n$  i.e. we technically say there were " $n$  iterations"

Program:

Now we will look one more simple but popular program to find the factorial of a number. If the number is, say, 5 its factorial is  $5 \times 4 \times 3 \times 2 \times 1$  i.e the product numbers from 1 to  $n$ .

```

#include<stdio.h>

#include <math.h>

main()
{

    int n,prod=0;

```

```

printf("input the number\n");

scanf("%d\n",&n);

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

{

    prod=(prod*i);

}

printf("The factorial is\n,%d",prod);

}

```

of course, the initial value of the loop need not always start with 1.

Program:

Let us illustrate by a simple case. We want to find the sum of the first 50 even numbers.

The program looks something like this

```

#include<stdio.h>

main()

{

    int i,sum=0;

    for(i=2; i<=100; ++i)

    {

        sum=sum+i;

    }

    printf("The sum of the numbers is,%d\n",sum);

}

```

## **BLOCK IV**

### **ARRAYS**

#### **BLOCK INTRODUCTION**

Whenever we have to handle data which are interrelated - like the marks scored by the different students of a class in a subject, the salaries of the employees of a dept etc - it is worthwhile to store and operate them under a single name. This leads us to the concept of arrays. Arrays need not be the one dimensional cases as above. They can have 2/more dimensions - like the marks scored by the different students if a class in different subjects, the salary details like BASIC,DA,HRA & CCA of the employees of a dept etc.. In this block, we learn to manipulate them.

Also, now that we have gained some confidence over the language, we write a large number of programs.



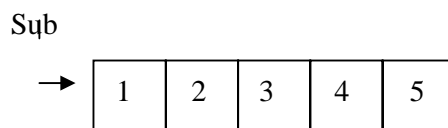
## ARRAYS

We have seen how to create variables - integer, float or character types, each variable is given a unique name and you can store one item under that name. Often, however, we need several similar items to be grouped under a single name. For example, marks scored by a student in different subjects, the salaries of employees of a particular department etc.. Ofcourse, if a student has 5 subjects you can declare 5 different variables - say sub1, sub2 .. sub5 and operate on them. But it is better if we can give a common name to them so that we can know that they are related to each other. Further, what if there are 100 employees in a dept? Declaring 100 different variables explicitly is quite difficult. In such cases, a data structure called "array" is used. ( Incidentally, a data structure can be approximately thought of as the method of grouping memory location in a particular form )

Suppose we consider individual variables like this



Then we can group them under a common name sub as follows:



Now all of them have a common name called sub and they can be individually addressed by their index number 1,2,3,4 & 5. So, the marks scored in subject 1 is stored in sub[1], that in subject 2 is sub[2] etc..

In C, you can declare arrays in the beginning just like declaring other variables

Examples `int sub[0]`, `float salary[50]` etc..

This means that in the array called `sub`, 5 different integer values can be stored, under `salary` 50 different float values are stored. To access each of them, their specific index number is to be given. However, in C, the first element index is 0, the second one as 1 etc..

For example `sub[0]` stores the marks of the first subject.... `Sub[4]` stores the marks of the fifth subject.

Just to familiarise ourselves with the way arrays operate, we see the following examples:

`/* to input the numbers between 1 to 10 */`

`Main()`

```
{
    Int rating-counters[11],i,response;
    for (i=1; i<=10;++i)
        rating_counters [i] =0;
    printf ("Enter your responses\n");
    for (i=1; i<=20; i++)
    {
        scanf ("%d",&response);
        if (response<1 || response >10)
            else
                ++rating_counters[response];
        printf("\n\nRating    Number of Responses\n");
        printf("_____ \n");
        for (i=1; i<=10; ++i)
            printf("%4d%14d\n",rating_counters[i]);
    }
}
```

OUTPUT:

Enter your responses

6

5

8

3

9

6

5

7

15

Bad response: 15

5

5

1

7

4

10

5

5

6

8

9

6

Rating    Number of Responses

-----

1	1
2	0
3	1
4	1
5	6
6	4
7	2
8	2
9	2
10	1

**/\* program to generate the first 15 fibonacci numbers \*/**

main()

{

    int fibonacci[15],i;

    fibonacci[0] = 0;       /\* by definition \*/

    fibonacci[1] =1;       /\*    - " - \*/

    for (i=2; i<15; ++i)

        fibonacci[i] = fibonacci [i-2] + fibonacci [i-1];

    for (i=0; i<15; ++i)

        printf("%d\n",fibonacci[i]);

}

**OUTPUT:**

0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
377

**Write a program to compute all prime numbers between 2 and 50**

```
main()
{
    int p,is_prime,i,primes[50],prime_index = 2;
    primes[0] =2;
    primes[1] =3;
    for (p=5; p<=50; p=p+2)
        is_prime =1;
```

```

    for (i=1; is_prime && p/primes[i] >=primes[i]; ++i)
        if (p%primes[i] == 0)
            is_prime = 0;
    if (is_prime)
    {
        primes [prime_index] =p;
        ++prime_index;
    }
}

for (i=0; i<prime_index; ++i)
    printf("%d",primes[i]);
printf("\n");
}

```

**OUTPUT:**

```

2    3    5    7    11   13   17   19   23   29   31   37
41   43   47

```

It is also possible to initialize the array elements at the time of declaration itself

```
/* write a program to print array values */
```

```
Main()
```

```
{
```

```
    static int array_values[10] = { 0,1,4,9,16};
```

```
    int i;
```

```

        for (i=5; i<10; ++i)

            array_values[i] = i * i;

        for(i=0; i<10; ++i)

            printf("array_values[%d]=%d\n",i,array_values[i]);

    }

```

OUTPUT:

Array\_values[0] = 0

Array\_values[1] = 1

Array\_values[2] = 4

Array\_values[3] = 9

Array\_values[4] = 16

Array\_values[5] = 25

Array\_values[6] = 36

Array\_values[7] = 49

Array\_values[8] = 64

Array\_values[9] = 81

Arrays need not be only single dimensional. They can be 2,3... dimensional also

Consider the very popular concept of matrices

A matrix can be thought as a set of numbers as follows

$$\begin{pmatrix}
 10 & 5 & 3 & 7 & 9 \\
 9 & 6 & 4 & 7 & 9 \\
 5 & 4 & -8 & 6 & 7 \\
 4 & 1 & 7 & 6 & -5
 \end{pmatrix}$$

This is a 4 \* 5 matrix, with 4 rows and 5 columns.

This can be stored in a integer variables M as follows

```
Int M[4] [5]
```

Because M has 4 rows (0,1,2,3) and 5 columns(0,1,2,3,4). As before, the array can also be initialised at declaration time itself.

```
Int M[4][5]= {
    { 10,  5,   3,   7,   9},
    { 9,   6,   4,   7,   9},
    {5,   4,  -8,   6,   7},
    {4,   1,   7,   6,  -5}
};
```

Ofcourse, in all intilisations, not all the elements need to be initialised, you can only intialise those elements that you need and leave out other elements. They will be initialised to 0.

We now see several other programs using arrays.

### **WRITE A PROGRAM TO FIND LARGEST, SMALLEST, SUM & AVERAGE OF FIVE NUMBERS**

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

```
main()
```

```
{
```

```
    int x[5];
```

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



```

float ave;

for (i=0; i<5; i++)
{
    printf ("\n enter number%d:",i+1);
    scanf ("%f",&x[i];
}

/* To print largest and smallest number */

large=small=x[0];
for (i=0; i<5; i++)
{
    if(x[i]>large)
        large=x[i];
    if(x[i]<small)
        small=x[i];
}

printf("\n The largest no is %d",large);
printf("\nThe smallest no. is %d",small);
/*To find the sum and average*/
for (i=0; i<5; i++)
    sum+=x[i];
ave=sum/5.0
printf("\nthe sum is %d",sum);
printf("\nthe average is:%5.2f",ave);
}

```

**OUTPUT:**

Enter number1:50

Enter number2:30

Enter number3:40

Enter number4:10

Enter number5:20

The largest No. is 50

The smallest No. is 10

The sum is 150

The average is 30.00

**WRITE A PROGRAM TO SORT NUMBERS IN ASCENDING ORDER USING  
BUBBLE –SORT**

```
#include<stdio.h>

main()
{
    int x[10], i, j, n,temp;

    printf("\n ENTER NUMBER OF ITEMS:");

    scanf("%d",&n);

    /*GET THE ARRAY FROM THE USER*/

    printf("\n Enter the numbers:\n");

    for (i=0;i<n;i++)

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

    /*SORT THE ARRAY */
```

```

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

/*TO PRINT THE ARRAY*/

printf("\n The Numbers in ascending order:\n");

for(i=0;i<n;i++)

printf("%d\n",x[i]);

}

```

**OUTPUT :**

Enter the numbers:

5      4      3      2      1

The Numbers in ascending order:

1      2      3      4      5

**WRITE A PROGRAM TO ADD TWO MATRICES**

```
#include<stdio.h>

main()
{
    int a[5][5],b[5][5],c[5][5];

    int i,j,m,n;

    /*GET THE MATRIX FROM THE USER*/

    printf("enter matrix A:\n");

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

        scanf("%d",&a[i][j]);

    printf("enter matrix B:\n");

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

        scanf("%d",&b[i][j]);

    /*ADD THE MATRICES*/

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

        c[i][j]=a[i][j] + b[i][j];

    /*TO PRINT THE SUM*/

    printf("\n THE SUM OF MATRICES:\n");

    for(i=0;i<m;i++)

        {
```

```

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

```

### OUTPUT :

Enter matrix A:

```

1      1      1
1      1      1
1      1      1

```

Enter matrix B:

```

2      2      2
2      2      2
2      2      2

```

THE SUM OF MATRICES:

```

3      3      3
3      3      3
3      3      3

```

### WRITE A PROGRAM FOR THE SUBTRACTION OF MATRICES

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

```
main()
```

```
{
```

```
    int a[5][5],b[5][5],c[5][5];
```

```

int i,j,m,n;

/*GET THE MATRIX FROM THE USER*/

printf("enter matrix A:\n");

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

    scanf("%d",&a[i][j]);

printf("enter matrix B:\n");

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

    scanf("%d",&b[i][j]);

/*SUBTRACTION THE MATRICES*/

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

    c[i][j]=a[i][j] - b[i][j];

/*TO PRINT THE DIFFERENCE*/

printf("\n THE DIFFERENCE OF MATRICES:\n");

for(i=0;i<m;i++)

{

    for(j=0;j<n;j++)

        printf("%d\t",c[i][j]);

    printf("\n");

}

}

```

**OUTPUT:**

Enter matrix A:

```
3    3    3
3    3    3
3    3    3
```

**Enter matrix B:**

```
2    2    2
2    2    2
2    2    2
```

**THE DIFFERENCE OF MATRICES:**

```
1    1    1
1    1    1
1    1    1
```

**WRITE A PROGRAM FOR FINDING THE PRODUCT OF MATRICES**

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

```
main()
```

```
{
```

```
    int a[5][5],b[5][5],c[5][5];
```

```
    int i,j,k,m,n,p,q;
```

```
    printf("\n ENTER ORDER OF MATRIX A:");
```

```
    scanf("%d%d",&m,&n);
```

```
    printf("\n ENTER ORDER OF MATRIX B:");
```

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

```
    if (n!=p)
```

```

{
    printf("\n\tCANNOT MULTIPLY");
    exit();
}

/*GET THE MATRIX FROM THE USER*/

printf("enter matrix A:\n");
for(i=0;i<m;i++)
for(j=0;j<n;j++)
    scanf("%d",&a[i][j]);

printf("enter matrix B:\n");
for(j=0;j<p;j++)
for(k=0;k<q;k++)
    scanf("%d",&b[j][k]);

/*PRODUCT OF MATRICES*/

for(i=0;i<m;i++)
{
    for(k=0;k<q;k++)
    {
        c[i][k]=0;
        for(j=0;j<n;j++)
            c[i][k]+=a[i][j] * b[j][k];
    }
}

```



```

/*TO PRINT THE PRODUCT OF MATRICES*/

printf("\n THE PRODUCT OF MATRICES:\n");

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

```

**OUTPUT:**

Enter matrix A:

```

2      2      2
2      2      2

```

Enter matrix B:

```

3      3
3      3
3      3

```

THE PRODUCT OF MATRICES:

```

18     18
18     18

```

**WRITE A PROGRAM TO READ A LINE OF TEXT AND COUNT THE NUMBER OF VOWELS, CONSONANTS, DIGITS AND BLANK SPACES.**

```
#include<stdio.h>

main()
{
    char line[50],c;

    int v=0,con=0,d=0,ws=0,count=0;

    printf("\n Enter a line of text:\n");

    scanf("%[^\\n]",line);

    while((c=toupper(line[count] )) != '\\0')
    {
        if(c== 'A' || c=='E' || c=='I' || c== 'U')

            v++;

        else if(c>='A' && c<= 'Z')

            con++;

        else if(c>= '0' && c<= '9')

            d++;

        elseif(c== ' ' || c=='\\t')

            ws++;

        ++count;
    }

    printf("\n Number of Vowels:%d",v)

    printf("\n Number of Consonants: %d",con);
```

```

printf("\n Number of Digits:%d",d);

printf("\n Number of Blank spaces:%d",ws);

}

```

**OUTPUT:**

Enter a line of text:

1 This is to test a line of text.

Number of Vowels: 9

Number of consonants: 14

Number of Digits:1

Number of Blank spaces:8

**WRITE A PROGRAM TO FIND THE TRANSPOSE OF A MATRIX**

```

#include<stdio.h>

main()
{
    int a[5][5];
    int i,j,m,n;
    printf("\n ENTER ORDER OF MATRIX");
    scanf("%d%d",&m,&n);
    /*GET THE MATRIX FROM THE USER */
    printf("Enter the matrix:\n");
    for(i=0;i<m;i++)
    for(j=0;j<n;j++)
        scanf("%d",&a[i][j]);
}

```

```

/*TRANSPOSE OF MATRIX*/

printf("\n TRANSPOSE OF MATRIX IS:\n");

for(i=0;i<n;i++)

{

    for(j=0;j<m;j++)

        printf("%d\t",a[i][j]);

    printf("\n");

}

}

```

**OUTPUT:****Enter matrix :**

```

1      2      3
4      5      6
7      8      9

```

**TRANSPOSE OF MATRIX IS:**

```

1      4      7
2      5      8
3      6      9

```

**WRITE A PROGRAM TO FIND WHETHER THE MATRIX IS SYMMETRICAL**

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

```
main()
```

```
{
```

```
    int a[5][5];
```

```

int i,j,k,m,n;

printf("\n ENTER ORDER OF MATRIX :");

scanf("%d%d",&m,&n);

if (m!=n)

{

    printf("\n***WARNING***\n ENTER  SQUARE MATRIX ONLY");

    exit();

}

/*GET THE MATRIX FROM THE USER*/

printf("enter matrix :\n");

for(i=0;i<m;i++)

for(j=0;j<n;j++)

    scanf("%d",&a[i][j]);

/*CHECK FOR SUMMETRY*/

for(i=0;i<n;i++)

for(j=0;j<m;j++)

if(a[i][j]!=a[j][i])

{

    printf("\n MATRIX IS NOT SYMMETRIC\n");

    exit(0);

}

printf("\nMATRIX IS SYMMETRIC\n");

}

```

**OUTPUT :**

Enter matrix :

```
1      2      3
2      1      4
3      4      1
```

MATRIX IS SYMMETRIC

**WRITE A PROGRAM TO FIND THE LARGEST ELEMENT IN AN ARRAY AND POSITION OF IT'S OCCURRENCE**

```
main()
{
    int A[10];

    int largest,position;

    int num,i;

    printf("\nEnter the number of elements in the array\n");

    scanf("%d",&num);

    printf("\nEnter the array elements\n");

    for (i=0; i<num; i++)

        scanf("%d",&A[i]);

    largest = A[0];

    for(i=1; i<num; i++)

    {

        if (largest < A[i])

            {
```

```

        largest = A[i];
        position = i;
    }

}

printf("The largest element of the array is %d",largest);

printf("and it is in position %d",position);

}

```

**OUTPUT :**

Enter the number of elements in the array

6

Enter the array elements

1      6      7      3      9      7

The largest element of the array is 9 and it is in position 4

**WRITE A PROGRAM FOR LINEAR SEARCH**

```

main()
{
    int A[100],item, num,i;

    /* read in number of elements*/

    printf("\nEnter the number of elements \n");

    scanf("%d",&num);

    /*read in array elements*/

    printf("\nEnter the array elements\n");

    for (i=0; i<num; i++)

```

```

scanf("%d",&A[i] );

/*read in item to be searched */

printf("\nEnter the item to be searched\n");

scanf("%d",&item);

/*check for the existence of the item in the array */

for(i=1; i<num; i++)

if (item== A[i] )

{

    printf("\n Item %d is found at position%d",item,I);

    exit(); /* quit the program if search is successful*/

}

printf("\n Item %dnot found in array\n",item);

}

```

**OUTPUT:**

Enter the number of elements

7

Enter the array elements

4      9      2      8      3      7      1

Enter the item to be searched

8

Item 8 is found at position 3



**WRITE A PROGRAM FOR BINARY SEARCH**

```

main()
{
    int low,high,middle,i,item,n,a[20];

    /*read in the element to be searched */

    printf("\n The element to be searched =");

    scanf("%d",&item);

    /*read in the number of elements in the array*/

    printf("Enter the number of elements in the array(<=20):");

    scanf("%d",&n);

    /*read in the array elements */

    printf("\nPlease enter the array element (sorted one) :\n");

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

        scanf("%d",&a[i] );

    low =1;

    high = n; /*initializing low and high index values */

    do

    {

        middle = (low + high)/2; /* initializing middle index*/

        if(item < a[middle] )

            high = middle -1;

        else

```

```

        if(item > a[middle] )

            low = middle + 1;

    }

    while(item != a[middle] && (low <=high));

    /*print the result*/

    if(item == a[middle] )

        printf("%d is found at position %d,item,middle);

    else

        printf("Item not found in the given array\n");

}

```

**OUTPUT:**

The element to be searched =6

Enter the number of elements in the array (<=20):8

Please enter the array elements (sorted one):

3      6      8      10      11      12      13      60

6 is found at position 2

## **UNIT - II**

### **UNIT INTRODUCTION**

In this unit, you are introduced to some of the fairly advance, but still frequently used concepts of functions, structures and files.

Quite often, the program will not be a single entity. It may have been written in different "modules", may be even by different people and different types. But, still, we should be able to combine these modules into a program and execute them. Obviously, such a scenario demands some sort of discipline in writing each of these modules. Each of them can be written as a "function" and can be linked to one another and used. We learn to write and use such functions.

Structures are complex data structures - like the marks scored by a student in different subjects or the personal details of a credit card holder. To hold such (often non-homogeneous) data. We define the concept of structures.

So far, we have been using Scanf & Printf as input & output. I.e whenever we need to input data, you do so using the keyboard and watch the output on the screen. But this cannot happen always - especially when the data being handled is huge or when we need it to be presented for later use. Alternatively, we can save such data in files and use them whenever we need them. So, we learn how to create and manage such files.

## **BLOCK I**

# **FUNCTIONS**

## **Block Introduction**

There are a number of situations wherein the program is too complex to be visualized as one single unit. We would rather divide it into a number of smaller tasks, write small programs to each of them & combine them. In fact each of them may be written by different people, using their own (local) variable names etc.. Each such unit can be called a function. They may all be linked and executed.

Such a scenario has two more advantages. One- if a program is to do the same function repeatedly - say it has to arrange numbers in ascending order in 5 different places - you need not include the code in all those places - just call the function from these five places.

Secondly - you need not even write the functions in the first place. If somebody in some other context has written the function - just use it. A concept used in the context of library functions.

In this block, you are introduced to the concept of such functions - how to write them & use them.

## **FUNCTIONS :**

A function is often defined as a section of a program performing a specific job. In fact the concept of functions (which were originally a subset of a concept called sub programs) came up because of the following argument.

Imagine a program wherein a set of operations are to be done often. (Not continuously N times or so, if that were the case, we can use loops). Instead of inserting the code for these operation at so many places, you write a separate program segment and compile it separately. As many times as you need it, you keep “calling” the segment, to get

the result. The separate program segment is called a function and the program that calls it is called a “main program”.

‘C’ went one step further, it divided the entire concept of programming to a combination of functions. The scanf(), printf(), main() etc.. that we have seen are all functions. C provides a lot of library functions, in addition you should be able to write your own functions & use them.

```
/* This function finds the Greatest Common Divisor of two nonnegative integer values */
gcd (u,v)
int u,v;
{
    int temp;
    printf("The gcd of %d and %d is",u,v);
    while( v!=0)
    {
        temp=u%v;
        u=v;
        v=temp;
    }
    printf("%d\n",u);
}
```

```
main()
{
    gcd(150,35);
    gcd(1026,405);
    gcd(83,240);
}
```

**OUTPUT:**

The gcd of 150 and 35 is 5

The gcd of 1026 and 405 is 27

The gcd of 83 and 240 is 1

/\* This function finds the Greatest Common Divisor of two nonnegative integer values and returns the result \*/

```
gcd (u,v)
int u,v;
{
    int temp;
    while( v!=0)
    {
        temp=u%v;
        u=v;
        v=temp;
    }
    return (u);
}
```

```

}

main()
{
    int result;

    result=gcd(150,35);

    printf("The gcd of 150 and 35 is %d\n", result);

    result = gcd(1026,405);

    printf ("The gcd of 1026 and 405 is %d\n",result);

    printf("The gcd of 83 and 240 is %d\n",gcd(83,240));

}

```

**OUTPUT:**

The gcd of 150 and 35 is 5

The gcd of 1026 and 405 is 27

The gcd of 83 and 240 is 1

```
/* Function to calculate the absolute value */
```

```
float absolute_value (x)
```

```
float x;
```

```

{
    if (x < 0)
        x=-x;
}

```

```

        return(x);
    }
main()
{
    float f1 = -15.5, f2=20.0, f3= -5,0;

    int  i1= -716;

    float result;

    result=absolute_value(f1);

    printf("result = %.2f\n", result);

    printf("f1 = %.2f\n",f1);

    result=absolute_value (f2) + absolute_value (f3);

    printf("result = %.2f\n",result);


    result=absolute_value (float) i1 );

    printf("%.2f\n", absolute_value (-6.0) /4);

}

```

**OUTPUT:**

Result = 15.50

F1= -15.50

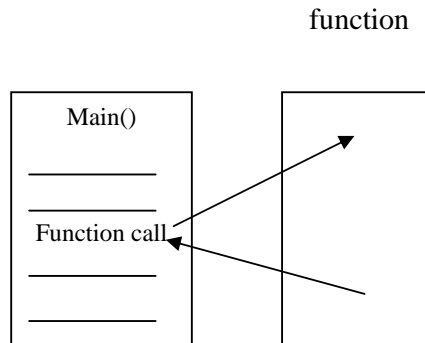
Result = 25.00

Result = 716.00

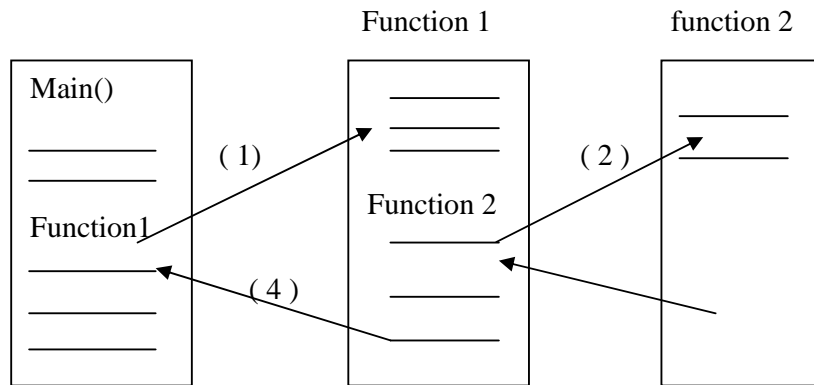
1.50



In each of these cases, the concept is very simple. Each function is given a name and a list of parameters. Whenever you want the function to be called, refer to the function name with a suitable list of arguments. The next line of program executed will be not from the main execution, the control comes back to the main, to the place where it was called



But there is no restriction that the main only should call a function. A function can call another, that another and so on. See figure below:



The number indicate the sequence of control flow:

```
/* Function to calculate the absolute value of a number */
```

```
float absolute_value (x)
```

```
float x;
```

```

{
    if (x < 0)

        x= -x;

    return (x);
}

/* Function to compute the square root of a number */

float square_root (x)

float x;

{
    float epsilon = .0001;
    float guess = 1.0;
    while (absolute_value (guess * guess -x) >=epsilon )
        guess = ( x/guess + guess ) / 2.0;
    return (guess);
}

main()
{
    printf("square_root (2.0) = %f\n", square_root (2.0);
    printf("square_root (144.0) = %f\n", square_root(144.0);
    printf("square_root (17.5) = %f\n",square_root (17.5);
}

```

**OUTPUT:**

Square\_root (2.0) = 1.414216

Square\_root (144.0) = 12.000000

Square\_root (17.5) = 4.183300

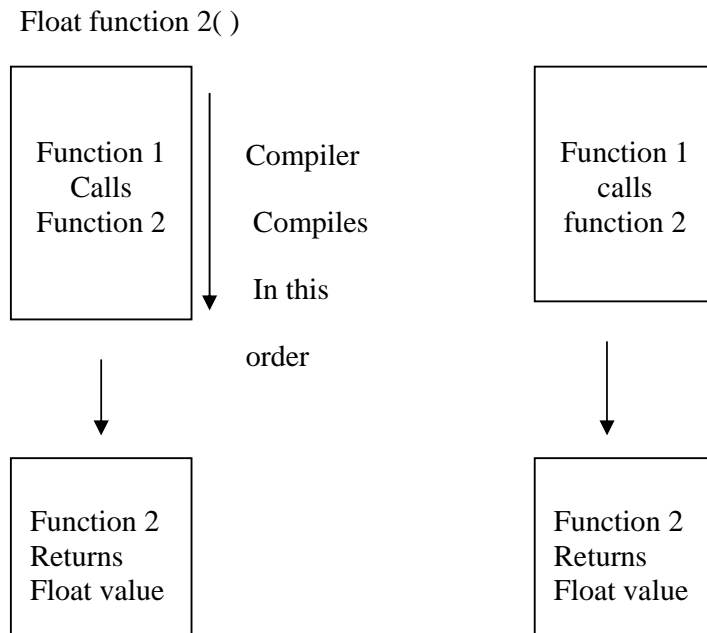
Now we look into certain terminology encountered w.r.f functions each function is given a name. It can be any valid variable name. This is called a function declaration. Also zero or more number of arguments to the function. They are similar to variables, but are used to transfer values to the function. Often they are called formal parameters. Formal parameters will also have to be declared inside the function.

In addition several variables are needed for the function. They are declared just like any other variable. They are called automatic local variables, because (i) They are local: their effect is limited to the function. (ii) They are automatic since they are automatically created whenever the function is called. Also their value can be accessed only inside the function, not from any other function ( some authors also use "auto" to indicate that they are automatically created).

Again each function should also return back one/more values, the statement `return (expression)` does the job. However, the type of value that the return expression returns (int, char or float) will also have to be indicated in the name of the function (like `int gcd()`, `float convert()` etc.. ). If, however, no such declaration is made, the function is expressed to return a int value.

If the function does not return any value, it is called a void function and should be indicated so. (`void main()` for example)

However, there is a small problem, since any function can call any other function, it is also possible that a function which is not yet defined can be used. (see example). In that case the compiler presumes that the called function returns only int values, even if return float or even if it void.



How does the compiler know

While compiling, that function 2

Returns a float?

Declare the function 2

in the beginning

To overcome this, it is desirable to declare before, how that function 2 return float  
(as shown)

```

/* Function to find the minimum in an array */
int minimum (values)
int values [10];
{
    int minimum_value, i;
    minimum_value = values [0];
    for ( i=1; i<10; ++i)
        if (values [i] < minimum_value)
            minimum_value = values[i];
    return (minimum_value);
}
  
```

```
}  
  
main()  
{  
  
    int scores[10], i, minimum_score;  
  
    printf("enter 10 scores \n");  
  
    for (i =0; i < 10; ++i)  
  
        scanf("%d",&scores[i]);  
  
    minimum_score = minimum (scores);  
  
    printf("\nMinimum score is %d\n",minimum_score);  
  
}
```

**OUTPUT:**

Enter 10 scores

69

97

65

87

69

86

78

67

92

90

Minimum score 65

## Sorting:

Sorting is a very popular activity in programming, which essentially arranges a set of given numbers in ascending/ descending order. One simple way is to simply keep finding the smallest number one after another and put them in another array ( using the previously defined method). However, more efficient method are available one of them is called the exchange sort method.

Here we begin comparing the first element with the second. If the first is smaller than second, we leave them as it is, otherwise, we "swap" then (i.e interchange them). Next compare the first element with the third. Again if the first one is smaller, leave it as it is, otherwise swap the first and third element. This way, if we keep comparing the first element with all other elements ( and keep swapping if necessary) at the end of the iterations, we end up with the first element being the smallest of elements.

Now, we can repeat the same with the second element and all other elements; third element and all other elements etc.. In the end, we will be left with an array in which the numbers are in ascending order.

This algorithm is called the exchange sort algorithm.

Step1: set i to 0

Step2: set j to i + 1

Step3: if  $a[i] > a[j]$  exchange their values

Step4: set j to j+1. If  $j < n$  goto step 3

Step5: set i to i+1 If  $i < n-1$  goto step 2

Step6: a is now sorted in ascending order.

`/* Sort an array of integers into ascending order */`

`void sort (a,n)`

`int a[];`

```

int n;
{
    int i,j,temp;

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

main()
{
    int i;

    static int array[16] = { 34,-5,6,0,12,100,56,22,44,-3,-9,12,17,22,6,11 };

    printf("The array before the store:\n");

    for (i=0; i< 16; ++i)

        printf("%d",array[i] );

    sort (array, 16);

    printf ("\n\n The array after the sort:\n");

    for ( i=0; i <16; ++i)

        printf("%d", array[i] );

    printf("\n");
}

```

```
}
```

### OUTPUT:

The array before the store:

34,-5,6,0,12,100,56,22,44,-3,-9,12,17,22,6,11

The array after the store:

-9 -5 -3 0 6 6 11 12 12 17 22 22 34 44 56 100

```
/* Function to multiply a 3 x 5 array by a scalar */
```

```
void scalar_multiply (matrix,scalar)
```

```
int matrix [3] [5] ;
```

```
int scalar;
```

```
{
```

```
    int row,column;
```

```
    for ( row = 0; row < 3; ++row )
```

```
        for ( column = 0; column < 5; ++column)
```

```
            matrix[row] [column] *=scalar;
```

```
}
```

```
void display_matrix (matrix)
```

```
int matrix [3] [5];
```

```
{
```

```
    int row,column;
```

```
    for (row =0; row < 3; ++row)
```

```
    {
```

```
        for ( column = 0; column < 5; ++column )
```



```

        printf("%5d", matrix[row][column] );

        printf("\n");
    }

}

main()
{
    static int sample_matrix[3][5] =

        {
            { 7,   16,   55,   13,   12 },
            { 12,  10,   52,   0,    7  },
            { -2,   1,    2,    4,    9  }

        };

    printf("Original matrix : \n");
    display_matrix (sample_matrix);
    scalar_multiply (sample_matrix, 2);
    printf("\nMultiplied by 2: \n");
    display_matrix (sample_matrix);
    scalar_multiply (sample_matrix, -1);
    printf ("\n Then multiplied by -1\n");
    display_matrix (sample_matrix);
}

```

**OUTPUT:**

Original matrix:

```
7      16      55      13      12
12     10     52      0       7
-2      1       2       4       9
```

Multiplied by 2:

```
14     32     110     26     24
24     20     104      0     14
-4      2      4       8     18
```

Then multiplied by -1:

```
-14    -32    -110    -26    -24
-24    -20    -104     0    -14
4       -2     -4      -8    -18
```

Before we proceed, we would introduce one more concept. Previously we talked about local variables (auto local variables). These variables cease to exist once that function is exited. So, in case we want their values to be available even after the function execution is complete, it can be done in two ways

- i) Make that variable Global. i.e define them before all functions. Then their vlues will be available from all the functions and they can be accessed from any of the functions.

Example:     #include<stdio.h.

```
Int a,b,c;
```

```
Main()
```

```
{
```

```
}
```

```
function()
{
    }
etc..
```

Here, a,b,c are global variables and can be accessed from the main as well as all other functions.

- ii) The second method is to call them static in the function in which it is used.

Example:

```
#include<stdio.h>

main()
{
    }

function1()
{
    static int a
    -   -   -   -
    -   -   -   -

}
```

Here, the scope of the variable a is only for the function1. I.e. it's values cannot be accessed from any other function. However, when we next time come back to function 1, the previously computed value of a is found to be retained.

## Recursive functions

C provides for a special type of function called Recursive function. Previously we have seen that in C any function can call any other function. The limiting case is can a function call itself? The answer is yes. A function can call itself. If a function repeatedly keeps calling itself ( until certain conditions are satisfied called terminating conditions), then such a function is called a recursive function.

Consider the following example:

The factorial of a number is defined as the product of all integers from 1 to that number.

For example the factorial of 4,

Represented as  $4! = 4 \times 3 \times 2 \times 1 = 24$

And  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

One method of evaluation is, of course, by a loop

```
for ( I=1; e<=n; ++I)
{
    fact= fact * I;
}
```

This method is the method of recursion.

There is another method

Now look at the previous example

5! Can be evaluated if we had known 4!

$5! = 5 \times 4!$

If we write the relevant numbers, we know that

$$4! = 4 \times 3!$$

$$3! = 3 \times 2! \text{ And}$$

$$2! = 2 \times 1!$$

But 1! Is known, it is 1.

So, if we work back wards, using the formula

$n! = n * (n-1)!$  Until we strike 1, then we can evaluate the factorial of any number n. This can be done, if a function, when called with an argument n, repeatedly calls itself with (n-1) until the value of (n-1) reaches 1. ( This value =1 is called the terminating condition).

The following program adequately illustrates the concept.

```
/* Recursive function to calculate the factorial of a positive integer */
```

```
long int factorial (n)
```

```
int n;
```

```
{
```

```
    long int result;
```

```
    if (n == 0)
```

```
        result = 1;
```

```
    else
```

```
        result = n * factorial ( n-1 );
```

```
    return (result);
```

```
}
```

```
main()
```

```
{
```

```
    int j;
```

```

    for ( j =0; j < 11; ++j)

        printf("%d! = %ld\n",j,factorial (j) );

}

```

**OUTPUT:**

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10!=3628800

```

Though in this case, the advantage of using recursive functions, may not be obvious at first sight, there are situations, when a recursive function reduces the complexity of program writing significantly.

```

/* program to find the first n entries in the fibonacci */

#include<stdio.h>

main(0

{

    int i,n;

```

```

int fibonacci(int,i);

/* read the number of entries */

printf("Enter no. of entries:");

scanf("%d",&n);

/* calculate and display the fibonacci series */

printf("The fibonacci series: \n");

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

    printf("%d\n",fibonacci(i) );

} /* end of main() */

int fibonacci(int x)    /* function to find the nth fibonacci no */

{

    if (( x== 1) || (x = = 2))

        return(1);

    else

        return(fibonacci(x-1) + fibonacci(x-2));

} /* end of fibonacci() */

```

**OUTPUT:**

Enter no.of entries: 9

The fibonacci series:

1

1

2

3

5

8

13

21

34

```
/* program to calculate the GCD of two numbers */
```

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

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

```
unsigned gcd(unsigned num1,unsigned num2);
```

```
main()
```

```
{
```

```
    unsigned num1,num2;
```

```
    printf("Enter two numbers whose GCD is to be found:");
```

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

```
    printf("GCD of %u and %u is = %u\n",num1,num2,gcd(num1,num2));
```

```
}
```

```
/* Function to calculate the GCD of two unsigned numbers */
```

```
unsigned gcd(unsigned num1, unsigned num2);
```

```
{
```

```
    unsigned num, gcdnum,i;
```

```
    num = num1 < num2 ? num1;
```

```
    for (i =1; i<=num; i++)
```



```

        if ((num1 % i == 0) && (num2 % i == 0))
            gcdnum = i;
        return gcdnum;
    }

```

**OUTPUT:**

Enter two numbers whose GCD is to be found: 25 10

GCD of 25 and 10 is = 5

Enter two numbers whose GCD is to be found: 47 10

GCD of 47 and 10 is = 1

Enter two numbers whose GCD is to be found: 16 64

GCD of 16 and 64 is = 16

```

/* Program to check the given integer prime or not */

```

```

#include <stdio.h>

```

```

#include<math.h>

```

```

int isprime(unsigned num);

```

```

main()

```

```

{

```

```

    int num;

```

```

    printf("\nEnter an integer:");

```

```

    scanf("%u",&num);

```

```

    if(num < 0)

```

```

        printf("Invalid entri\n");

```

```
else
{
if (isprime(num))
    printf("%d is a prime number\n",num);
else
    printf("%d is not prime number\n",num);
}
}

int isprime(unsigned num)
{
    unsigned index sq;
    if((num == 1) || (num == 2))
        return 1;
    sq = sqrt((double)num);
    for(index = 2; index <= sq; index++)
    {
        if (num % index == 0)
            return;
    }
    return 1;
}
```

**OUTPUT:**

Enter an integer 15

15 is not a prime number

Enter an integer: 29

29 is a prime number

Enter an integer: 397

397 is a prime number

```
/* Program to find prime numbers upto N */
```

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

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

```
int isprime(unsigned num);
```

```
main()
```

```
{
```

```
    unsigned index,pcnt=0;
```

```
    int max;
```

```
    printf("\nEnter integer N upto which primes are to be found:");
```

```
    scanf("%u",&max);
```

```
    if(max < 0)
```

```
    {
```

```
        printf("Prime nos. Upto %u:\n",max);
```

```
        for (index = 1; index <= max; index++)
```

```
            if (isprime(index))
```

```
            {
```

```

        printf("%u\n",index);

        pcnt++;

    }

    printf("\nNumber of primes = %u",pcnt);

}

}

```

/\*Function to find whether a given unsigned \*/

/\* integer is a prime number \*/

int isprime(unsigned num)

```

{

    unsigned index sq;

    if((num == 1) || (num == 2))

        return 1;

    sq = sqrt((double)num);

    for (index = 2; index <=sq; index++)

    {

        if num%index == 0)

            return 0;

    }

    return 1;

}

```

Enter integer N upto which primes are to be found: 10

Prime nos. Upto 10;

1

2

3

5

7

Number of primes = 5

```
/* Program to find  $n!/(n-r)!$  And  $n!(n-r)!r!$  */
```

```
/* for given values of n and r*/
```

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

```
double factorial (int num);
```

```
main()
```

```
{
```

```
    /* Variable declaration */
```

```
    int n,r;
```

```
    double val1,val2;
```

```
    printf("Enter 2 integer numbers(n & r):");
```

```
    scanf("%d%d",&n,&r);
```

```
    val1 = factorial(n)/factorial(n-r);
```

```
    val2 = factorial(n)/(factorial(n-r) * factorial(r));
```

```
    printf("%d!/(%d-%d)! = %d1f\n",n,n,r,val1);
```

```
    printf("%d!/(%d-%d)! %d! = %d1f\n",n,n,r,r,val2);
```

```
}
```

```
/* Function to find the factorial of a number */
```

```
double factorial(int num);
{
    unsigned i;
    double fact = 1;

    /* Repeat until i reaches 1 */
    for ( i=num; i>1; i--)
        fact = fact * i;
    return fact;
}
```

**OUTPUT:**

Enter 2 integer numbers(n & r): 10 2

$10!/(10-2)! = 90.000000$

$10!/10-2)! 2! = 45.000000$

Enter 2 integer numbers(n & r): 5 3

$5!/(5-3)! = 60.000000$

$5!/5-3)! 3! = 10.000000$

Enter 2 integer numbers(n & r): 3 2

$3!/(3-2)! = 6.000000$

$3!/3-2)! 2! = 3.000000$

## **BLOCK II**

# **STRUCTURES**

There are occasions wherein the data we are handling is fairly complex - both in numbers & in nature. Items like the marks of a student etc.. can be managed with multi-dimensional arrays. But the problem with arrays is that they can handle only uniform type of data - all integers, all reals etc. In cases where this is not the case - say the personal details of an employee - his name is a string, age, salary etc.. are integers; address is a combination of strings & integers etc. arrays cannot be used. In such situations, we use the concept of structures. Each field of a structure can be of a different data type. We learn how to declare & use such structures in this block.

## STRUCTURES

We have learnt to create group elements of the same type into a single logical entity- arrays. If it looks like a jargon, consider this, the variables are being put in a group, they are referred by a single name - and importantly they were all of the same type. Either all were integers, all floats etc.. what if we want to group different types of items into one logical entity say day, month and year to be called the date? Further what if we want to group these elements into bigger entities?

Consider the following example: every date is made up of 3 parts day, month and year. Day is an integer, month is a string of character and year an integer. Now suppose, we want to declare date of birth, date of joining service, date of marriage etc.. each of them are similar to the variable date - each having a day which is an integer, a month which is a string of character and year, an integer.

C provides a nice way of declaring and managing such situations. It makes use of a concept called structure.

Struct date

```
{  
    int day;  
    char month[3];  
    int year;  
};
```

This means date is a structure - it has 3 parts - an integer called day, a character string of 3 elements called month and another integer called year.

Now we can declare date of birth as a structure of the type date structure date\_of\_birth.



Similarly structure date date\_of\_mar etc..

To get the actual month, say, of date of marriage,

We say date\_of\_marriage .date

To get the actual year of date of birth,

We use date\_of\_birth .year

And soon.

We can use these as if they are variable names. A few examples will clarify their usage.

```
/* program to illustrate a structure */
main()
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct date today;
    today.month = 9;
    today.day = 25;
    today.year = 1988;
    printf ("Today's date is %d/%d/%d\n", today.month, today.day, today.year % 100);
}
```

**OUTPUT:**

Today's date is 9/25/88.

You can use these structures in arrays also

Let us say you have an array class with 20 elements each elements pertaining to a student of that class. Now, extending the previous example, I can store the dates of birth of these 20 students, by saying

```
Struct date birthdays[20]
```

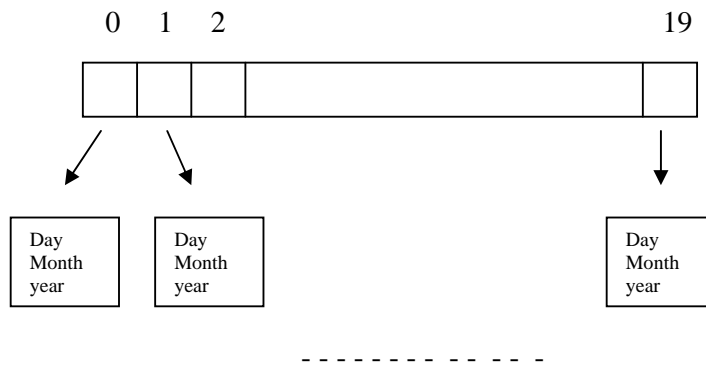
Where there are 20 birthdays, each of the type date (day,month,year)

Suppose I want to find the month of birth of 10<sup>th</sup> student, I can ask

```
Birthdays[9].month
```

Birthdays is the array, 9 indicates it is the 10<sup>th</sup> student(note that indices start from 0) and ,month indicates that we are referring to the month.

In fact the whole situation can be shown as in figure below:



Now suppose we want to store the number of days in each month of the year and take a print out. The program looks something like this

```
/* program to illustrate structures and arrays */
```

```
struct month
```

```
{
```

```

int number_of_days;

char name[3]

};

main()
{
    int i;

    static struct month monts[12] =

    {
        {31,{'J', 'a', 'n'}}, {28, {'F', 'e', 'b' } },
        {31,{'M', 'a', 'r'}}, {30, {'A', 'p', 'r' } },
        {31,{'M', 'a', 'y'}}, {30, {'J', 'u', 'n' } },
        {31,{'J', 'u', 'l'}}, {31, {'A', 'u', 'g' } },
        {30,{'S', 'e', 'p'}}, {31, {'O', 'c', 't' } },
        {30,{'N', 'o', 'v'}}, {31, {'D', 'e', 'c' } } };

    printf ("Month  Number of Days\n");

    printf ("_____");

    for (i=0; i<12; ++i)

        printf ("%c%c%c%c          %d\n",

                months[i].name[0],months[i].name[1],

                months[i].name[2],months[i].number_of_days);

}

```

**OUTPUT:**

Month	Number of Days
-----	-----
Jan	31
Feb	28
Mar	31
Apr	30
May	31
Jun	30
Jul	31
Aug	31
Sep	30
Oct	31
Nov	30
Dec	31

In fact, there is some flexibility allowed in the declaration of structure type. Instead of first declaring the structure and then indicating the element name, they can be included in one statement

For ex:

Struct date

```
{
    int day;
    char month[3];
    int year;
    date_of_birth, date_of_marriage;
}
```

of course it is a matter of individual preference and convenience to choose from amongst the options.

## **BLOCK - III**

# **FILES**

Files are one of the most useful concepts in programming. Infact the use of computers in data processing would not have picked up but for the concept of files.

Since the number of inputs that can be given through the keyboard is limited because of obvious reasons - it is desirable to store them somewhere ( in the computer memory itself) and ask the computer to take the input from that place. This is the concept of files ( of course, files need not always contain only data but still the concept holds). The other inherent advantage is that the output can also be in another file - which in turn can be accessed by a second program and so on.

## FILES

Quite often, it becomes necessary to manipulate large amounts of data, which cannot be typed manually through the keyboard. It may also be possible that the output of one program becomes an input to the other. In such cases, the data are simply stored in files. Files are what they mean in normal English – they just hold data. Each file is given a name. Instead of asking for the input from the keyboard, the program simply opens the files and reads them. Similarly one can ask the program to write it into a file. Obviously this will be a much faster & more accurate method. It also has another advantage. As we have seen above. If the output of one program is to become the input to the other, one program will write into the file and the other will read from it. Such files can also be copied into floppies and transported.

All this to simply indicate that files can be extremely handles when dealing with large amounts of data.

Now to the creation of files, every file is given a file name. The data is entered separately, or it can be even a blank file, into which data is written by the program.

Depending on what operation is done on the program, the file has to be “opened”. You cannot operate on the file until it is opened. A file can be opened for one of the three basic operations – to read data from it, to write data into it or to attach additional data it. There modes are called read, write and append modes respectively indicated by the letters r, w and a. The opened files are assigned to a file pointer.

See the following example:

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

```
FILE * infile;
```

```
Infile = fopen(“exam”, ‘r’);
```

What these statements do is as follows:

The second line indicates that infile is a pointer to a file. Each file that is being used by the program should have a separate pointer.

The next line opens a file called “ exam” in the read mode and assigns it to the infile. (fopen() is a library function used to open files and every files has to be opened in one of the three modes before it can be used). Now, we can read the contents of the file with commands similar to scanf() & printf(). The corresponding commands are fscanf & fprintf.

A typical scanf command is

```
fscanf(“exam”, “%d”, &marks);
```

Wherein the integer marks read is stored in the variable marks. Now compare this with the format of scanf.

The difference is only fscanf instead of scanf & the filename appearing in the beginning. Otherwise it is the same. The reason is that the compiler considers scanf as a special case of fscanf. When scanf is used, it presumes that the filename is not necessary but the input is from the standard i/o file namely the keyboard.

Similarly to write into a file, you open the file in the write mode, and use appropriate fprintf statements. If you open an existing file in the write mode, the file is overwritten. i.e the existing data in the file is lost. If this is to be over come, the file should be opened in the “append” mode. Then the new data is not written “ over” the existing data, but is attached to the end of the file.

To read or write single characters, just as we have getchar & putchar in normal i/o mode, we have get c & put c in the file mode.

Each existing file will be marked with a EOF (end of file) in the end which indicates that the file has ended, and this is useful while reading the file. While writing, the EOF is automatically attached to the file.

Once all operations are the file are over, it should be closed using a fclose(filename) command. A program successfully terminates only if all it’s files are closed properly.

Before we start practicing the programs one more information if a file called for writing into is not existing, the system automatically creates it. However, if a non existent file is called for reading, it declares an error.

Now let us see a simple program that reads an input file character by character and unites into another file. The file names of both the files are given at run time.

```
/* Program to copy one file to another */

#include<stdio.h>
main()
{
    char in_name[25], out_name[25];
    FILE *in, *out;
    int c;
    printf("Enter name of file to be copied: ");
    scanf("%24s", in_name);
    printf("Enter name of output file:");
    scanf("%24s", out_name);
    if ( ( in = fopen (in_name, "r")) == (FILE *) NULL)
        printf("Couldn't open %s for reading.\n",in_name);
    else if ( (out = fopen (out_name, "w")) == (FILE *) NULL)
        printf("Couldn't open %s for writing.\n",out_name);
    else
    {
        while ( (c = getc (in)) !=EOF)
            putc (c,out);
        printf("File has been copied.\n");
    }
}
```

#### OUTPUT:

Enter name of file to be copied: **copyme**

Enter name of output file: **here**

File has been copied.



## **UNIT – III**

### **UNIT INTRODUCTION**

In this, unit, we look at some of the really advanced features of C. In fact some of them have made C stand out as a programming language of a different stature.

The first one is the concept of pointers. Instead of always indicating a variable by its name, you can indicate it by its address what is called a " pointer" to the variable - Though at first it appears fairly complex - why should we complicate the matters by including one more level of abstraction? - we will see shortly that it is a fairly powerful method with several options.

The second concept is about bitwise operators. We know that computers store and manipulate data in terms of bytes. But often - in certain situations like in system programming for ex:- We will be interested to know what each bit contains things like whether the 4<sup>th</sup> bit is a 1 or 0 may be needed as also certain operations on these bits. The second block of this unit deals with such bitwise operations.

Another important ability of C is its preprocessing. These are certain user friendly commands that Unix provides for the benefit of its users. You see some of them and their usefulness in the unit.

## BLOCK – I

# POINTERS

## UNIT INTRODUCTION

Instead of always indicating a variable by its name, you can indicate it by its address what is called a "pointer" to the variable - Though at first it appears fairly complex - why should we complicate the matters by including one more level of abstraction? - we will see shortly that it is a fairly powerful method with several options.

## POINTERS

Pointers are one of the very powerful & special features of C. It simply means instead of using a variable name, you use a pointer ( or indicator) to it. At a simple level, it can be thought of as an

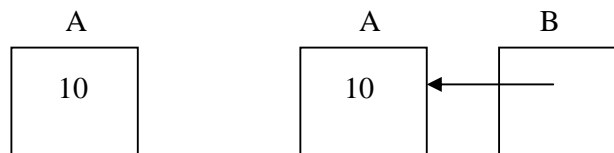


fig (i)

fig (ii)

address to the variable. The above figure illustrates the feature. In a variable A, the value 10 is stored. Now to access this, you can ask for variable A. On the other hand, you can ask for a pointer like B, shown in fig(ii). If you say B is of the pointer type, then you are actually not interested in the contents of B (which indicates where A is there), but using them you come back to A.

It is like, say, if you are interested in going to a place, you can either know where that place is (normal variables) or know a person who in them knows where the place is situated (pointer variables). Some others also call this "indirection ". You are approaching A indirectly, through its pointer B/

At this stage, one question is likely to arise. Why should we go about this round about way, instead of directly going to the variable. One advantage is as in the case of single variables, it is easier to work with normal, direct variable names. But, when you are using large chunks of data, it is easier to work with pointers. Like say when you have 100 items in an almirah, it is easier to handle a key to the almirah ( a pointer ) rather than the individual items. if you want to hand over these items to someone, it is easier to hand over the key, rather than each of the individual contents. Such situations arise, when we are passing parameter to a function and back. It is easier to pass the pointers to the function, than each of those individual items.

These are other advantages also, which become evident as we processed through this chapter. Before that, we see the general format of the pointer variable.

Suppose, we define a variable A as 10 (say)

We say `int a=10;`

Now, we have to define one more variable, which is a pointer to A, say B (see the fig above). To distinguish that A is an actual variable and B is a pointer, we indicate B as

`int *B`

The asterisk \* indicates that B is of the type pointer. Now to finally indicate that B points to A, we write

`B= &a`

The situation can be depicted as below



Both A and B are locations. A contains an integer value 10, whereas B contains a pointer to A.

We go back to the steps again

- i) Declare A as a variable
- ii) Declare B as a pointer
- iii) Link A and B

Now, how do we get the contents of A ( and say store it in X) ?

The direct way is to say  $X=A$ . Alternatively, we can say

$X = *B$ .

Then the control goes to B, takes it's contents as a pointer to A then goes to A and gets the contents.

Now let us illustrate these concepts with a simple program

```
/* Program to illustrate pointers */  
  
main()  
{  
  
    int count = 10,x;  
  
    int *int_pointer;  
  
  
    int_pointer = &count;  
  
    x = *int_pointer;  
  
    printf ("count = %d, x= %d\n", count,x);  
  
}
```

**OUTPUT:**

Count = 10, x = 10

Pointers need not always point to integers only. See the following program

```
/* Further examples of pointers */
main()
{
    char c = ' Q ';
    char *char_pointer = &c;

    printf("%c %c\n", c,*char_pointer);
    c = '/';
    printf("%c %c\n", c, *char_pointer);
    *char_pointer = '(';
    printf("%c %c\n", c,*char_pointer);
}
```

**OUTPUT:**

Q Q

/ /

( (

/\* more on pointers \*/

```
main()
{
    int i1,i2;
```

```

int *p1, *p2;

il = 5;

p1 = &il;

i2 = *p1 / 2 + 10;

p2 = p1;

printf("il=%d, i2=%d, p1=%d, p2=%d\n", il,i2,p1,p2);

}

```

**OUTPUT:**

il = 5, i2 = 12, \*p1 = 5, \*p2 = 5

These concepts are extremely simple but are very useful and you have to be fully confident of their usage.

Pointers can be used just like normal variables for arithmetic operations for example if you want to add 10 to a value pointed to by ptr1 (where ptr1 is a pointer variable), we simply write

$$*ptr1 = *ptr1 + 10$$

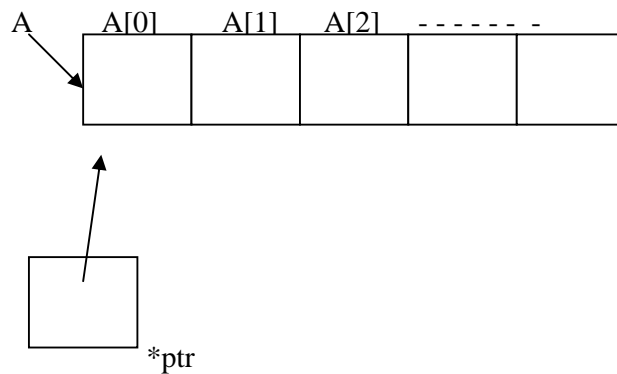
( Compare: If we want to add 10 to A, we say  $A=A+10$ )

Similarly we can say  $*ptr3 = *ptr1 + *ptr2$

But we cannot say  $ptr3=ptr1+ptr2$  ( why ?).

**Pointers to arrays:**

When an array of say 100 elements is declared, the compiler creates a storage of 100 consecutive locations, say starting from 1000. If  $A[0] = 1000$ ,  $a[I]=1000+1$ ;  $A[2]= 1000 +2$  etc.. i.e the addresses of the first element  $a[0]$  is called the base address, to which the location of the element which is required ( 1,2 ..... ) ( which is called the offset is added to get the actual address. We can as well pass a pointer to the base address of the array.



If `ptr` is the pointer to `A`, then we can declare

`Ptr = & A`, which means `ptr` is a pointer to `A`.

See the following program:

```
/* Function to sum the elements of an integer array*/
int array_sum(array,n)
int array[];
int n;
{
    int sum = 0, *ptr;
    int *array_end = array + n;

    for ( ptr = array; ptr < array_end; ++ptr)
        sum += *ptr;
    return (sum);
}
```

```

main()
{
    static int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5};
    printf("The sum is %d\n",array_sum(values, 10));
}

```

## OUTPUT

The sum is 21

Just to become familiar with the various operations using strings, we write down a few programs that necessarily operate on strings.

**/\* Program to copy a string from one string to another \*/**

```

void copy_string (from,to)
char from,to;
{
    for( ; *from != '\0'; ++from, ++to)
        *to = *from;
    *to = '\0';
}

main()
{
    static char string1[] = "A string to be copied,";
    static char string2[50];
    copy_string(string1, string2);
    printf("%s\n",string2);
}

```



```

        copy_string("So is this.",string2);

        printf("%s\n",string2);

    }

```

**OUTPUT:**

A string to be copied

So is this.

Write a program to find the number of characters in a string using pointers.

```

/* NUMBER OF CHARACTERS IN A GIVEN STRING USING POINTERS */

#include<stdio.h>

main()
{
    char a[80];

    int n;

    printf("\n ENTER A STRING :\n");

    gets(a);

    n=str-len(&a);

    printf("\n NUMBER OF CHARACTERS IN THE STRING:%d",n);

}

/* END OF MAIN */

/*FUNCTION str-len*/

str_len(s)

char *s;

{

```

```

    int i=0;

    while (*s!='\0')
    {
        i++;
        s++;
    }

    retrun(i);
}

/*END OF FUNCTION srt_len*/

```

**OUTPUT:**

ENTER A STRING :

Hi! How are you and how are your studies going on ?

**OUTPUT:**

NUMBER OF CHARACTERS IN THE STRING : 50

Write a program to count number of vowels, consonants, digits, blank-space characters and other characters in a line of text entered form the keyboard.

```

/*PROGRAM TO READ TEXT AND TEXT CHARACTERS USING POINTERS */

#include<stdio.h>

main()
{
    char line[50],s;

    int v=0, c=0, d=0, ws=0, other=0;

    void scan-line();
}

```

```

printf("\n Enter a line of text:\n");

scanf("%[^\\n]",line);

scan_line(line,&v,&c,&d,&ws,&other);

printf("\n Number of Vowels:%d",v);

printf("\n Number of Consonants:%d",c);

printf("\n Number of Digits:%d",d);

printf("\n Number of Blank spaces:%d",ws);

printf("\n Number of other characters:%d",other);

}

/* END OF MAIN */

/*FUNCTION scan_line */

void scan_line (line,pv,pc,pd,pws,pother)

char line[ ];

int pv,pc,*pd,*pws,*pother;

{

    char s;

    int count=0;

    while((s=toupper(line[count])) !='\0')

    {

        if(s= 'A' || s= 'E' || s= 'I' || s= 'O' || s= 'U')

            ++*pv;

        else if (s>='A'&& S<='Z')

            ++*pc;

```

```

        else if (s>='0' && s<'9')
            ++*pd;
        else if (s>==' ' || s=='\t')
            ++*pws;
        else
            ++*pother;
        ++count;
    }
}

/* END OF FUNCTION scan_line*/

```

**OUTPUT:**

Enter a line of text:

This 1 line is to test all characters in a line !@#\$%.

Numbers of Vowels : 14

Number of Consonants:22

Number of digits: 1

Number of Blank spaces : 11

number of other characters : 5

Write a program to copy the contents of one string to another string using pointers.

```
/* COPYING A STRING USING POINTERS */
```

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

```
main()
```

```

{
    char a[80], b[80];

    printf("\n ENTER STRING 'A':\n");

    gets(a);

    printf("\n");

    strcpy(&a,&b);

    printf("CONTENTS OF STRING 'B' :\n");

    output(&b);
}

/*FUNCTION MAIN */

strcpy(s,d)

char *d,*s;

{
    while (*s!='\0')
    {
        *d = *s;

        d++;

        s++;

    }

    *d++ = '\0';
}

/* END OF FUNCTION strcpy */

/*FUNCTION output */

```

```

output(s)
char *s;
{
    while(*s!="\0")
    {
        printf("%c",*s);
        s++;
    }
}
/* END OF FUNCTION output */

```

**OUTPUT:**

String A will be copied into String B

CONTENTS OF STRING 'B':

String A will be copied into String B.

Write a program to concatenate two strings into a new string.

```
/* TO CONCATENATE THE GIVEN TWO STRINGS */
```

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

```
main()
```

```

{
    char a[80],b[80],c[160];

    printf("\n ENTER  A STRING :\n");

    gets(a);

```

```

printf("\n ENTER ANOTHER STRING:\n);

gets(b);

stringcat(&a,&b,&c);

printf("\n THE CONCATENATED STRING IS :\n");

output(&c);

}

/*END OF MAIN */

/* FUNCTION stringcat */

stringcat(s1,s2,d)

char s1,s2,*d;

{

    while(*s1 !='\0')

    {

        *d = *s1;

        d++;

        s1++;

    }

    *d++ = '\0';

    while (*s2 != '\0')

    {

        *d = *s2;

        d++;

        s2++;

    }

}

```

```

    }

    *d = '\0';

}

/*END OF FUNCTION strcat */

/*FUNCTION output */

output(s)

char *s;

{

    while (*s != '\0')

    {

        printf("%c", *s);

        s++;

    }

}

/* END OF FUNCTION output */

```

**OUTPUT:**

ENTER A STRING: MAHANTESH

ENTER ANOTHER STRING: TEGGI

THE CONCATENATED STRING IS: MAHANTESHTEGGI

Pointers and structures : Pointers can also be used in structures. We have seen, previously, the “date” structure as follows:

Struct date



```

{
    int month;
    int day;
    int year;
}

```

Just as we have defined the type struct date as in

Struct date birthday,

We also define a variable to be a pointer to the structure

Struct date \* date pointer and then say

Date pointer = &birthday;

Then, we can go on to access, say the month of the birthday by saying (\*date pointer),month

We see an example

```
/* program to illustrate structure pointers */
```

```
main()
```

```
{
```

```
    struct date
```

```
    {
```

```
        int month;
```

```
        int day;
```

```
        int year;
```

```
    };
```

```
    struct date today, *date_pointer;
```

```

date_pointer = &today;

date_pointer ->month = 9;

date_pointer ->day = 25;

date_pointer ->year = 1988;

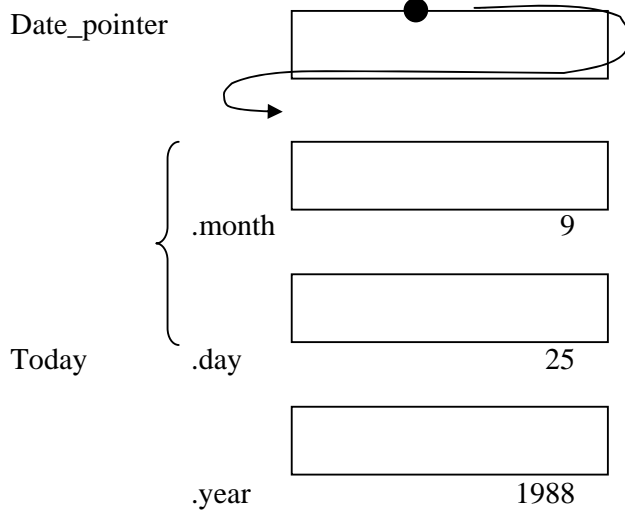
printf("Today's date is %d/%d/%d.\n",date_pointer->month,date_pointer->day,

      date_pointer->year %100);

```

**OUTPUT:**

Today's date is 9/25/88.



This was about pointers to structures. We can also have pointers in structures. i.e the structure elements themselves can be pointers. Something like

Struct int pointers

```

{
    int *p1;
    int *p2;
}

```

};

See the following example

```
/* structures containing pointers */

main()
{
    struct int_pointers
    {
        int *p1;
        int *p2;
    };

    struct int_pointers pointers;

    int i1 = 100, i2;

    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;

    printf (" i1 = %d, *pointers.p1 = %d\n", i1, *pointers.p1);
    printf ("i2 = %d, *pointers.p2 = %d\n", i2, *pointers.p2);
}
```

### OUTPUT:

i1 = 100, \*pointers.p1 = 100

i2 = -97, \*pointers.p2 = -97

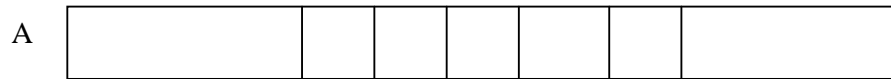
## LINKED LISTS

The concept of linked lists is one of the powerful features in C. It will also bring to you some of the powerful features of pointers.

Now let us go back to the arrays we have previously said that if an array of 100 elements are to be declared, the compiler catches hold of 100 consecutive memory locations and marks them as the elements of the array. Whenever an element is to be visited, it's address is calculated based on the offset from the base element ( see the previous section )

Herein lies a problem. Suppose, we need an array of 10,000 elements. Sometimes, it may not be possible to get 10000 successive locations in the memory. Then, even though 10,000 locations are there ( spread all over the memory) just because they are not in consecutive locations(in one place). We will have to say it is not possible to allocate the array.

We have not addressed one or two other issues with the arrays. Let us say, suppose, we want to rename the 20<sup>th</sup> element & shift the other elements back.



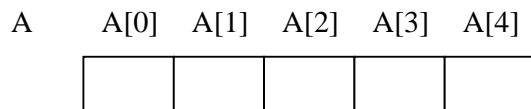
A[20] 21 22 - - - - -

Then what we have to do is to shift the 21<sup>st</sup> element to 20<sup>th</sup> location. 22<sup>nd</sup> element to 21<sup>st</sup> location etc.. etc.. suppose the array is 1000 elements long. It is still possible, but takes enormous amount of effort.

Similar is the case, if we want to insert an element we have to push the 1000<sup>th</sup> element to ( 1001<sup>th</sup> place ( if there is space) and push 999<sup>th</sup> to 1000<sup>th</sup> place etc.. And if the array is just declared as only a 1000 element one, then we cannot do the operation all. We have to create a new array, ( a longer one) and copy these elements to the new one.

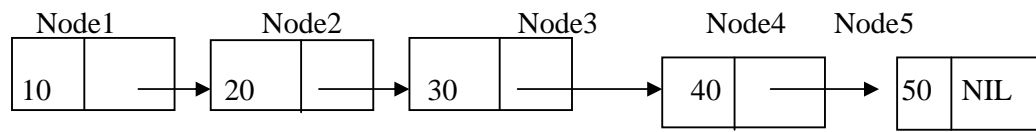
i.e if, before hand, we are not sure of the actual size of the array elements, it is difficult to manage the array.

All these difficulties lead us to the concept of “ dynamic ” allocation. (Arrays are supposed to be static: you declare them at one time, that is it). Look at the following the figures. A is an array with 5 elements.

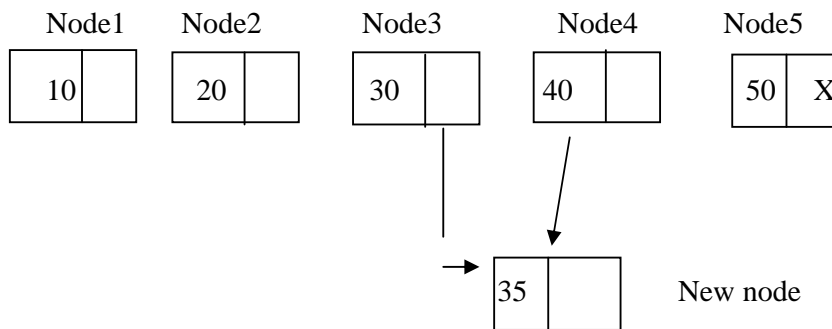


Now I cannot add a new element to this, because of the reasons described above, I can delete an element, but it needs some effort on my part.

Now consider the same data represented as follows:



The second is what is called a “ linked list” a list of linked elements. Each “ node” has 2 elements one is a data element and the other a pointer to the next node. Your knowledge of structures should immediately indicate to you that this can be declared as a structure. Once done, go back to the difficulties of using arrays, which we had listed above. Because we are using pointers, the nodes need not be consecutive. We need not even know before hand how many nodes we need. As and when we want to add new elements, we can keep creating new nodes and keep connecting them using pointers.



i.e create a new node, make the pointer of node 3 point to new node and make the pointer of new node point to node4.

Similarly, if we want to delete node2, all that we have to do is to make the pointer of node1 point to node3.

The following examples illustrate how to create & traverse such linked lists.

```
/* linked lists */
```

```
main()
```

```
{
```

```
    struct entry
```

```
    {
```

```
        int value;
```

```
        struct entry *next;
```

```
    };
```

```
    struct entry n1,n2,n3;
```

```
    int i;
```

```
    n1.value = 100;
```

```
    n2.value = 200;
```

```
    n3.value = 300;
```

```
    n1.next = &n2;
```

```
    n2.next = &n3;
```

```

        i = n1.next->value;
        printf ("%d ",i);
        printf("%d\n", n2.next->value);
    }

```

## OUTPUT

200 300

```
/* List traversal */
```

```
main()
```

```
{
```

```
    struct entry
```

```
    {
```

```
        int value;
```

```
        struct entry *next;
```

```
    };
```

```
    struct entry n1,n2,n3;
```

```
    struct entry *list_pointer = &n1;
```

```
    n1.value = 100;
```

```
    n1.next = &n2;
```

```
    n2.value = 200;
```

```
    n2.next = &n3;
```

```
    n3.value = 300;
```

```
    n3.next = (struct entry *) 0;
```

```
    while (list_pointer != (struct entry *) 0)
```

```
    {
```

```

        printf ("%d\n", list_pointer->value);
        list_pointer = list_pointer->next;
    }
}

```

## OUTPUT

100

200

300

In fact, there is a lot of literature available on linked lists, but we stop at this stage and move on to the other use of pointers.

## Pointers and functions:

Pointers and functions get on very well. Whenever data is to be passed to a function, especially large amounts of them, it is easier to pass a pointer to a function and to get back a pointer as the return value of the function.

Now, consider the very simple problem of interchanging two integer values. Suppose A is storing 5 and B is storing 10, after exchanging A should hold 10 and B should hold 5 obviously another temporary location, temp, is used to hold one of the values, while interchanging is taking place.

```
/* more on pointers and functions */
```

```
void exchange (pint1, pint2)
```

```
int *pint1, *pint2;
```

```
{
```

```
    int temp;
```

```
    temp = *pint1;
```



```

    *pint1 = *pint2;

    *pint2 = temp;
}

main()
{
    int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

    printf("i1 = %d, i2 = %d\n", i1, i2);

    exchange(p1, p2);

    printf("i1 = %d, i2 = %d\n", i1, i2);

    exchange(&i1, &i2);

    printf("i1 = %d, i2 = %d\n", i1, i2);
}

```

## OUTPUT

i1 = -5, i2 = 66

i1 = 66, i2 = -5

i1=-5, i2 = 66

Now consider a more complicated program. The function here searches for a particular value in a linked list. If it is found, it returns a pointer to the node holding the value. Otherwise it returns a null pointer.

struct entry

```

{
    int    value;

    struct entry *next;

};

struct entry *find_entry (lpointer, match_value)

struct entry *lpointer;

int          match_value;

{
    while (lpointer !=(struct entry *) 0)
        if (lpointer->value == match_value)
            return (lpointer);
        else
            lpointer = lpointer->next;
    return (struct entry *) 0);
}

main()

{
    struct entry n1,n2,n3;

    struct entry *lptr, *list_start = &n1;

    int    i,search;

    n1.value = 100;

    n1.next = &n2;

    n2.value = 200;

```

```

n2.next = &n3;

n3.value = 300;

n3.next = (struct entry *) 0;

printf(" Enter value to locate:");

scanf("%d", &search);


if(lptra != (struct entry *) 0)

printf("Found %d\n",lptra->value);

else

printf("Not found.\n");

}

```

## OUTPUT

Enter Value to locate: 200

Found 200.

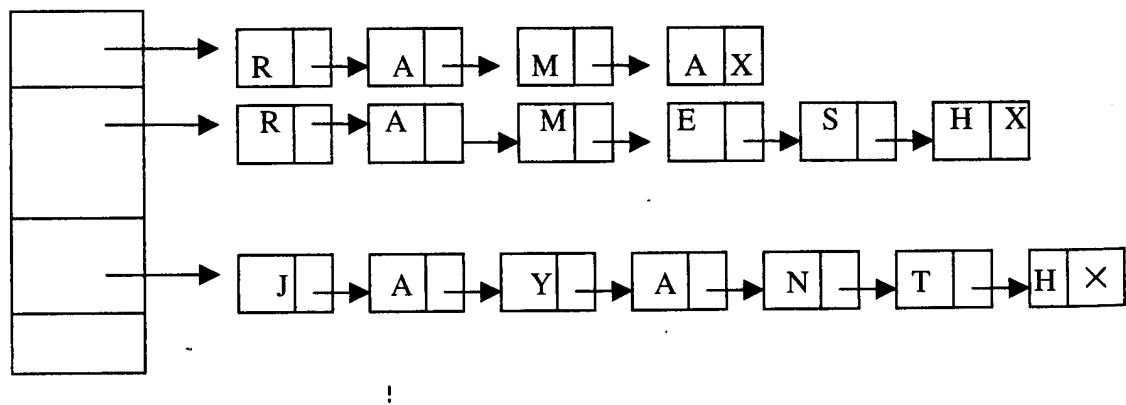
Enter Value to locate: 400 (Re-run)

Not found 200.

Enter Value to locate: 300 (Re-run)

Found 300.

In fact the concept of pointers can keep expending to any extent, you can have an array of pointers. For example in a class there are 30 students. You have store their names. While the no. of students is constant at 30, the no. of characters in their names is variable. Then the simplest thing is to create an array of pointers of size 30, each pointing to a linked list of one name.



However, we conclude the discussions of pointers at this stage and move on to the next topic.

## **BLOCK II**

# **BITWISE OPERATORS**

## **Block Introduction**

We know that computers store and manipulate data in terms of bytes. But often - in certain situations like in system programming for example, we will be interested to know what each bit contains. Things like whether the 4<sup>th</sup> bit is a 1 or 0 may be needed, as also certain operations on these bits. This Block deals with such bitwise operations.

## BITWISE OPERATORS

We know that computers operate on one byte at a time i.e. 8 bits of data a manipulated as one unit. However, these are situations wherein we may need to know and operate on individual bits, like say depending on whether the 5<sup>th</sup> bit is 1 or 0, some operations have to be done. Such situations occur in several applications, like in systems programming. 'C' provides facilities for such operations.

The normally used bit operators are listed below

Symbol	Operation
&	Bitwise AND
	Bitwise Inclusive-OR
^	Bitwise Exclusive-OR
~	Ones Complement
<<	Left Shift
>>	Right Shift

For those not familiar with boolean algebra, a brief description of the above operations follows.

**Bitwise AND:** this operation on 2 bits. When both bits are 1, the output is one, otherwise it is Zero.

Consider the example of bitwise ANDing of 14 and 12 .

In a 8 bit machine, it turns out to be

```

      00001110
      00001100
      -----
Result 00001100
      -----

```

Bitwise inclusive OR: This also operates on two bits. The output is 1 if either of the two bits is 1, otherwise it is 0.

```

Consider   10011010
           01100011
           -----
The result is 11111011
           -----

```

Bitwise Exclusive OR : This operation returns a 1 if either of the two bits (but not both) is a 1.

The ones complement : This works on a single number and simply converts each 1 to 0 and each 0 to 1.

Consider      01101101    After complementing it becomes  
                          10010010

The left shift operator: this also works on single number. Each bit of the number is shifted one bit to it's left. The left most bit is lost and the right most bit, that becomes vacant is filled with a 0.

                 Consider      00101000 = 40  
 After Shifting left      01010000 = 80

                 Note also that a left shift operation results in the number getting multiplied by 2.  
 (There is an exception to this. Figure out when it is so).

The right shift operator

                 This the reuses of the left shift operator. It just shifts each bit one place to it's right.  
 The rightmost bit is lost, the left most bit is made 0.

                         00101000 = 40  
 becomes            00010100 = 20

Note that this results in the number getting divided by 2.

Though the concepts presented are simple, we write one program that uses the concept.

Write a program to rotate a given number called value a given number of times, n. If n is positive rotate it left, otherwise right.



[ Note: Rotation means shifting each bit by one place and recovering the lost bit. For ex. In left shift, each bit is shifted one place to the left and the leftmost bit, which comes out is returned to the right most place.]

```
/* Function to rotate an unsigned int left or right */
```

```
unsigned int rotate (value, n)
```

```
unsigned int value;
```

```
int n;
```

```
{
```

```
    unsigned int result, bits;
```

```
    if(n== 0|| n== -16 || n== 16)
```

```
        return (value);
```

```
    else if (n > 0)          /* left rotate */
```

```
    {
```

```
        n=-n;
```

```
        bits = value << (16 - n);
```

```
        result = value << n | bits;
```

```
    }
```

```
    else
```

```
    {
```

```
        n= -n;
```

```
        bits = value << (16 -n);
```

```
        result = value >> n | bits;
```

```

}
return (result);
}
main()
{
    unsigned int w1 = 0xalb5, w2 = 0xff22;

    printf("%x\n", rotate (w1, 4);

    printf("%x\n", rotate (w1, -4);

    printf("%x\n", rotate (w2, 8);

    printf("%x\n", rotate (w2, -2);
    printf("%x\n", rotate (w1, 0);
}

```

\*\*\*\* Out put of the above program

1b5a

5a1b

22ff

bfc8

alb5

## **BLOCK III**

# **THE PREPROCESSOR**

## **Block Introduction**

Another important ability of C is its preprocessing. These are certain user friendly commands that Unix provides for the benefit of its users. You see some of them and their usefulness in the Block.

## THE PREPROCESSOR

This is one of the special features available in C, but not found in many other languages. The preprocessor is a part of C-compilation process that recognizes special statements (that may be found anywhere in the program) and takes appropriate action(s). The programmer can use these special statements to make his program easier to read and modifications, customise or write easily transportable programs. Each preprocessor statement is identified by a # sign, which must be the first character of the line.

We shall briefly see some of the preprocessor statements and their uses.

The define statements : This is one of the most popularly used C preprocessor statements, with multiple users. The format is something like

```
#define TRUE 1
```

```
#define false 0
```

(note that there is no semicolon after the statement)

Once this is done, throughout the program, you can use True wherever constant 1 is to be used and false where 0 is to be used. Note that they can be used only as constants and not variables. For example we can not add 1 to True etc.

All that the preprocessor does is to scan through the program and replace True with 1 and false with 0, so that the C compiler can understand them.

You may ask why it is needed at all. Quite often, we need to take decisions based on whether something is true or false. Unlike other languages (like Pascal, for ex.) C does not have Boolean variables that can be set to true or false. In such a case, the programmer has to artificially remember to replace true with 1 and false as 0 (or any other combination for that matter) and write the program. This may lead to confusions or mistakes. Thus by defining them in the beginning and then writing the program in the way we think. We will be able to write better programs. We see an example to classify the simulation.

```
#define TRUE 1

#define FALSE 0

/* Function to determine if an integer is even */

int is_even (number)

int number;

{

    int answer;

    if (number % 2 == 0)

        answer = TRUE;

    else

        answer = FALSE;

        return(answer);

}

main()

{

    if (is_even(17) == TRUE)

        printf("YES");

    else

        printf("NO");

    if(is_even (20) == TRUE)

        printf("YES\n");

    else

        printf("NO\n");
```

```
}
```

\*\*\*Output

NO YES

Now, going back to the format, we specifically insisted that no semicolon should be there after define. This is because the preprocessor blindly replaces the True value with 1 and false with 0. Suppose we have written

```
# define True 1 ;
```

Then in the place where the statement `answer = True;` appears, True gets replaced by 1; so the compiler sees a statement `answer = 1;;` which is invalid. Hence no semicolons should appear in define.

- i) Program extensibility : suppose you have written a program which has something to do with 1000 students. So, your array is of the size 1000, these may be a number of for loops which terminate at 1000 etc. now, after sometime, the number of students increases to 1500. Then wherever 1000 appears, you have to change it 1500. This is easier said than done. You are bound to miss some commit errors etc. An easier method is to simply define no of students as 1000.

```
# no_of_students 1000
```

and wherever 100 is to be used to indicate the students, use the constant `no_of_students` like for

```
(i=0;i<no_of_students;++i)
```

Whenever the number of students change, we can change only the define statement and not throughout the program.

### **Program Portability :**

Certain parameters are machine dependent, like the maximum integer you can store etc. when you go from a 16 bit machine to a 32 bit machine or viceversa, you have to change them. Instead, you can use a define statement like

`#define wordlength 16` , and when you go through a 32 bit machine, simply change the statement.

Simple Functions: Define need not always be simple numbers. It can be pointers or even simple equalians consider `#define square(s) x*x`

This would same almost as a library function and whenever we write say

`y = square(z)`, it gets replaced to `y = z*z`

Consider one more interesting case. You have to check whether a particular year is a leap year and then take some decisions. Then, wherever we have to check for leap year, we should write `if (year%4 == 0) && (year%100!=0 || year % 400 == 0)`.

Instead at one place, write

```
# define leap_year ((year%4 == 0) && (year%100!=0 \
    || year % 400 == 0))
```

and use `if(leap_year)` in the program

(incidentally the `\` is used whenever the define exceeds one line.)

Since define statement s can be used as definitions, they are often called "macros", a form used for expandible assembly language functions.

The `# include` statement:

If you keep writing a large number of C porgrams, you realise that certain define functions are needed again and again. Instead of writing them in define functions in every program, you can collect all of them in a file and include that file in the program.

Now you can have a clear concept of `# include` files we were using for input/output functions. Only things `stdio.h`, `math.h` etc. file are provided by the system. But you can also write your won files and include them using the include statement.