

PROGRAMMING FOR PROBLEM SOLVING

UNIT-I & UNIT-II

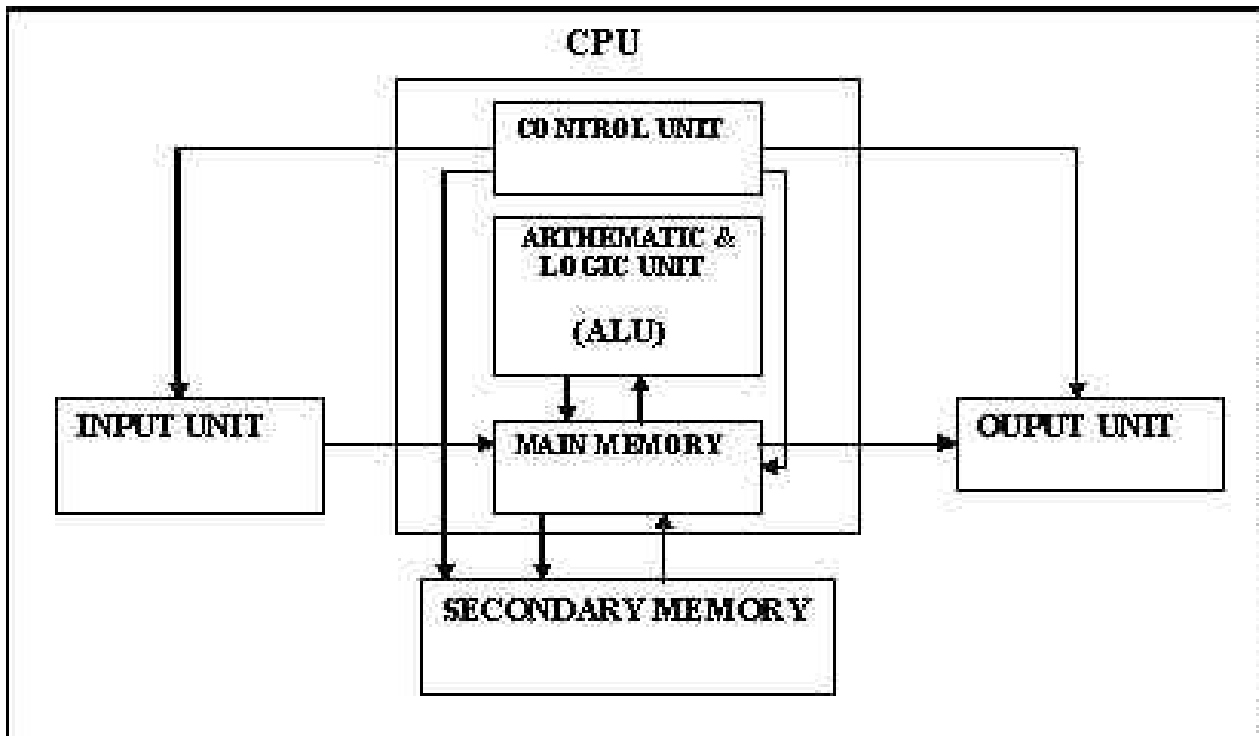
INTRODUCTION TO COMPUTERS:

What is a Computer?

Historically, a computer, broadly defined, as an electronic device that helped in computation – counting, calculating, addition, multiplying, dividing etc. As you might guess from the word computation computer is defined.

Today computer is an electronic device that stores, manipulates and analyzes the information.

BLOCK DIAGRAM OF A COMPUTER:



COMPONENTS OF A COMPUTER SYSTEM:

1. INPUT UNIT:

It obtains information (data and programs) from various input devices and place this information at the disposal of the other units so that information may be processed. Most information is entered into computer today through keyboard and mouse devices. Information can also enter by speaking and by scanning images. Some of other input devices are touch screen (used in ATM's), joystick, electronic pen etc.

2. CENTRAL PROCESSING UNIT (Processor):

It is the mind and heart combined with the central nervous system of a computer. So the CPU performs functions similar to the mind, heart of a human body. It consists of three components

- 2.1) Arithmetic & Logic Unit (ALU)
- 2.2) Storage Unit
 - a. Main memory
 - b. Secondary memory
- 2.3) Control Unit

The functions of CPU (Processor) are to:-

- (i) Stores data as well as instructions.
- (ii) Controls the sequence of operations as per the stored instructions.
- (iii) Issue commands to all parts of the computer system.
- (iv) Carry out data processing and send results to output

2.1 Arithmetic and Logic Unit (A.L.U):

It operates on the data available in the main memory and sends them back after processing, once again to the main memory. A.L.U performs two functions

- (i) It carries out arithmetical operations like addition, subtraction, multiplication and division .
- (ii) It performs certain logical actions based on AND and OR functions.

2.2 Control Unit:

The control unit directs all operations inside the computer. It is known as nerve center of the computer because it controls and coordinates all hardware operations i.e. those of the CPU and input-output devices. Its actions are

- i. It gives command to transfer data from the input device to the memory to ALU.
- ii. It also transfers the results from ALU to the memory and onto the output device for printing.
- iii. It stores the program in the memory, takes instructions one by one, understands them and issues appropriate commands to the other units.
- iv. It fetches the required instructions from the main storage.

2.3 Storage Unit:

a) Primary Memory (RAM-Random access memory):

It is also called as main memory because, like the human memory it is able to store information, which can be recalled or accessed when required. The program of instructions has to be stored in the main memory in order to make it work automatically.

Any item of data or any instruction stored in this memory can be retrieved by the computer at high speed. The modern computer does this in Nano seconds. Main high speed memory limited in size and very costly to buy.

Examples: RAM(Random access memory)

b) Secondary Memory (Hard Disk):

These are also known as extended or auxiliary memory. These are the devices that hold the mass of information, which may be transferred during processing. These devices are used for permanent storage of data. As compared to the primary memory, it has a much larger capacity, but is not as fast. The computer thus takes slightly more time to retrieve from secondary storage. Secondary storage is much cheaper than main memory.

Example: Hard disk, Compact disks (CDs)

3. OUTPUT UNIT:

The result of any computer processing has to be communicated to the user. Output devices translate the computers output into a form understandable to human beings. Some of the output devices are display screen, printer etc.

HARDWARE:

It is a general term used to represent the physical and tangible components of the computer itself i.e. those components which can be touched and seen. It includes

- (1) Input devices
- (2) Output devices
- (3) Central processing Unit
- (4) Storage devices

SOFTWARE:

It is a general term to describe all the forms of programs associated with a computer. Without software, a computer is like a human body without a soul. There are Four major categories of software

- (1) System Software (Operating Systems)
 - (2) Application Software (MS Office & Internet Browsers)
 - (3) Utility Software (Antivirus, Disk Repair and Disk Cleaners)
 - (4) Language processors (Compiler, Interpreter & Assembler)
-

OPERATING SYSTEM:

It is the interface between the user and the computer hardware. These programs are in-built into the computer and are used to govern the control of the computer hardware components, such as processors, memory devices and input/output devices.

Examples: MS-DOS, Win XP, Win 2007, Solaris ,Unix , Linux, Android, IOS.

It is system software which controls the hardware and software resources of the computer and offers common services for computer programs.

The operating system is used as a midway between programs and the computer hardware for hardware functions such as input and output memory allocation. The operating system is present in computer, mobile phones and video game consoles to supercomputers and web servers.

There are some different operating systems can be used for different purposes. Several operating systems are used for normal things on personal computers and others are used for particular work.

An operating system has several works in which entirely programs can use the CPU, system memory, displays, input devices, and other hardware. Operating system also talks to other computers or devices on a network. There are several operating systems such as macOS, Linux, and Microsoft Windows.

PROGRAMMING LANGUAGES:

Programming languages are broadly classified into the following three forms.

1. MACHINE LANGUAGE:

Computer is an electronic device which can understand only pulse and no-pulse (1 and 0) conditions. Therefore, the instructions given to the computer and the data fed to it must be written using binary numbers 1 and 0. A program written in binary coded form(0's and 1's) which can be directly fed to computer for execution is known as MACHINE LANGUAGE and represents the lowest level of programming.

Computers are not identical in design. Therefore, each computer has its own machine language. It is very hard to understand and remember the various combinations of 0's and 1's representing numerous data and instructions. It takes more time to write programs and to debug. Thus machine language programs are difficult to write and maintain.

Also, since every computer has its own machine language, the programmer cannot communicate with other computers if he does not know its machine language.

Advantage: 1) It executes very fast.

Disadvantage: 1) It is very hard to develop an application or program.

2. ASSEMBLY LANGUAGE:

To ease the programmer's burden, mnemonic operation codes were developed. One of the first steps in improving the program preparation process was to replace the numeric language

operation codes with mnemonics.

Thus, an assembly language uses mnemonic codes(abbreviations) rather than numeric codes(as used in machine lang).For example

```
ADD  CP  SP
SUB  CP  SP
```

As the computer understands only machine language instructions, a program written in assembly language must be translated into machine language before it can be executed. This translation is done by another program called “ASSEMBLER”. For example

```
SUB          CP          SP
01111111    011111111111 000000111111
```

Assembly language has several advantages over machine language. These programs are easy to write and modify than machine language programs. However, assembly language is again a machine oriented language and the program has to be different for different machines.

ASSEMBLER:

An assembly language can be translated into a machine language by replacing the mnemonic and symbolic information by its numeric equivalent. A program which converts an assembly language program into a machine language program is called an assembler.

3. HIGH LEVEL LANGUAGES:

A set of languages have been devised which are very close to English language. Such a set of languages is called HIGH LEVEL LANGUAGES. A program has to be translated into machine language before the computer can perform all operations specified by this language. Special programs called language processors are available to do this job. These special programs accept the user program and check each statement and, if it is grammatically correct, produce a corresponding set of machine language instructions. Language processors are also known as translators.(Compilers and Interpreters)

The most important characteristic of high level language is that it is machine independent. The advantage is it is easy to develop an application(program) using HLL's then AL's and MLL and the disadvantage is its take extra time for conversion and thus are less efficient as compared to the machine language programs.

COMPILER:

A high level language can also be converted into a machine language by a program called compiler.

A compiler checks the entire user-written program (known as the source program) and, if error free, produces a complete program in machine language (known as object program). The source program is retained for possible modifications and corrections and the object program is loaded into the computer's memory for execution.

INTERPRETER:

An interpreter does a similar job but in a different style. The interpreter (as the name implies) translates one statement at a time and, if error-free, executes the instruction. This continues till the last statement in the program has been translated and executed. Thus, the interpreter translates and executes the first instruction before it goes to the second, while a compiler translates the whole program before execution can begin.

The major differences between them are:

- a) Error Correction is much simpler in the case of an interpreter.
- b) Interpreter take more time for the execution of a program compared to compilers because a statement has to be translated every time the program is executed.

LINKER:

C programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project. Linker is a program which links the object code with the required functions (stored in header files) to produce an executable code.

LOADER:

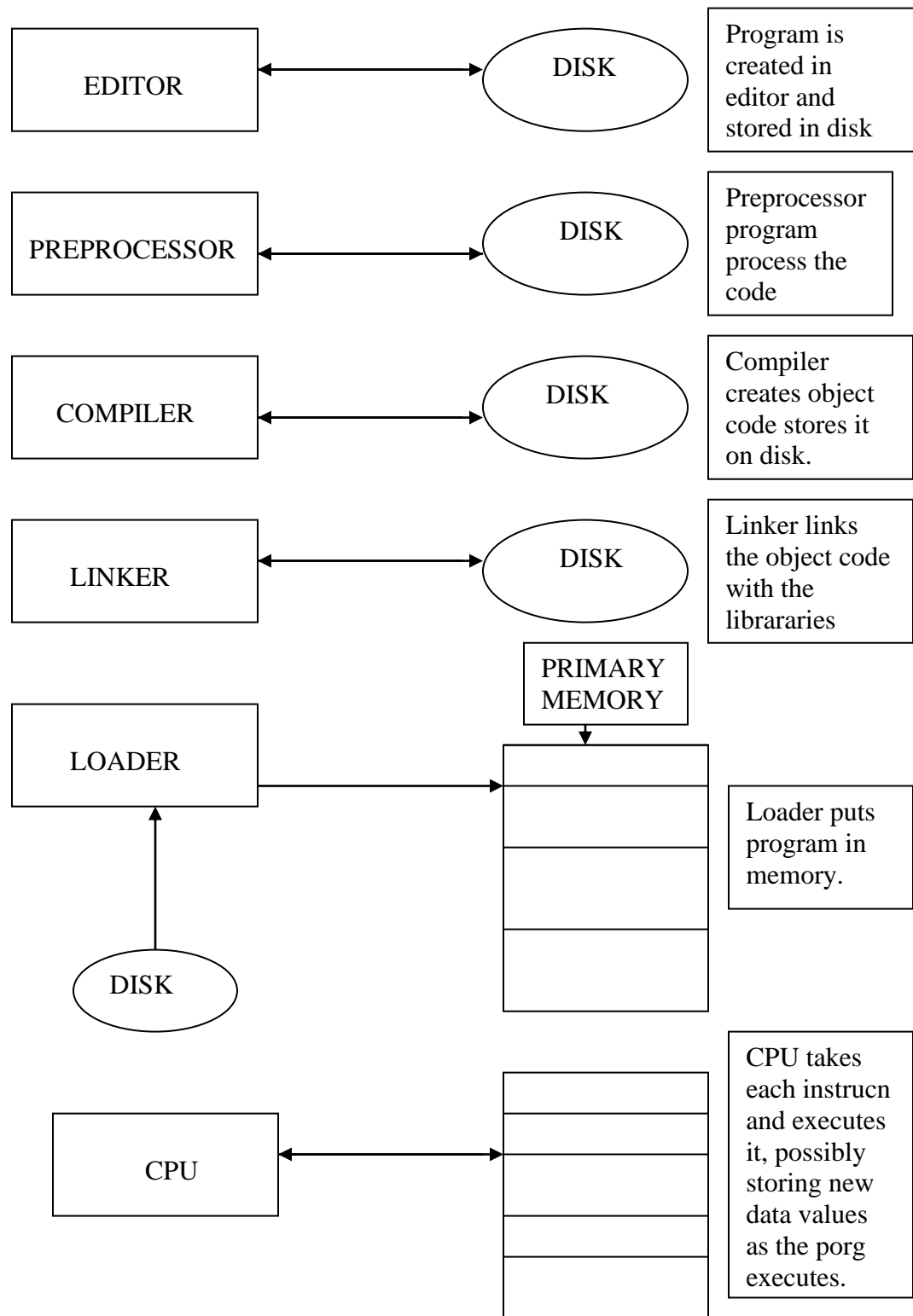
The next phase is loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which is a program that takes the executable code from disk and loads it into the memory. Additional components from shared libraries that support the program are also loaded by the loader.

HISTORY OF C:

C language was developed by Dennis Ritchie at AT & T Bell Laboratories in USA in the year 1972.

C language evolved from Basic combined programming language (BCPL) (developed by Martin Richards for writing operating system software and compilers in 1967) and B (developed by Ken Thomson for developing UNIX operation system in 1970).

BASICS OF TYPICAL C PROGRAM DEVELOPMENT ENVIRONMENT/
DIFFERENT PHASES OF COMPILATION:



C programs typically go through six phases to be executed .These are Edit, preprocess, compile, link, load and executes.

The first phase consists of editing a file. This is accomplished with an editor program. C/C++ integrated program development environment software packages such as Borland C and Turbo C for pc's and compatibles have built-in editors that are smoothly integrated into the programming environment. The programmer types a C program with the editor and makes

corrections if necessary, then storing the program on a secondary storage device such as a disk. C program file names should end with the .C extension.

Next, the programmer gives the command (Alt+f9) to compile the program. The compiler translates the C program into machine language code (also referred to as object code). In a C system, a preprocessor program executes automatically before the compiler's translation phase begins. The C preprocessor obeys special commands called preprocessor directives which indicate that certain manipulations are to be performed on the program before compilation. The preprocessor is automatically invoked by the compiler before the program is converted to machine language.

The next phase is called linking. C programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project. Linker links the object code with the required functions to produce an executable code.

The next phase is loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which takes the executable code from disk and loads in to the memory. Additional components from shared libraries that support the program are also loaded by the loader.

Finally, the computer under the control of the CPU executes the program one instruction at a time. After executing the program we get the required output.

THE C STANDARD LIBRARY:

C programs consist of modules or pieces called functions. You can program all the functions you need to form a C program, but most C programmers take the advantage of a rich collection of existing functions called *C Standard Library*. Thus, there are really two pieces to learning the C. The first is learning the C language itself and the second is learning how to use the functions in the standard C library. The standard C library's required reading for programmers who need a deep understanding of the library functions, how to implement them and how to use them to write portable code.

e.g. printf() and scanf() functions are defined in standard input/output header file (stdio.h).

pow(a, b) and sqrt (n) defined in math header file (math.h).

clrscr() and getch (); defined in console input/output header file (conio.h)

C CHARACTER SET:

A Character denotes any alphabet, digit or special symbol used to represent information. There are 255 characters defined in c. Every character is represented with an ASCII value.

ASCII- AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

CHARACTERS	SYMBOLS	ASCII VALUES
ALPHABETS	A,B,C.....Z	65-90
	a,b,c.....z	97-122
DIGITS	0,1,2,3,.....9	48-57
SPECIAL SYMBOLS	~,!,@,#,\$,%,^,&,*,(,),_,+, ,`,.,>,<=,-,?/,\\,},{,[,],:;,,, etc	0-47,58-64,91-96 123-127

CONSTANTS:

C constants are divided into following five categories:

1. INTEGER CONSTANTS:

- a) An Integer constant must have at least one digit.
- b) It must not have a decimal point.
- c) It can be either positive or negative.
- d) If no sign precedes it considers as positive no.
- e) No commas or blanks are allowed within an integer constant.
- f) The allowable range for integer constants is -32768 to 32767.

Valid-Ex: 426

+789

-3456

Invalid - 546.89 (decimal point)

10,000 (comma is present)

20 000(blank is present)

2. REAL CONSTANTS:

- a) A Real constants must have at least one digit.
- b) It may or may not have a decimal point.
- c) It can be either positive or negative.
- d) If no signs precede it considers as positive no.
- e) No commas or blanks are allowed within a real constant.

EX: 325.345

-345.322

3. CHARACTER CONSTANTS:

A character constant is a single alphabet, a single digit or a single special symbol enclosed with in ' (single inverted commas). The maximum length of a character constant can be 1 character. If more than 1 characters it is not a character constant.

Valid Examples: 'a'
'1'
'%'

Invalid: 'ab'
'34'
'2a'

4. STRING CONSTANTS:

A single character or group of characters enclosed in "" (double quotes) is called as string constant.

Examples: "a" "ab" "Hello" "2345"

5. BACKSLASH CHARACTER CONSTANTS (ESCAPE SEQUENCES)

An escape sequence is a sequence of characters that does not represent itself when used inside a string constant, but is translated into another character or a sequence of characters that may be difficult to represent directly.

In C, all escape sequences consist of two or more characters, the first of which is the backslash, \ (called the "Escape character"); the remaining characters determine the interpretation of the escape sequence. For example, \n is an escape sequence that denotes a newline character.

Suppose we want to print out Hello, on one line, followed by world! on the next line then following example can be seen.

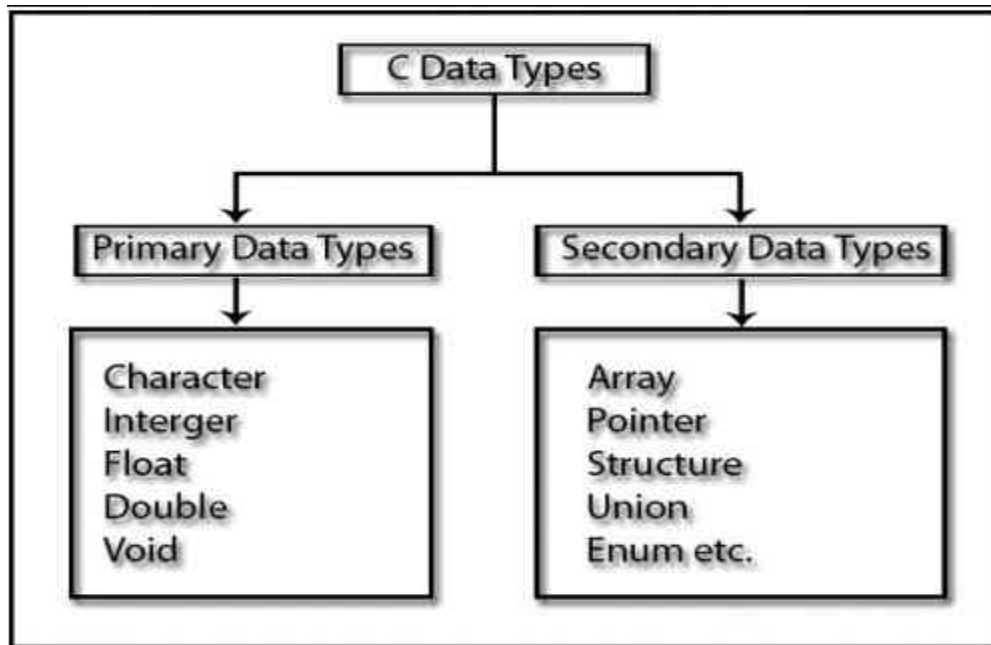
```
#include <stdio.h>
void main()
{
    printf("Hello\nworld!");
}
```


Escape sequence	Character represented
\a	Alert (Beep, Bell)
\b	Backspace
\n	Newline (Line Feed)
\t	Horizontal Tab
\v	Vertical Tab
\\	Backslash
\'	Apostrophe or single quotation mark
\"	Double quotation mark

DATA TYPES:

C language is rich in its data types. The variety of data types available allows the programmer to select the type appropriate to the needs to the application. C has different data types for different types of data and can be broadly classified as:

1. Primary data types
2. Secondary data types



INTEGER DATA TYPE:

Integers are whole numbers with a range of values, range of values are machine dependent. Generally an integer occupies 2 bytes memory space. A signed integer uses one bit for storing sign and rest 15 bits for number.

To control the range of numbers and storage space, C has three classes of integer storage namely short int, int and long int. All three data types have signed and unsigned forms. A short int requires half the amount of storage than normal integer. Unlike signed integer, unsigned integers are always positive and use all the bits for the magnitude of the number. Therefore the range of an unsigned integer will be from 0 to 65535. The long integers are used to declare a longer range of values and it occupies 4 bytes of storage space.

Syntax: int <variable name>; like

```
int num1=20000;
```

```
short int num2=50;
```

```
long int num3=40000;
```

REAL(FLOATING POINT) DATA TYPE:

The float data type is used to store fractional numbers (real numbers) with 6 digits of precision. Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision and takes double space (8 bytes) than float. To extend the precision further we can use long double which occupies 10 bytes of memory space.

Syntax: float <variable name>; like

```
float num1=3.14;
```

```
double num2=65.777;
```

```
long double num3=123.45&78;
```

CHARACTER DATA TYPE:

Character type variable can hold a single character. As there are signed and unsigned int (either short or long), in the same way there are signed and unsigned chars; both occupy 1 byte each, but having different ranges.

Syntax: char <variable name>; like

```
char ch = 'a';
```

```
char c = '$';
```

VOID DATA TYPE:

The void type has no values therefore we cannot declare its variables as we did in case of integer and float. The void data type is usually used with function to specify its type. Like in our first C program we declared “main()” as void type because it does not return any value.

Data Type	Size(Bytes)	Format Specifier	Range
char or signed char	1	%c	-128 to +127
unsigned char	1	%c	0 to +255
int or signed int	2	%d	-32768 to +32,767
unsigned int	2	%u	0 to +65535
short int or signed short int	1	%d	-128 to +127
unsigned short int	1	%u	0 to +255
long int or signed long int	4	%ld	-2 ³¹ to +2 ³¹
unsigned long int	4	%lu	0 to +2 ³²
float	4	%f	-3.4E+38 to +3.4E+38
double	8	%lf	-1.7E+308 to +1.7E+308
long double	10	%Lf	-3.4E+4932 to +1.1E+4932

ENUMERATION (OR enum) IN C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

The keyword ‘enum’ is used to declare new enumeration types in C

```
/*An example program to demonstrate working  
of enum in C*/  
#include<stdio.h>
```

```
enum week{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

```
void main()  
{  
    enum week day;  
    day = Wed;  
    printf("%d\t",day);  
    day = Fri;  
    printf("%d\t",day);  
    day = Sun;  
    printf("%d\t",day);  
}
```

Output:

```
2    4    6
```

typedef in C

typedef keyword is used to assign a new name to a type. This is used just to prevent us from writing more.

For example, if we want to declare some variables of type unsigned int, we have to write unsigned int in a program and it can be quite hectic for some of us. So, we can assign a new name of our choice for unsigned int using typedef which can be used anytime we want to use unsigned int in a program.

Syntax: typedef current_name new_name;

```
typedef unsigned int ui;  
uint i, j;
```

Now, we can write ui in the whole program instead of unsigned int. The above code is the same as writing:

```
unsigned int i, j;
```

//Example prog. on typedef

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

```
void main() {  
    typedef unsigned int ui;  
    ui i = 5, j = 8;  
    printf("i = %d\n", i);  
    printf("j = %d\n", j);  
}
```

VARIABLES:

An entity that may vary during program execution is called a variable.

Syntax for variable declaration: data_type variable_name;

Ex: int n; float x,y,z;

Syntax for variable initialization: data_type variable_name=value;

Ex: int n=10; char a='#';

VARIABLE NAMES: Variable names are the names given to memory locations(variables). These locations can contain integer, real or character constants. An integer variable can hold only an integer constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

RULES FOR CONSTRUCTING VARIABLE NAMES:

- (a) A variable name is any combination of alphabets, numbers and only a special character underscore(_) is allowed.
- (b) The length of the variable name should not exceed 31 characters.
- (c) The first character in the variable name must be an alphabet or underscore.
- (d) Keywords cannot be used as variable names.
- (e) Upper case and lower case variables are different. Ex: TOTAL , total and Total are different variables.

Valid Examples: int area, Total;
int bas_sal, gross_sal;
char ch;
float _abc;
int Count_1;
double a3c;

Invalid Examples: int 2ab; (Variable name started with no)
float bas-sal, gross sal ;(Special symbols other than Underscore is used)
int void, if; (Keywords used as variable names)

C KEYWORDS:

Keywords are the words whose meaning has already explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the compiler. These keywords are also called as “Reserved Words”.

There are only 32 keywords in C.

Auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

A SIMPLE C PROGRAM:

Printing a line of text

1. `/* A first program in C */`
2. `#include<stdio.h>`
3. `void main()`
4. `{`
5. `printf (“Welcome to C!”);`
6. `}`

Output: Welcome to C.

Line 1: Begin with `/*` and end with `*/` indicating with these two lines are a comment. Programmers insert comments to document programs and improve program readability. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C compiler and do not produce any machine language object code. Comments also help other people read and understand (purpose of the program).

Line 2: This line tells the compiler to include the contents of standard input/output header file (`stdio.h`) in the program. This header file contains information and declarations used by the compiler when compiling standard input/output library functions such as `printf` and `scanf`.

Line 3: `main ()` Main is a special function defined in C. Every program must have `main()` function. The program execution starts from this function. The main function can be located anywhere in the program but the general practice is to place it as the first function for better readability.

Line 4: The left brace `{`, must begin the body of every function. A corresponding right brace must end each function. This pair of braces and the portion of the program between the braces are called a block.

Line 5: `printf` instructs the computer to perform an action, namely to print on the screen the string of characters between double quotes.

Note: Every C statement must end with semicolon; this `;` acts as a statement terminator.

C is a case-sensitive language and not space sensitive. Case-sensitive means upper case and lower case are different. All C statements are entered in small case letters only. C has no specific rules for the position at which a statement is to be written. That’s why it is often called a free-form language.

e.g.. If u write `Main ()` it gives error. Give `main ()`.

If we give `printf (“welcome to c”);` it is error free
because it is free-form language.

```

1)  /* Addition  Program */
2)  #include<stdio.h>
3)  void main()
4)  {
5)      int integer1,integer2,sum;          /* Declaration of variables */
6)      printf("Enter first integer\n");    /* Prompt */
7)      scanf("%d",&integer1);             /* read an integer */
8)      printf("Enter second integer\n");   /* prompt */
9)      scanf("%d",&integer2);             /*read an integer */
10)     sum=integer1+integer2;              /*assignment of sum */
11)     printf("Addition is %d \n",sum);     /* Print Sum */
12) }                                       /* End of the program */

```

Output: Enter first integer

10

Enter second integer

20

Addition is 30

int integer1,integer2,sum;

It is declaration of variables. The words integer1, integer2 and sum are the names of variables. A variable is a location in memory where a value can be stored for use by a program. This declaration specifies that the variables integer1, integer2 and sum are of type int which means that this variables will hold integer values i.e.. whole numbers such as 7, 0, 2345 etc(but not 2.3, 0.987). All variable must be declared with a name and a data type immediately after the left brace that begins the body of main before they can be used in a program.

Note: In C uninitialized variables contains garbage values.

scanf() Function:

Syntax: scanf("format string", list of address of arguments);

e.g... scanf("%d",&a);
scanf("%d%d",&x,&y);

%c character data

%d integer data

%f float data

The ampersand symbol & before each argument/variable name is an operator that specifies the variable name's address. We must always use this operator, otherwise unexpected results may occur.

MEMORAY CONCEPTS:

Variable names such as integer1, integer2 and sum correspond to locations of the computers memory. Every variable has a name, a type and a value.

In the above program when the statement

scanf("%d",&integer1);

is executed, the value typed by the user is place into a memory location to which the name integer1 has assigned. Suppose user enters the number 10 as the value for integer1. The computer will place 10 into location integer1 as shown below.

integer1

Garbage value

After declaring integer1

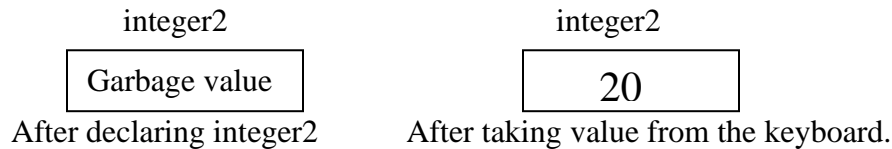
integer1

10

After taking value from the keyboard.

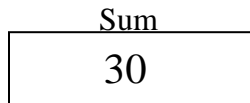
Suppose if user enters 20 as the value for integer2. The computer will place 20 into location

integer2 as shown below.



Once the program has obtained values for integer1 and integer2, it adds these values and places the sum into the variable sum. The statement

Sum= integer1 + integer2; finds the calculated sum of integer1 and integer2 and places it into location sum (destroying the value that may already in sum i.e garbage value).



printf() Function:

Syntax: printf("control string", list of arguments);

e.g... printf("The value of a=%d", a);
printf("Sum of %d and %d is %d\n", a,b,sum);

In the above program if printf("Sum is %d",sum); occurs it prints the corresponding value in the variable sum i.e 30 at the place where the format specifier %d is present in the control string .

SYNTAX AND LOGICAL ERRORS IN COMPILATION:

When a programmer writes a code in high-level language there are two types of errors that they might make: syntax errors and logical errors.

Syntax errors are mistakes such as misspelled keywords, a missing punctuation character, a missing bracket, or a missing closing parenthesis. If you try to execute a program that includes syntax errors, you will get error messages on the screen and the program won't be executed. You must correct all the errors and then try to execute the program again.

Logical errors are those errors that prevent your program from doing what you expected it to do. With logic errors you get no warnings at all. Your code may compile and run but the result is not the expected one. Logical errors are the most difficult errors to detect. You must revisit your program thoroughly to determine where your error is. For example consider a program that prompts the user to enter three numbers, and then calculates and displays their average value. The programmer, however, made a typographical error where he divides the sum of three numbers by 5 instead of 3. Of course the program is executed as usual without any error messages but also without the desired output.

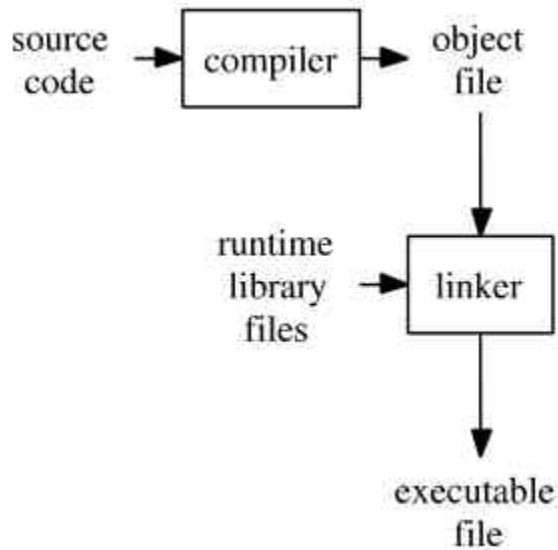
OBJECT AND EXECUTABLE CODE:

Computers do not understand human languages. In fact, at the lowest level, computers only understand sequences of bits 0's and 1's that represent operational codes (op codes for short). On the other hand, it would be very difficult for humans to write programs in terms of op codes. Therefore, programming languages were invented to make it easier for humans to write computer programs.

Programming languages are for humans to read and understand. The program (source code) must be translated into machine language so that the computer can execute the program (as the

computer only understands machine language

The following illustrates the programming process for a compiled programming language like C.



A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file. This process is diagrammed below.

OPERATORS IN C:

1. ARITHMETIC OPERATORS:

<i>Operator</i>	<i>Meaning</i>
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division (gives Quotient)
%	Modulo Division (gives Remainder)

```
#include<stdio.h>
void main ()
{
    int  months, days ;
    printf("Enter days\n") ;
    scanf("%d", &days) ;
    months = days / 30 ;
    days  = days % 30 ;
    printf("Months = %d  Days = %d", months, days) ;
}
```

Output: Enter days
 265
 Months = 8 Days = 25


```

#include <stdio.h>
void main()
{
int a = 100,b = 3,c;
c = a + b;
printf( "a + b = %d\n", c );
c = a - b;
printf( "a - b = %d\n", c );
c = a * b;
printf( "a * b = %d\n", c );
c = a / b;
printf( "a / b = %d\n", c );
c = a % b;
printf( "a % b = %d\n", c );
}

```

Output:a+b=103
 a-b=97
 a*b=300
 a/b=33
 a%b=1

2. RELATIONAL OPERATORS:

Executable C statements either performs actions(such as calculations or input or output of data) or make decisions. We might make a decision in a program, for example, to determine if a person is passed or failed in an exam. To check the condition(for example in if control structure) we use Relational operators.

<i>Operator</i>		<i>Meaning</i>
>	x > y	x is greater than y
<	x < y	x is less than y
>=	x >= y	x is greater than or equal to y
<=	x <= y	x is less than or equal to y
==	x == y	x is equal to y
!=	x != y	x is not equal to y

```

#include<stdio.h>
void main()
{
int a,b;
printf("Enter 2 numbers");
scanf("%d%d",&a,&b);
if(a==b)
printf("a is equat to b\n");
if(a!=b)
printf("a is not equat to b\n");
if(a>b)
printf("a is greater than b\n");
if(a<b)
printf("a is less than b\n");
}

```

Out put
Enter 2 numbers8 4
a is not equal to b
a is greater than b

3. LOGICAL OPERATORS:

C provides logical operators that may be used to form more complex condition by combining simple conditions. The logical operators are

<i>Operator</i>	<i>Meaning</i>
&&	logical AND
	logical OR
!	logical NOT

Like the simple relational expressions($x > y$), a logical expression also yields a value of zero or one.

AND operator:

(cond1 && cond2)

If both conditions are true the output of the logical AND expression is true(i.e it returns 1) and if any of the condition is false the output is false(i.e. it returns 0)

Example:

```
#include<stdio.h>
void main()
{
    int a,b,c;
    printf("Enter 3 numbers");
    scanf("%d%d%d",&a,&b,&c);
    if((a>b)&&(a>c))
        printf("a is bigger than b and c\n");
    if((b>a)&&(b>c))
        printf("b is bigger than a and c\n");
    if((c>a)&&(c>b))
        printf("c is bigger than a and b\n");
}
```

OR Operator:

(condt1 || cond2)

If any of the condition is true the output of the expression is true and the output will be false when both the conditions are false

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char ch;
    clrscr();
    printf("Enter alphabet:");
    scanf("%c",&ch);
    if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u')
        printf("You entered vowel");
    else
        printf("You entered consonant");
    getch();
}
```

```
}
```

Output: Enter alphabet: a
You entered vowel.

Not Operator:

(!conditon)

If the condition is true the output of the expression will be false and if the condition is false the output of the expression is true.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b;
    clrscr();
    printf("enter two no");
    scanf("%d %d",&a,&b);
    if(!(a>b)) //Here if the conditon a>b is true the total !(a>b) becomes false
    printf("b is greater");
    else
    printf("a is greater");
    getch();
}
```

4. ASSIGNMENT OPERATORS:

These ops. are used to assign the result of an expression to a variable . In addition to the usual assignment op. '=', C has a set of shorthand assignment ops. of the form.

v op=exp;

The operator op= is known as the shorthand operator. The above statement can be written as

v=v op (exp);

Ex: x=8; y=5;

x+=y+1; (here += is shorthand op. Similarly we can also use -=, *=, /=, %= as shorthand ops.)

The above statement is same as

x=x + (y+1); (here x value becomes 14)

5. INCREMENT AND DECREMENT OPERATORS:

C has two very useful operators called increment and decrement operators. The operator ++ add 1 and operator -- subtracts 1. Both are unary operators (because we are operating on one operand).

preincrement	++m	equivalent to
postincrement	m++	m=m+1; or m+=1;
predecrement	--m	equivalent to
postdecrement	m--	m=m-1; or m-=1;

we use increment and decrement statements in for and while loops extensively.

When m++ and ++m mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following programs:

```
1) #include<stdio.h>
#include<conio.h>
void main()
{
    int i=5;
```

```
2) #include<stdio.h>
#include<conio.h>
void main()
{
    int i=5,j;
```

```

clrscr();
++i;
printf("%d\t",i);
i=5;
i++;
printf("%d",i);
getch();
}

```

Output: 6 6

```

clrscr();
j=++i;
printf("%d %d \n",i,j);
i=5;
j=i++;
printf("%d %d",i,j);
getch();
}

```

Output: 6 6
6 5

If we observe the second program for preincrement of i first i is incrementing and afterwards it is assigning to j and in the postincrement case first the value of i is assigned to j(i.e 5) and afterwards the value in the i is incrementing. Therefore we get the output 6 6 and 6 5.

6. CONDITIONAL OPERATOR:(Ternary Operator)

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

conditional expression ? expression1:expression2

The conditional expression is evaluated first. If the result is nonzero(true) expression1 is evaluated and returned, otherwise expression 2 is evaluated and its value is returned.

Example:

```

1) #include< stdio.h>
   #include<conio.h>
   void main()
   {
       int a,b,c;
       clrscr();
       printf("Enter two no's\n");
       scanf("%d %d",&a,&b);
       c = (a>b?a:b);
       printf("%d is greater",c);
       getch();
   }

```

OUTPUT: Enter two no's
56 78
78 is greater

```

2) #include<stdio.h>
   #include<conio.h>
   void main()
   {
       int a,b,c,max;
       clrscr();
       printf("Enter three no's\n");
       scanf("%d %d %d",&a,&b,&c);
       max=(a>b?(a>c?a:c):(b>c?b:c));
       printf("%d is greater",max);
       getch();
   }

```

OUTPUT: Enter three no's
34 55 23
55 is greater.

7. BITWISE OPERATORS:

These ops. are used for manipulation of data at bit level. These ops. are used for testing the bits or shifting them right or left.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise Exclusive OR
~	One's complement
<<	Shift left
>>	Shift right

```

1 1 1 1 1 0 1 1 Operand1
& 1 0 1 0 1 1 1 0 Operand2
-----

```

```

1 0 1 0 1 0 1 0 Operand1 & Operand2
If any one or both the operands are 0 then result is 0 otherwise 1.

```

```

1 0 1 0 0 0 0 0 Operand1
| 1 0 0 0 0 0 1 1 Operand2
-----

```

```

1 0 1 0 0 0 1 1 Operand1 | Operand2
If any one or both the operands are 1 then result is 1 otherwise 0.

```

```

1 0 1 0 1 1 0 0 Operand1
^ 1 0 0 1 1 1 1 0 Operand2
-----

```

```

0 0 1 1 0 0 1 0 Operand1 ^ Operand2
If both the operands are same then result is 0 otherwise 1.

```

```

0 0 1 1 1 0 0 0 Operand
-----
1 1 0 0 0 1 1 1 ~Operator
1 becomes 0 and 0 becomes 1.

```

```

0 0 0 1 1 0 0 1 Operand
-----
0 1 1 0 0 1 0 0 Operand<<2 (Bits shifted towards left for 2 positions)

```

```

1 0 1 1 0 0 0 1 Operand
-----
0 0 0 1 0 1 1 0 Operand>>3 (Bits shifted towards right for 3 positions)

```

8. SPECIAL OPERATORS:

These are comma op., sizeof op., pointer ops.(& and *) and member selection ops.(. and ->).

Comma operator: It can be used to link the related expressions together. These expressions are evaluated left to right and the value of right most expression is the value of the combined expression.

Ex: v=(x=10, y=5, x+y);

Here the value of the right most expression is stored in variable v i.e, 15.

sizeof operator: It is used to find out size of any data type. It will take data type as argument and return size of it in number of bytes whose datatype is unsigned long int.

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

```
void main()
```

```
{
    printf("%lu\n", sizeof(char));
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(float));
    printf("%lu", sizeof(double));
}
```

Output:

1
4
4
8

TYPE CONVERSION OR TYPE CASTING:

Implicit type casting: Changing one data type to other is called type casting. Type casting which is automatically done by the compiler is called implicit type casting.

Ex: int i; float b;

i=3.5; //Here i is int and 3.5 is float. So the compiler will implicitly convert float value(3.5) to int value(3) and then assigns it to variable i.

b=30; //Here b is float and 30 is int. So the compiler will implicitly convert int value(30) to float value(30.000000) and then assigns it to variable b.

Explicit Type Casting: In this the type casting is purposely done by the programmer.

Syntax for explicit casting is

	(type-name)expression
Ex:	Action
x=(int)7.5	7.5 is converted to integer(7) by truncation.
a=(int)21.3/(int)4.5	Evaluated as 21/4 and the result would be 5.

// C program to demonstrate explicit type casting

```
#include<stdio.h>
void main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);
}
```

Output:

sum = 2

OPERATOR PRECEDENCE:

Operator precedence describes the order in which C reads expressions. For example, the expression $a=4+b*2$ contains two operations, an addition and a multiplication. Does the C compiler evaluate $4+b$ first, then multiply the result by 2, or does it evaluate $b*2$ first, then add 4 to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associativity" column on the right gives their evaluation order.

Operator Type	Operator	Associativity
Primary Expression Operators	() [] . -> <i>expr</i> ++ <i>expr</i> --	left-to-right
Unary Operators	* & + - ! ~ ++ <i>expr</i> -- <i>expr</i> (<i>typecast</i>) sizeof	right-to-left
Binary Operators	* / %	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	&&	
Ternary Operator	?:	right-to-left
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= =	right-to-left
Comma	,	left-to-right

PSEUDO CODE:

Pseudo code is an artificial and informal language that helps programmers develop algorithms. Pseudo Code is similar to everyday English; it is convenient and user-friendly although it is not an actual computer programming language.

Pseudo code programs are not actually executed on computers. Rather, they merely help the programmers “think out” a program before attempting to write it in a programming language such as C.

Example: If student’s percentage is greater than 40
 Print “passed”
 Else
 Print “failed”

ALGORITHM:

It is a step by step description of the problem in a language similar to English, but not in full sentences, but as a set of commands.

Properties of the algorithm:

1. Finiteness: An algorithm must always terminate after a finite number of steps.
2. Definiteness: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. Input: An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.
4. Output: An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.
5. Effectiveness: An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

Method of writing Algorithms for:

1) Selection control Statements (Branching)

<i>Selection Control</i>	<i>Example</i>	<i>Meaning</i>
IF (Condition) THEN ... ENDIF	IF (X > 10) THEN Y=Y+5 ENDIF	If condition X>10 is True execute the statement between THEN and ENDIF
IF (Condition) THEN ... ELSE ENDIF	IF (X > 10) THEN Y=Y+5 ELSE Y=Y+8 Z=Z+3 ENDIF	If condition X>10 is True execute the statement between THEN and ELSE otherwise execute the statements between ELSE and ENDIF

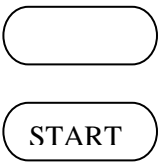
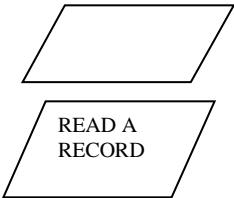
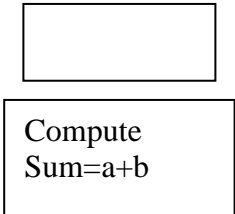
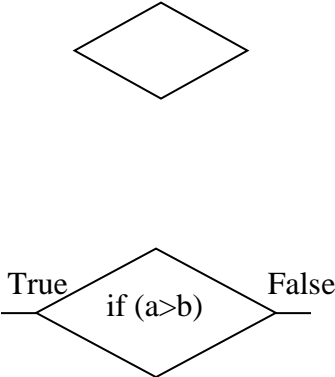
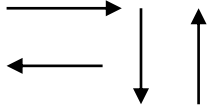
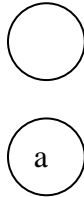
2) Repetition control Statements (Looping)


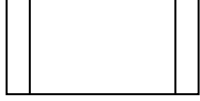
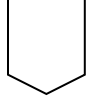
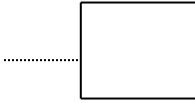
<i>Selection Control</i>	<i>Example</i>	<i>Meaning</i>
WHILE (Condition) DO ENDDO	WHILE (X < 10) DO print x x=x+1 ENDDO	Execute the loop as long as the condition is TRUE

FLOW CHART:

As the name suggests, it is a chart to describe the flow of the solution process. It is also defined as pictorial representation of an algorithm. You breakup the solution into simple steps and connect them to describe the flow so that the last of these blocks leads you to the solution.

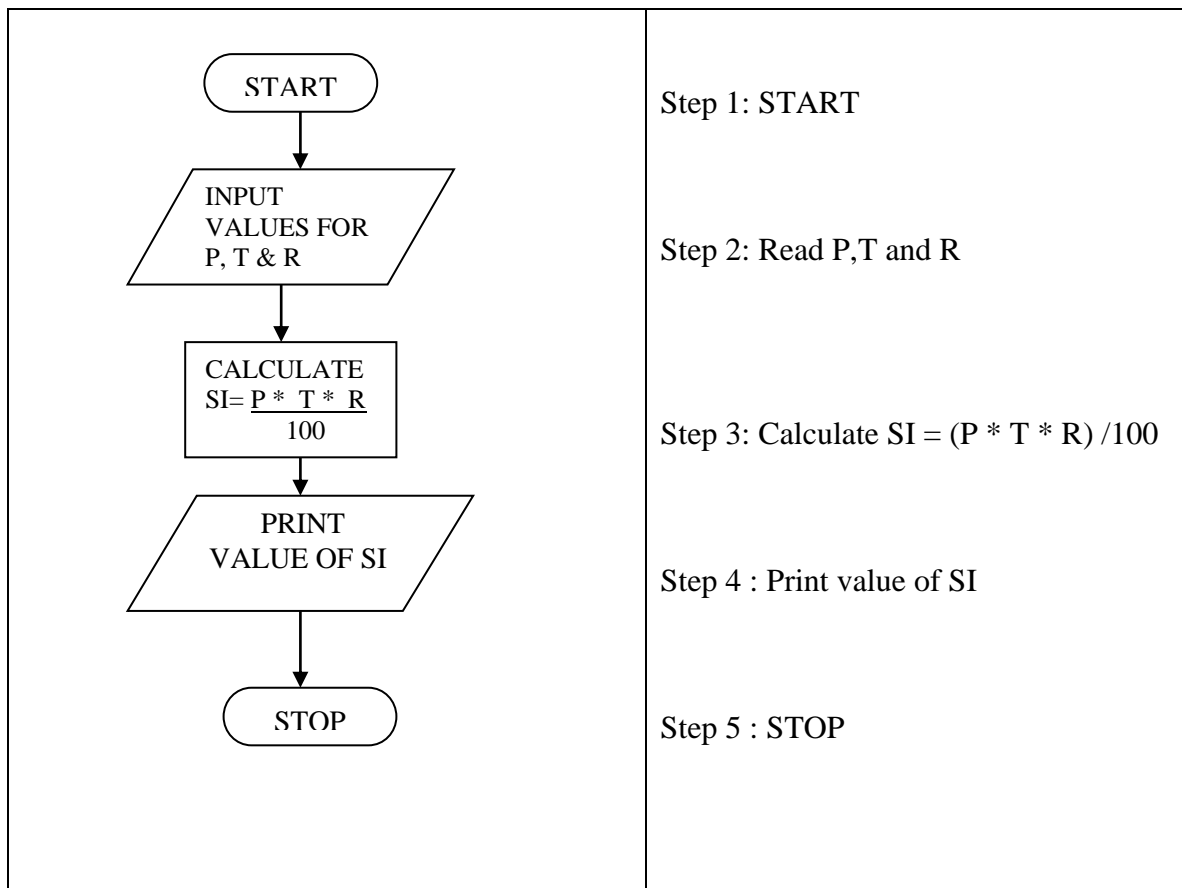
We briefly describe the commonly used symbols to represent these blocks and how to combine them to form the solution.

<p>1. <i>Terminal</i>: 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. <i>Input/Output</i>: 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>	
<p>3. <i>Processing</i>: 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.</p>	
<p>4. <i>Decision</i> : 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.</p>	
<p>5. <i>Flowline</i>: 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. <i>Connector</i>: 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>	

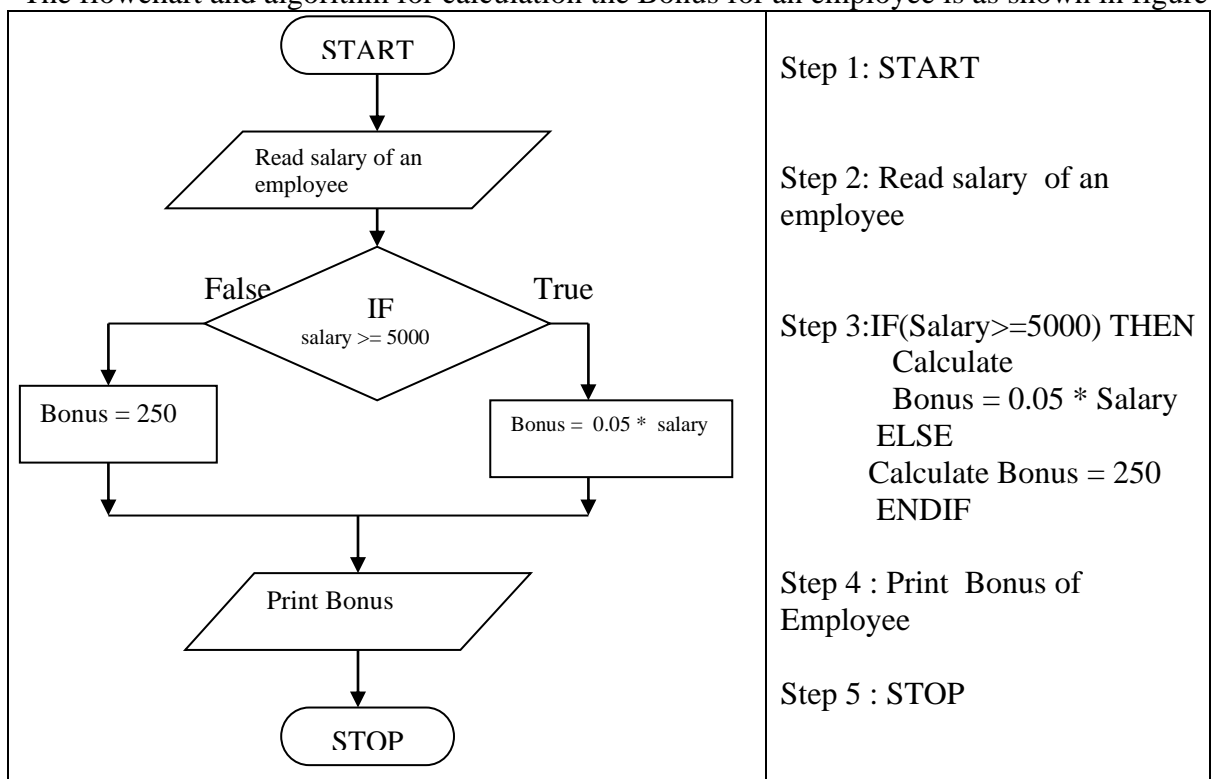
7. <i>Preparation:</i> This symbol is used for the house-keeping operations. All those operations, which are done before starting the actual processing of data, come in this category. For example, setting the number of students to zero before processing of data starts.	
8. <i>Predefined:</i> This symbol 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.	
9. <i>Offpage Connector:</i> This symbol is used in place of connector when the exit and the corresponding entry point are shown on different pages in the flowchart.	
10. <i>Annotation, Comments:</i> 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.	

Some of the examples below will classify the concepts.

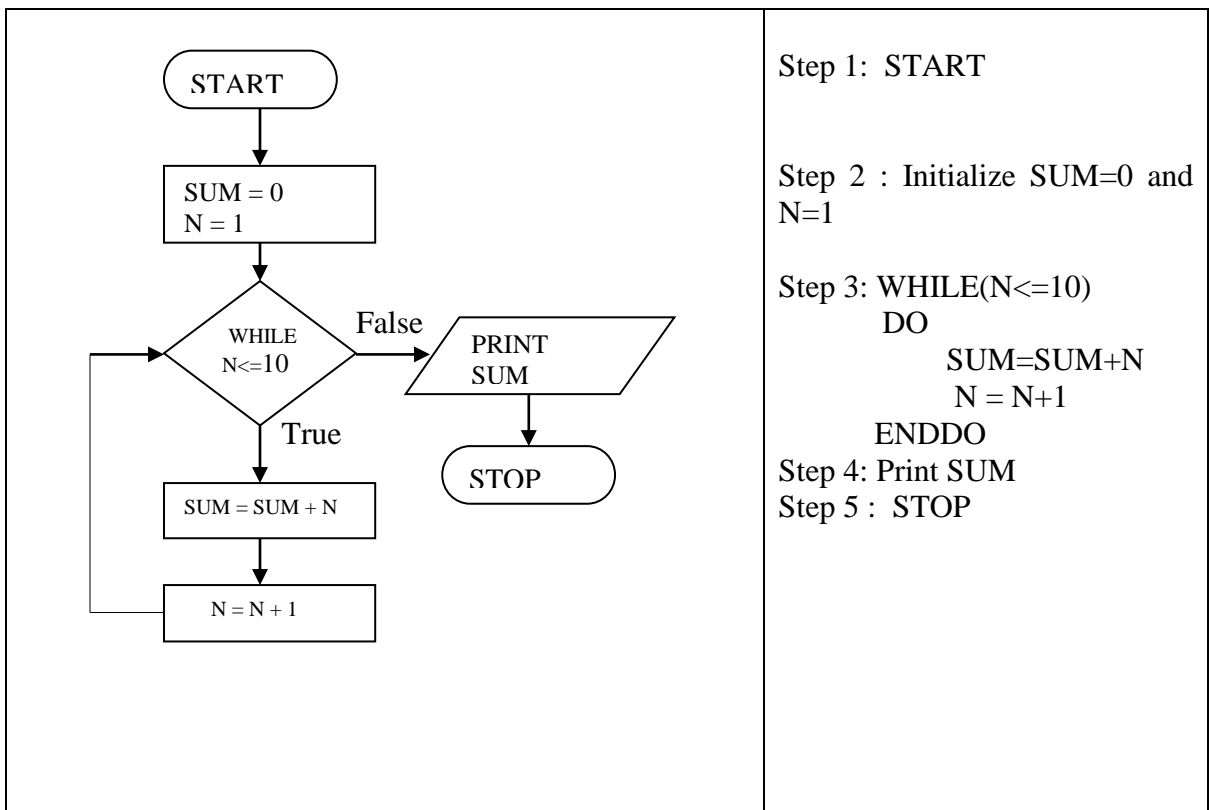
Example a) Simple interest is calculated using the formula $SI = (P * T * R) / 100$



Example b) Victor Construction Company plans to give a 5% year-end Bonus to each of its employees earning Rs. 5,000 or more per year, and a fixed Rs. 250 bonus to the other employees. The flowchart and algorithm for calculation the Bonus for an employee is as shown in figure



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



CONTROL STRUCTURES:

In C programming, “**control structures**” are the **structures** used to **control** the flow of execution of a program.

There are three control structures in C

- 1) Sequential control structure
- 2) Selection control structure (Branching)
- 3) Repetition control structure (Looping)

1) SEQUENTIAL CONTROL STRUCTURE:

This is default structure. Unless directed otherwise (means selection and repetition), the computer automatically executes C statements one after the other in the order in which they are written.

2) SELECTION CONTROL STRUCTURE:

C provides following types of selection structures.

- a) if
- b) if-else
- c) nested if-else
- d) else if ladder
- e) switch

a) The if selection structure:

if selection structure either performs (selects) an action if a condition is true or skips the

action if the condition is false.

```
if(condition)
{
    if-block statement(s)
}
statement-x;
```

The statement block may be a single statement or a group of statements. If the condition is true, the if-block statement(s) will be executed; otherwise the if-block statement(s) will be skipped and the execution will jump to the statement-x. Remember, when the condition is true both the if-block statement(s) and the statement-x executes in sequence.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num1,num2;
    printf("Enter the numbers\n");
    scanf("%d %d",&num1,&num2);
    if(num1==num2)
    printf("%d is equal to %d",num1,num2);
    if(num1>num2)
    printf("%d is greater than %d",num1,num2);
    if(num1<num2)
    printf("%d is less than %d",num1,num2);
}
```

output : Enter the numbers:
3 4
3 is less than 4

b) The if-else selection structure:

The if-else structure performs an action if the condition is true and performs a different action if condition is false.

```
if(condition)
{
    if-block statement(s)
}
else
{
    else-block statement(s)
}
statement-x;
```

If the condition is true, then the if-block statement(s), immediately followed by the if condition are executed. Otherwise the else-block statement(s) are executed.

% symbol is a modulo division operator which gives remainder whereas / operator gives quotient.

e.g a=15 b=2 a%b=1 and a/b=7

Example:

```
/* program to find whether a number is even or odd */
```

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

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

```
void main()
```

```
{
```

```
    int n;
```

```
    clrscr();
```

```
    printf("Enter the nr ");
```

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

```
    if(n%2==0)
```

```
        printf("The nr is even");
```

```
    else
```

```
        printf("The nr is odd");
```

```
    getch();
```

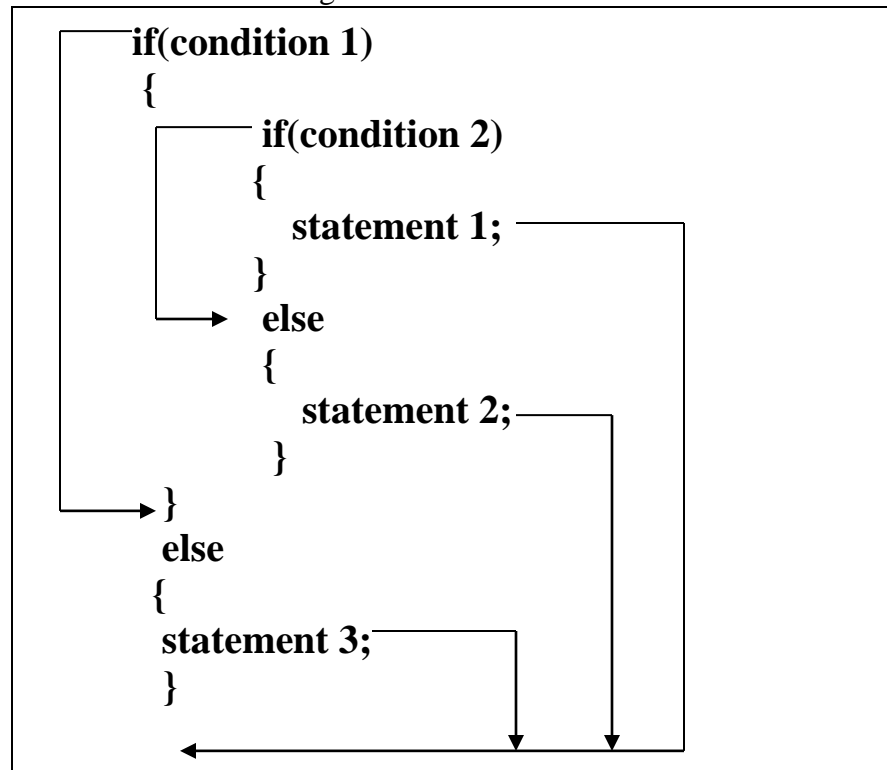
```
}
```

OUTPUT : Enter the nr 21

The nr is odd

c) Nested if-else selection structure:

When a series of decisions involved, we may have to use more than one if...else statement. We can write an entire if-else construct within either the body of the if statement or the body of an else statement. This is called nesting of if else.



If the condition1 is false, the statement 3 will be executed; otherwise it continues to perform the second test. If the condition 2 is true, the statement 1 is evaluated; otherwise the statement 2 will be evaluated and then the control is transferred outside.

Example:

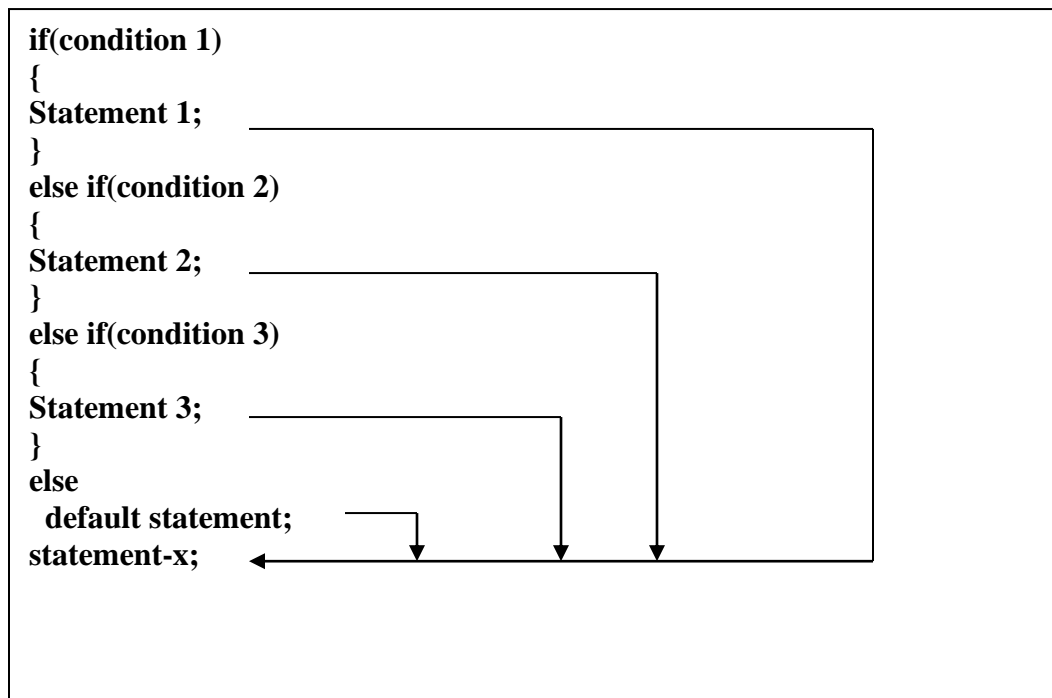
```

/*Program to find biggest of three no's */
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter values for a, b and c:\n");
    scanf("%d%d%d",&a,&b,&c);
    if(a>b)
    {
        if(a>c)
            printf("a is biggest");
        else
            printf("c is biggest");
    }
    else
    {
        if(b>c)
            printf("b is biggest");
        else
            printf("c is biggest");
    }
    getch();
}

```

d) The else-if ladder selection structure:

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with else is an if.



Example:

```

/*Program to find a roots of quadratic equation. */
#include <stdio.h>
#include <math.h>
void main()
{
float root1, root2,D;
float a, b, c; /*the coefficients*/
printf("\n Enter the coefficients a, b and c\n");
scanf ("%f %f %f", &a, &b, &c);
D = b*b - 4 * a * c;
if (D < 0)
printf("The roots are imaginary");
else if (D == 0)
{
printf ("\n the roots are real and equal:");
root1 = -b / (2*a);
root2 = root1;
printf ("\n root1 = %f and root2 = %f", root1, root2);
}
else
{
printf ("\n the roots are real and unequal:");
root1 = (- b + sqrt(D)) / (2*a);
root2 = (- b - sqrt(D)) / (2*a);
printf ("\n root1 = %f and root2 = %f", root1, root2);
}
getch();
}

```

e) The switch selection structure:

The switch selection structure performs one of many different actions depending on the value of an expression. The switch structure is called multiple-selection structure because it selects among many different actions.

```

switch (expression)
{
    case const1:
        block-1;
        break;
    case const2:
        block-2;
        break;
    .....
    .....
    default :
        default block;
        break;
}
statement-x;

```

The expression is an integer expression or characters. const1,const2.... are constant expressions(evaluable to an integral constant) and are known as case labels. Each of these values

should be unique within a switch statement. block-1, block-2... are statements lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a colon(:)

When the switch is executed, the values of the expression is successively compared against the values const1, const2.... if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

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

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action take place if all matches fail and the control goes to the statement-x.

Example:

```
/* Calculator program using switch case */
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,n;
    clrscr();
    printf("Enter two no's\n");
    scanf("%d %d", &a,&b);
    printf("1.Addition\n2.Substraction\n3.Multiplication\n4.Division\n");
    scanf("%d",&n);
    switch(n)
    {
        case 1:c=a+b;
            printf("Addition=%d",c);
            break;
        case 2:c=a-b;
            printf("Subtraction=%d",c);
            break;
        case 3:c=a*b;
            printf("Multiplication=%d",c);
            break;
        case 4:c=a/b;
            printf("Division=%d",c);
            break;
        default: printf("Invalid Option\n");
            break;
    }
    getch();
}
```

3) REPETITION CONTROL STRUCTURE:

C provides three repetition structures

- a) while loop
- b) do-while loop
- c) for loop

A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.

a) The while loop:

The basic format of while loop is

```
Initialization;
while(condition)
{
    body of loop
incrementation/decrementation;
}
```

The while is entry-controlled loop statement. The test-condition is evaluated and if the condition is true then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the next statements. The braces needed only if the body contains one or more statements.

<pre>// Prog. to Display natural numbers up to 50 #include<stdio.h> #include<conio.h> void main() { int i; clrscr(); i=1; printf("Display of natural numbers up to 50\n"); while(i<=50) { printf("%d\t",i); i++; } getch(); }</pre> <p>OUTPUT: Display of natural numbers up to 50</p> <pre>1 2 3 4 5 6.....48 49 50</pre>	<pre>//Program to find Factorial of a number #include<stdio.h> #include<conio.h> void main() { int fact=1,i=1,n; clrscr(); printf("Enter the number: "); scanf("%d",&n); while(i<=n) { fact=fact*i; i++; } printf("Factorial = %d",fact); getch(); }</pre> <p>OUTPUT: Enter the number: 5</p> <p>Factorial = 120</p>
---	---

b) The do while loop:

The general format of do while is

```

Initialization;
do
{
    body of loop
incrementation/decrementation;
}while(condition);

```

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

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

Example:

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,n;
    clrscr();
    do
    {
        printf("Enter two no's\n");
        scanf("%d %d", &a,&b);
        printf("1-Addition\n2-Substraction\n3-Multiplication\n4-Division\n5-Exit\n");
        scanf("%d",&n);
        switch(n)
        {
            case 1:c=a+b;
                printf("%d",c);
                break;
            case 2:c=a-b;
                printf("%d",c);
                break;
            case 3:c=a*b;
                printf("%d",c);
                break;
            case 4:c=a/b;
                printf("%d",c);
                break;
        }
    } while(n<5);
    getch();
}

```

Difference between while and do while

while	do while
1) It is entry-controlled loop	1) It is exit-controlled loop
2) condition is written before the loop	2) condition is written after the loop
3) There is no guaranty that the loop will get executed for at least one time.	3) There is a guaranty that the loop will get executed for at least one time.
4) only one keyword used i.e., while	4) Two keyword used i.e., do and while
5) There is no semi colon after while condition	5) There is a semi colon after while condition

c) The for loop:

The for loop is another entry-controlled loop that provides a more concise loop control structure.

```
for(initialization;condition;incrementation/decrementation)  
{  
body of loop  
}
```

The execution of the for statement is as follows:

- 1) Initialization of the control variables is done first, using assignment statements such as `i=1` and `count=0`. The variable `i` and `count` are known as loop-control variables.
- 2) The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as `i<10` that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately followed the loop.
- 3) When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as `i=i+1` and the new value of the control variable is again tested to see whether it satisfied the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues still the value of the control variable fails to satisfy the test-condition.

Example:

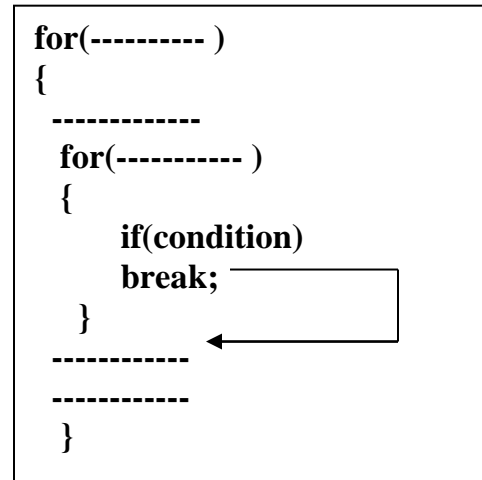
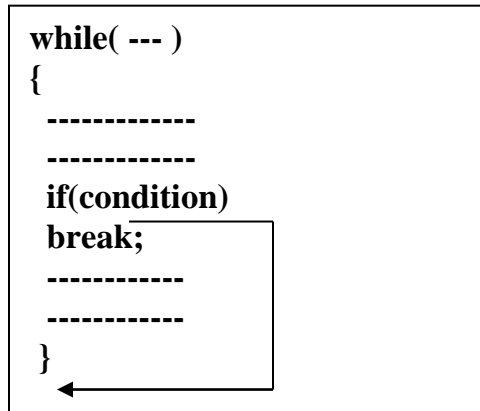
```
/* program to generate the table of a given no */  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int n,i,a;  
    clrscr();  
    printf("Enter the no\n");  
    scanf("%d",&n);  
    for(i=1;i<=10;i++)  
    {  
        a=n*i;  
        printf("%d * %d = %d",n,i,a);  
    }  
    getch();  
}
```

Unconditional Branching:

1) The break Statement:

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, break will exit only a single loop. The syntax of the break statement is

break;



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1;i<=10;i++)
    {
        if(i==5)
            break;
        printf("%d\t",i);
    }
    printf("\nBroke out of the loop at i = %d\n",i);
    getch();
}
```

OUTPUT: 1 2 3 4
 Broke out of the loop at i=5

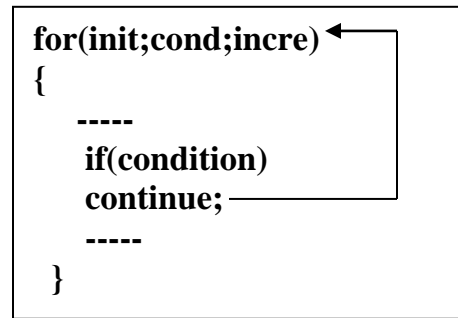
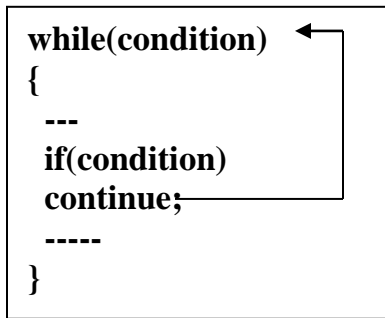
2) The continue Statement:

When the continue statement is encountered in the loop, as the name implies, it causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler, “SKIP THE FOLLOWING STATEMENTS AND CONTINUE THE NEXT ITERATION”. The syntax of the continue statement is

continue;

In while and do loops, continue causes the control to go directly to the test-condition and then to continue the iteration process. In the case of for loop, the increment section of the loop is

executed before the test-condition is evaluated.



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1;i<=10;i++)
    {
        if(i==5)
        continue;
        printf("%d\t",i);
    }
    printf("\nUsed continue to skip printing the value 5");
    getch();
}
```

OUTPUT: 1 2 3 4 6 7 8 9 10
 Used continue to skip printing the value 5

ONE DIMENTIONAL ARRAYS:

INTRODUCTION

The variables are capable of holding only one value at a time. For example

```
#include<stdio.h>
void main()
{
    int x;
    x=5;
    x=10;
    printf("%d",x);
}
```

The output of the program is 10 because when the value of 10 is assigned to x, the earlier value of x i.e. 5 is overwritten (the value 5 is lost). Thus ordinary variables hold only one value at a time.

There are some situations in which we would want to store more than one value at a time in a single variable.

For example, suppose I want to store the marks obtained by 100 students in a class. In such case we have two options to store these marks in memory.

- 1) Construct 100 variables to store marks obtained by 100 student's i.e each variable containing one student's marks.
- 2) Construct one variable capable of storing all the hundred values.

Obviously, the second option is better because it would be much easier to handle one

variable than handling 100 variables. So C language provides us a data type called array to store values of similar data type.

Definition: An array is a collection of elements of similar data type which are stored in consecutive memory locations. Array elements are grouped under a common name.

ARRAY DECLARATION:

The array is declared same as other variables before it is used in the program. The general format (Syntax) is

data_type arr_name[size];

data_type refers to the type of the data elements stored in array.

arr_name is the name of the array.

size indicates the maximum size of the array.

e.g int a[100]; /*an integer type array of size 100*/

 char name[20]; /*a char type array of size 20*/

Here, int specifies the type of the variable, just as it does with the ordinary variables and a specifies the name of the variable. The number 100 tells how many elements of the type int will be in our array. The [] tells the

compiler that we are dealing with an array.

NOTE: An array declared of one data type cannot store values of another data type.

ARRAY INITIALISATION

Till the array elements are not give any specific values, they are supposed to contain garbage values. To initialize arrays the following is the general format

data_type arr_name[size]={list of values separated by commas};

e.g. int a[10]= {1,2,3,4,5,6,7,8,9,10};

 char ch[5]={‘a’,‘e’,‘i’,‘o’,‘u’};

 int x[]={23,12,14,15};

If the array is initialized where it is declared mentioning the size of the array is optional as in 3rd example .

After declaration of the array the array elements are stored in contiguous memory locations in memory. For example

 int arr[7]={45,98,76,86,90,56,78};

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]
45	98	76	86	90	56	78
1000	1002	1004	1006	1008	1010	1012

14 bytes get immediately reserved in memory. 14 bytes because each of the 7 integers would be 2 bytes long.

Memory allocated= no of elements*sizeof(datatype);

Accessing Elements of an Array

Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, the number in the square brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Thus, arr[2] is not the second element of the array but the third.

Entering data into an array

For example if I want to take marks of the student in to an array. Here is the code that places data into the array.

```
int marks[6],i;
printf("Enter the marks\n");
for(i=0;i<6;i++)
{
```

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

We use for loop for reading elements into array. The first time through the loop, i has a value 0, so the scanf() statement will cause the value typed first to be stored in the array at marks[0] because we are passing the address of this particular array position to the scanf() function(i.e &marks[0]). This process will be repeated until i becomes 6 (i.e Until the 6 elements are stored in array marks[])

EXAMPLE 1: #include<stdio.h>
#include<conio.h>
void main()
{
int a[100],i,n;
clrscr();
printf("HOW MANY ELEMENT U WANT TO STORE INTHE ARRAY\n");
scanf("%d",&n);
printf("ENTER THE ELEMENTS\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);/*Reading elements into array*/
printf("CONTENTS OF THE ARRAY ARE\n");
for(i=0;i<n;i++)
printf("a[%d] = %d\n",i,a[i]);/*retrieving data from array*/
getch();
}

OUTPUT: HOW MANY ELEMENTS U WANT TO STORE IN ARRAY

```
5
ENTER THE ELEMENTS
1 2 3 4 5
CONTENTS OF THE ARRAY ARE
a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5
```

TWO DIMENSIONAL ARRAYS:

So far we have looked at arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The maximum limit of dimensions is of compiler dependent.

Two-dimensional arrays are declared as follows

```
data_type array_name[row_size][column_size];
```

Example: int arr[3][4];

Initializing a two-dimensional array is as follows

```
int arr[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12} };
```

Or even this would work.....

```
int arr[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

NOTE: It is important to remember that while intialising an array it is necessary to mention the second(column) dimension, whereas the first dimension(row) is optional.

Thus the declaration

```
int arr[2][3]={12,34,23,45,56,45};
```



```

    int arr[][3]={ 12,34,23,45,56,45};
are acceptable
whereas,
    int arr[][]={ 12,34,23,45,56,45};
    Int arr[2][]={ 12,34,23,45,56,45};
would never work;

```

Example: Program to read matrix into 2-dimensional array and print.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a[20][20],i,j,m,n;
    clrscr();
    printf("Enter the order of the matrix\n");
    scanf("%d %d",&m,&n);
    printf("Enter the elements in to array\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    }
    printf("The matrix elements are\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d\t",a[i][j]);
        printf("\n");
    }
    getch();
}

```

OUTPUT: Enter the order of matrix
3 3
Enter the matrix elements into array
1 2 3
4 5 6
7 8 9
The matrix elements are
1 2 3
4 5 6
7 8 9

MEMORY MAP OF A 2 DIMENSIONAL ARRAY

The elements of the array of the above program cannot store in rows and columns in memory. The array arrangement is only conceptually true. This is because memory doesn't contain rows and columns. In memory whether it is a one-dimensional or two dimensional array the array elements are stored in one continuous memory locations. The arrangement of array elements of a two dimensional array in memory is shown below:

Suppose if I initialize an array

```
int a[4][3]={ { 23,45,34},{ 12,67,98},{ 22,16,29},{ 73,75,13} };
```

the elements are stored in memory as follows

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]	a[3][0]	a[3][1]	a[3][2]
23	45	34	12	67	98	22	16	29	73	75	13
1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

CHARACTER ARRAYS AND STRINGS:

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings. Many languages internally treat strings as character arrays but somehow conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text such as words and sentences.

SINGLE DIMENSIONAL CHARACTER ARRAYS

The purpose of character arrays is to hold together a group of characters that represent a word or a line of text.

DECLARATION:

Char array_name[size];

Example: char name[20];

INITIALIZATION:

Character arrays may be initialized when they are declared. The following is the form to initialize character array.

Char a[20]={‘H’,’Y’,’D’,’E’,’R’,’A’,’B’,’A’,’D’};

The memory allocated for this is as follows

H	Y	D	E	R	A	B	A	D
---	---	---	---	---	---	---	---	---	-------

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]

Each character in array occupies one byte of memory.

Example 1:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char text[100],ch;
    int i=0,sz;
    printf("Enter the line of text(ctrl+z to stop)\n");
    while((ch=getchar())!=EOF)
    {
        text[i]=ch;
        i++;
        if(i==100)
            break;
    }
    sz=i;
    for(i=0;i<sz;i++)
```

```
printf("%c",text[i]);
getch();
}
```

OUTPUT: Enter the line of text

I love my country ^z

I love my country

STRINGS

Definition: A string is an array of characters terminated with a null character('\0').

Char city[]="HYDERABAD"

or

Char city[]={'H','Y','D','E','R','A','B','A','D','\0'};

In first case when the compiler assigns string constant to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore the size of the string should be equal to the maximum number of characters in the string plus one.

In second case when we initialize a string array by listing its elements, we must supply explicitly the null terminator ('\0').

The null character looks like two characters, but it is actually only one character, with the \ indicating that what follows it is something special. '\0' is called null character. Note that '\0' and '0' are not same. ASCII value of '\0' is 0, where as ASCII value of '0' is 48. The terminating null character is important, because it is the only way the function that work with a string can know where the string ends. In fact, a string not terminated by a '\0' is not really a string, but merely a collection of characters.

The following is the internal representation of string array.

H	Y	D	E	R	A	B	A	D	\0
---	---	---	---	---	---	---	---	---	----	-------

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

EXAMPLE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[20]="INDIA";
    int i=0;
    clrscr();
    while(name[i]!='\0')
    {
        printf("%c",name[i]);
        i++;
    }
    getch();
}
```

OUTPUT: INDIA

READING STRINGS FROM TERMINAL

A string can be assigned to an array using scanf. For example, the following statement

assigns a string to character array word[20].

```
scanf("%s",word);
```

The string entered by the user is stored in word(Note that word is an array, which is, of course, a pointer, so the & is not needed with argument word). Function scanf will read characters until space, new line or end-of-file indicator is encountered. For example see the following program in which it reads only one word because space appeared after one word.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char word[20];
    clrscr();
    printf("Enter the text\n");
    scanf("%s",word);
    printf(" The entered text is \n");
    printf("%s",word);
    getch();
}
```

OUTPUT: Enter the text
ANDHRA PRADESH
The entered text is
ANDHRA

The output is only one word i.e. ANDHRA because after that word space appears. Since scanf function will read characters until a space, new line or EOF indicator is encountered.

We have to use printf() function with %s format to print strings to the screen. The %s can be used to display an array of characters that is terminated by the null character('\0').

STRING MANIPULATION FUNCTIONS OF THE STRING HANDLING LIBRARY:

The string handling library provides many useful functions for manipulating strings. Following are the most commonly used string – handling functions. These functions are defined in string.h header file.

strlen():

This function counts and returns the number of characters in a string.

The function prototype of this function is

```
int strlen(const char *s)
```

It returns length of the string therefore the return type of the function is int and it takes string constant as argument.

Example: a) len=strlen("INDIA");

b) char str[10]="INDIA"

```
int len;
```

```
len=strlen(str);
```

strrev():

This function is used to reverse the given string. Reversed string will be placed in the same array. The prototype of this function is

```
char *strrev(char *);
```

strcpy():

This function copies the contents of one string into another. The base of the source and the target string should be supplied to this function. The prototype of this function is

```
char *strcpy(char *target,const char *source);
```

It takes the general form

```
strcpy(str1,str2);
```

The above function assigns the contents of string2 to string1. Str2 may be a character array variable or a string constant. For example

```
char city1[20],city2[20]="CULCUTTA";  
strcpy(city1,"DELHI");
```

will assign the sting "DELHI" to the string array city1. Similarly statement

```
strcpy(city1,city2);
```

will assign the contents of the string variable city2 to the string variable city1.

NOTE: The size of the target string should be greater or equal to the size of the source string.

strcat():

This function concatenates the source string at the end of the target string. The function prototype is

```
char *strcat(char *target,const char *source);
```

It takes the following form

```
strcat(target,source);
```

Here source string is appended to target string. It does so by removing the null character at the end of the target string and placing source string from there. The source string remains unchanged.

For example,

```
char source[20]="John";  
char target[20]="Hello";  
strcat(target,source);
```

On concatenating will result into target string "HelloJohn". That is source string is added at the end of target string. So target string containing the combined string.

C allows the nesting of strcat function

Example: `strcat(strcat(str1,str2),str3);`

Is allowed and concatenates all the three strings together. The result is stored in string 1.

strcmp():

This is a function which compares two strings to find out whether they are similar or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, strcmp() returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. The prototype of this function is

```
int strcmp(const char *s1,const char *s2);
```