# COMPUTER PROGRAMMING

## LECTURE NOTES

## UNIT-I

## INTRODUCTION TO COMPUTERS

### COMPUTER SYSTEMS

"A Computer is an electronic device that stores, manipulates and retrieves the data."

We can also refer computer computes the information supplied to it and generates data.

A System is a group of several objects with a process. For Example: Educational System involves teacher, students (objects). Teacher teaches subject to students i.e., teaching (process). Similarly a computer system can have objects and process.

The following are the objects of computer System

a) User ( A person who uses the computer)
b) Hardware
c) Software

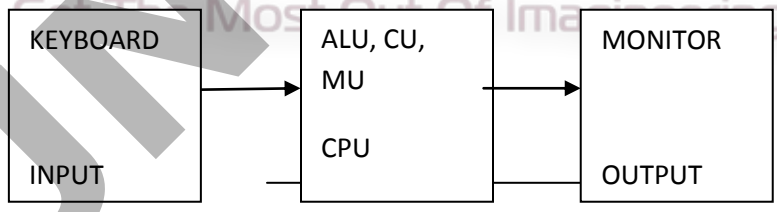
Hardware: Hardware of a computer system can be referred as anything which we can touch and feel. Example : Keyboard and Mouse.

The hardware of a computer system can be classified as

Input Devices(I/P)

Processing Devices (CPU)

Output Devices(O/P)




ALU: It performs the Arithmetic and Logical Operations such as

+,-,*,/ (Arithmetic Operators)

&&, || ( Logical Operators)

CU: Every Operation such as storing , computing and retrieving the data should be governed by the control unit.

MU: The Memory unit is used for storing the data.

The Memory unit is classified into two types.

They are 1) Primary Memory

2) Secondary Memory

**Primary memory:** The following are the types of memoruies which are treated as primary

ROM: It represents Read Only Memory that stores data and instructions even when the computer is turned off. The Contents in the ROM can't be modified once if they are written . It is used to store the BIOS information.

RAM: It represents Random Access Memory that stores data and instructions when the computer is turned on. The contents in the RAM can be modified any no. of times by instructions. It is used to store the programs under execution.

**Cache memory:** It is used to store the data and instructions referred by processor.

**Secondary Memory:** The following are the different kinds of memories

Magnetic Storage: The Magnetic Storage devices store information that can be read, erased and rewritten a number of times.

Example: Floppy Disks, Hard Disks, Magnetic Tapes

**Optical Storage:** The optical storage devices that use laser beams to read and write stored data.

Example: CD(Compact Disk),DVD(Digital Versatile Disk)



**COMPUTER SOFTWARE**

Software of a computer system can be referred as anything which we can feel and see.

Example: Windows, icons

Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.

**System Software**

**System software** consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

**System support software** provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, **system development software**, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

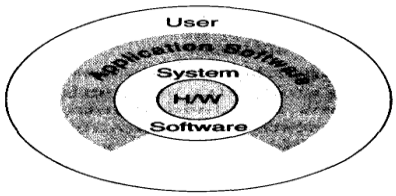
**Application software**

**Application software** is broken in to two classes: general-purpose software and application –specific software. **General purpose software** is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors, database management systems ,and computer aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

**Application –specific software** can be used only for its intended purpose.

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relationship between system and application software is shown below. In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.




Relationship Between System and Application Software

## COMPUTING ENVIRONMENTS

The word 'compute' is used to refer to the process of converting information to data. The advent of several new kinds of computers created a need to have different computing environments.

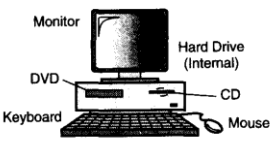
The following are the different kinds of computing environments available

1. Personal Computing Environment
2. Time Sharing Environment
3. Client/Server Environment
4. Distributed Computing Environment

### Personal Computing Environment

In 1971, Marcian E. Hoff, working for INTEL combined the basic elements of the central processing unit into the microprocessor. If we are using a personal computer then all the computer hardware components are tied together. This kind of computing is used to satisfy the needs of a single user, who uses the computer for the personal tasks.

Ex: Personal Computer

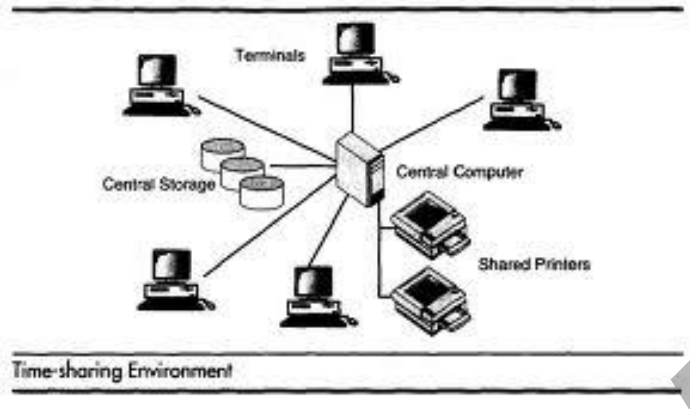



Personal Computing Environment

### Time-Sharing Environment

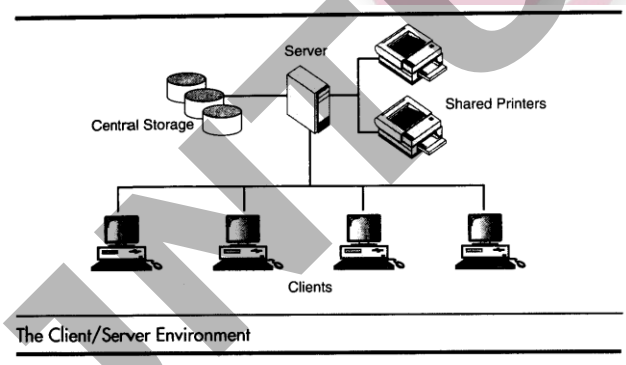
The concept of time sharing computing is to share the processing of the computer basing on the criteria time. In this environment all the computing must be done by the central computer.

The complete processing is done by the central computer. The computer which ask for processing are only dumb terminals.



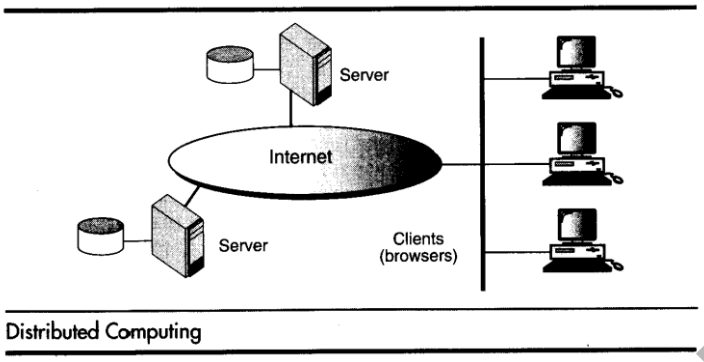

**Client/Server Environment**

A Client/Server Computing involves the processing between two machines. A client Machine is the one which requests processing. Server Machine is the one which offers the processing. Hence the client is Capable enough to do processing. A portion of processing is done by client and the core(important) processing is done by Server.




**Distributed Computing**

A distributed computing environment provides a seamless integration of computing functions between different servers and clients. A client not just a requestor for processing the information from the server. The client also has the capability to process information. All the machines Clients/Servers share the processing task.




Example: Ebay on Internet

**COMPUTER LANGUAGES**

To write a program (tells what to do) for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages. The following is the summary of computer languages

| | | |
|---|---|---|
| 1940's | -- | Machine Languages |
| 1950's | -- | Symbolic Languages |
| 1960's | -- | High Level Languages |

Machine Language

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made of streams of 0's and 1's. The instructions in machine language must be in streams of 0's and 1's. This is also referred as binary digits. These are so named as the machine can directly understood the programs

Advantages:

1) High speed execution
2) The computer can understood instructions immediately
3) No translation is needed.

Disadvantages:

1) Machine dependent
2) Programming is very difficult
3) Difficult to understand
4) Difficult to write bug free programs
5) Difficult to isolate an error

Example Additon of two numbers

| | | | |
|---|---|---|---|
| | 2 | | → 0 0 1 0 |
| + | 3 | | → 0 0 1 1 |
| | --- | | --- |
| | 5 | ← | 0 1 0 1 |
| | --- | | --- |

Symbolic Languages (or) Assembly Language

In the early 1950's Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language. These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions. These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language. A special program called an **Assembler** translates symbolic code into the machine language. Hence they are called as Assembly language.

Advantages:

1) Easy to understand and use
2) Easy to modify and isolate error
3) High efficiency
4) More control on hardware

Disadvantages:

1) Machine Dependent Language
2) Requires translator
3) Difficult to learn and write programs
4) Slow development time
5) Less efficient

Example:

| | |
|---|---|
| 2 | PUSH 2,A |
| 3 | PUSH 3,B |
| + | ADD A,B |
| 5 | PRINT C |

High-Level Languages

The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of high-level languages.

High-level languages are portable to many different computer allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.

| | |
|---|---|
| C | A systems implementation Language |
| C++ | C with object oriented enhancements |
| JAVA | Object oriented language for internet and general applications using basic C syntax |

Advantages:

1) Easy to write and understand
2) Easy to isolate an error
3) Machine independent language
4) Easy to maintain
5) Better readability
6) Low Development cost
7) Easier to document
8) Portable

Disadvantages:

1) Needs translator
2) Requires high execution time
3) Poor control on hardware
4) Less efficient

Example: C language

```
#include<stdio.h>
void main()
{
    int a,b,c;
    scanf("%d%d%",&a,&b);
```

```
    c=a+b;
    printf("%d",c);
}
```

Difference between Machine, Assembly, High Level Languages

| **Feature** | **Machine** | **Assembly** | **High Level** |
|---|---|---|---|
| Form | 0's and 1's | Mnemonic codes | Normal English |
| Machine Dependent | Dependent | Dependent | Independent |
| Translator | Not Needed | Needed(Assembler) | Needed(Compiler) |
| Execution Time | Less | Less | High |
| Languages | Only one | Different Manufactgurers | Different Languages |
| Nature | Difficult | Difficult | Easy |
| Memory Space | Less | Less | More |

**Language Translators**

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be excuted by the computer.

1) Compiler: It is a program which is used to convert the high level language programs into machine language
2) Assembler: It is a program which is used to convert the assembly level language programs into machine language
3) Interpreter: It is a program, it takes one statement of a high level language program, translates it into machine language instruction and then immediately executes the resulting machine language instruction and so on.

Comparison between a Compiler and Interpreter

| **COMPILER** | **INTERPRETER** |
|---|---|
| A Compiler is used to compile an entire program and an executable program is generated through the object program | An interpreter is used to translate each line of the program code immediately as it is entered |

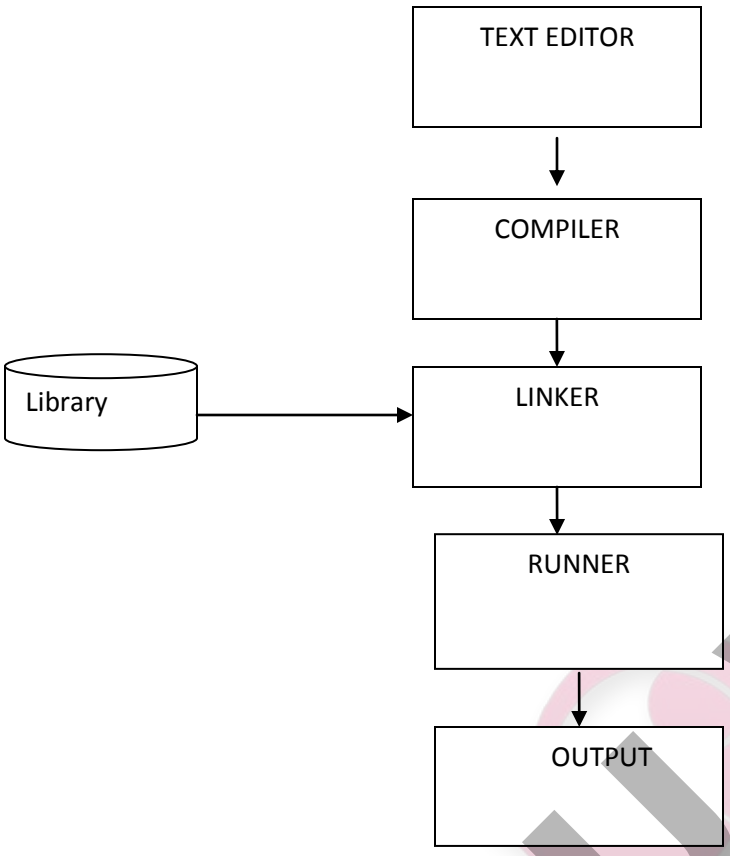
| | |
|---|---|
| The executable program is stored in a disk for future use or to run it in another computer | The executable program is generated in RAM and the interpreter is required for each run of the program |
| The compiled programs run faster | The Interpreted programs run slower |
| Most of the Languages use compiler | A very few languages use interpreters. |

**CREATING AND RUNNING PROGRAMS**

The procedure for turning a program written in C into machine Language. The process is presented in a straightforward, linear fashion but you shuld recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

The following are the four steps in this process

1) Writing and Editing the program
2) Compiling the program
3) Linking the program with the required modules
4) Executing the program




| Sl. No. | Phase | Name of Code | Tools | File Extension |
|---|---|---|---|---|
| 1 | TextEditor | Source Code | C Compilers Edit, Notepad Etc.., | .C |
| 2 | Compiler | Object Code | C Compiler | .OBJ |
| 3 | Linker | Executable Code | C Compiler | .EXE |
| 4 | Runner | Executable Code | C Compiler | .EXE |

Writing and Editing Programs

The software used to write programs is known as a text editor. A text editor helps us enter, change and store character data. Once we write the program in the text editor we save it using a filename stored with an extension of .C. This file is referred as source code file.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The Compiler is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language. This translation process is called *compilation.* The entire high level program is converted into the executable machine code file. The Compiler which executes C programs is called as C Compiler. Example Turbo C, Borland C, GC etc.,

The C Compiler is actually two separate programs:

The Preprocessor

The Translator

The Preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries. The result of preprocessing is called the translation unit.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in the machine language.

Linking Programs

The Linker assembles all functions, the program's functions and system's functions into one executable program.

Executing Programs

To execute a program we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the **loader.** It locates the executable program and reads it into memory. When everything is loaded the program takes control and it begin execution.

## ALGORITHM

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

The ordered set of instructions required to solve a problem is known as an *algorithm*.

The characteristics of a good algorithm are:

- Precision – the steps are precisely stated (defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

**Example**

Q. Write a algorithem to find out number is odd or even?

Ans.

```
step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
            print "number even"
         else
            print "number odd"
         endif
step 5 : stop
```

## FLOWCHART

Flowchart is a diagrammatic representation of an algorithm. Flowchart is very helpful in writing program and explaining program to others.

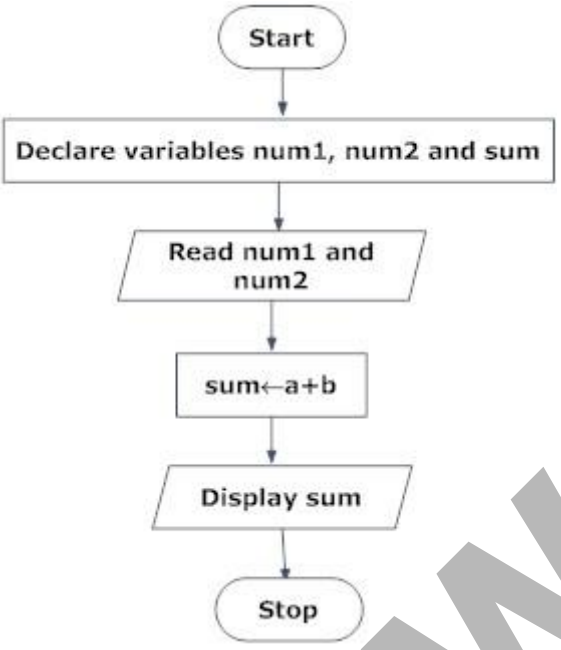
Symbols Used In Flowchart

Different symbols are used for different states in flowchart, For example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart

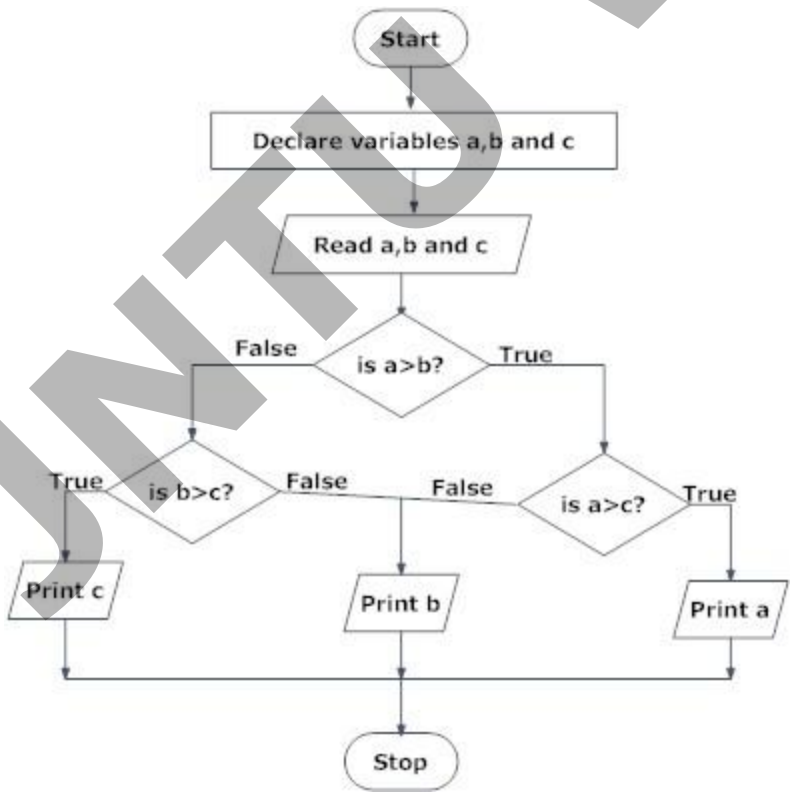
| Symbol | Purpose | Description |
|---|---|---|
|  | Flow line | Used to indicate the flow of logic by connecting symbols. |
|  | Terminal(Stop/Start) | Used to represent start and end of flowchart. |
|  | Input/Output | Used for input and output operation. |
|  | Processing | Used for airthmetic operations and data-manipulations. |
|  | Desicion | Used to represent the operation in which there are two alternatives, true and false. |
|  | On-page Connector | Used to join different flowline |
|  | Off-page Connector | Used to connect flowchart portion on different page. |
|  | Predefined Process/Function | Used to represent a group of statements performing one processing task. |

Examples of flowcharts in programming

Draw a flowchart to add two numbers entered by user.




Draw flowchart to find the largest among three different numbers entered by user.




**INTRODUCTION TO C LANGUAGE**

C is a general-purpose high level language that was originally developed by Dennis Ritchie for the Unix operating system. It was first implemented on the Digital Eqquipment Corporation PDP-11 computer in 1972.

The Unix operating system and virtually all Unix applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

***Facts about C***

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institue (ANSI).
- By 1973 UNIX OS almost totally written in C.
- Today C is the most widely used System Programming Language.
- Most of the state of the art software have been implemented using C

***Why to use C?***

C was initially used for system development work, in particular the programs that make-up the operating system. C was adoped as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers
- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C Program File

All the C programs are writen into text files with extension ".c" for example ***hello.c***. You can use "vi" editor to write your C program into a file.

## HISTORY TO C LANGUAGE

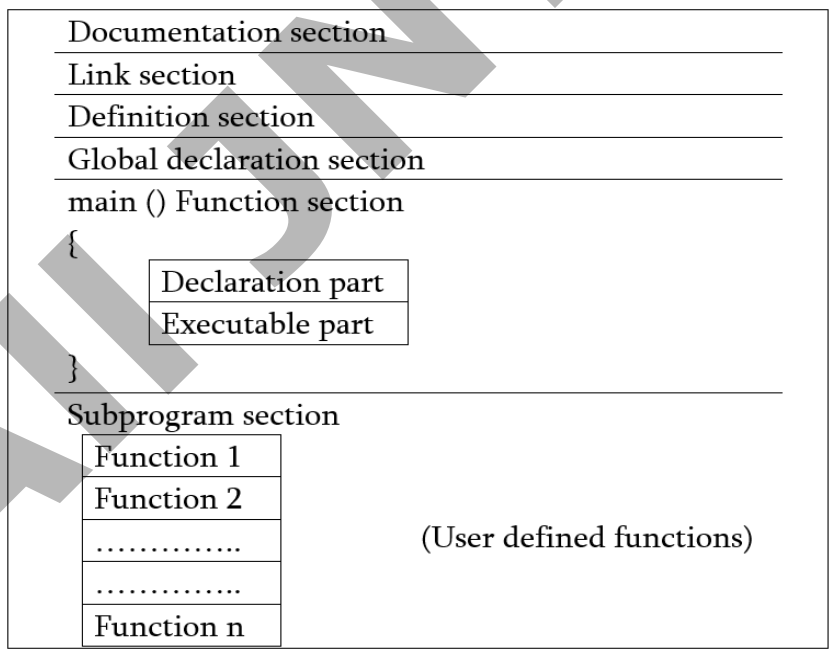
C is a general-purpose language which has been closely associated with the **UNIX** operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

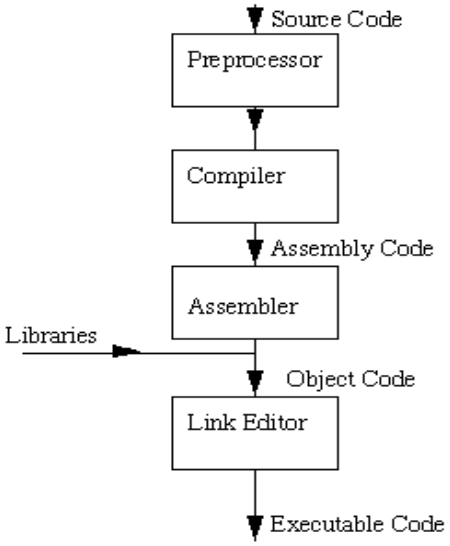
## BASIC STRUCTURE OF C PROGRAMMING



1. **Documentation section:** The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
2. **Link section:** The link section provides instructions to the compiler to link functions from the system library such as using the *#include directive*.
3. **Definition section:** The definition section defines all symbolic constants such using the *#define directive*.
4. **Global declaration section:** There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the *user-defined functions*.
5. **main () function section:** Every C program must have one main function section. This section contains two parts; declaration part and executable part
    1. **Declaration part:** The declaration part declares all the *variables* used in the executable part.
    2. **Executable part:** There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The *program execution* begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.
6. **Subprogram section:** If the program is a *multi-function program* then the subprogram section contains all the *user-defined functions* that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

**PROCESS OF COMPILING AND RUNNING C PROGRAM**

We will briefly highlight key features of the C Compilation model here.




**The C Compilation Model**

***The Preprocessor***

The Preprocessor accepts source code as input and is responsible for

- removing comments
- Interpreting special ***preprocessor directives*** denoted by #.

For example

- #include -- includes contents of a named file. Files usually called ***header*** files. ***e.g***
  - #include <math.h> -- standard library maths file.
  - #include <stdio.h> -- standard library I/O file
- #define -- defines a symbolic name or constant. Macro substitution.
  - #define MAX_ARRAY_SIZE 100

***C Compiler***

The C compiler translates source to assembly code. The source code is received from the preprocessor.

***Assembler***

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files.

***Link Editor***

If a source file references library functions or functions defined in other source files the ***link editor*** combines these functions (with main()) to create an executable file.

## C TOKENS

C tokens are the basic buildings blocks in C language which are constructed together to write a C program.

Each and every smallest individual unit in a C program is known as C tokens.

C tokens are of six types. They are

Keywords (eg: int, while),
Identifiers (eg: main, total),
Constants (eg: 10, 20),
Strings (eg: “total”, “hello”),
Special symbols (eg: (), {}),
Operators (eg: +, /,-,*)

## C KEYWORDS

**C keywords** are the words that convey a special meaning to the c compiler. The keywords cannot be used as variable names.

The list of C keywords is given below:

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

## C IDENTIFIERS

Identifiers are used as the general terminology for the names of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character.

There are certain rules that should be followed while naming c identifiers:

They must begin with a letter or underscore (_).

They must consist of only letters, digits, or underscore. No other special character is allowed.

It should not be a keyword.

It must not contain white space.

It should be up to 31 characters long as only first 31 characters are significant.

Some examples of c identifiers:

| **Name** | **Remark** |
|---|---|
| _A9 | Valid |
| Temp.var | Invalid as it contains special character other than the underscore |
| void | Invalid as it is a keyword |

## C CONSTANTS

A C constant refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

### Integer Constants

Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive.

There are three types of integer constants:

**Decimal Integer Constants**

Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.

Example of valid decimal integer constants

341, -341, 0, 8972

### Octal Integer Constants

Integer constants consisting of sequence of digits from the set 0 through 7 starting with 0 is said to be octal integer constants.

Example of valid octal integer constants

010, 0424, 0, 0540

**Hexadecimal Integer Constants**

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15.

Example of valid hexadecimal integer constants

0xD, 0X8d, 0X, 0xbD

It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

**Real Constants**

The numbers having fractional parts are called real or floating point constants. These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or – sign preceding it.

Example of valid real constants in fractional form or decimal notation

0.05, -0.905, 562.05, 0.015

**Representing a real constant in exponent form**

The general format in which a real number may be represented in exponential or scientific form is

**mantissa e exponent**

The mantissa must be either an integer or a real number expressed in decimal notation.

The letter e separating the mantissa and the exponent can also be written in uppercase i.e. E

And, the exponent must be an integer.

Examples of valid real constants in exponent form are:

252E85, 0.15E-10, -3e+8

**Character Constants**

A character constant contains one single character enclosed within single quotes.

Examples of valid character constants

'a' , 'Z', '5'

It should be noted that character constants have numerical values known as ASCII values, for example, the value of 'A' is 65 which is its ASCII value.

**Escape Characters/ Escape Sequences**

C allows us to have certain non graphic characters in character constants. Non graphic characters are those characters that cannot be typed directly from keyboard, for example, tabs, carriage return, etc.

These non graphic characters can be represented by using escape sequences represented by a backslash() followed by one or more characters.

**NOTE**: An escape sequence consumes only one byte of space as it represents a single character.

| **Escape Sequence** | **Description** |
|---|---|
| a | Audible alert(bell) |
| b | Backspace |
| f | Form feed |
| n | New line |
| r | Carriage return |
| t | Horizontal tab |
| v | Vertical tab |
| \ | Backslash |
| “ | Double quotation mark |
| ‘ | Single quotation mark |
| ? | Question mark |
| | Null |

**STRING CONSTANTS**

String constants are sequence of characters enclosed within double quotes. For example,

“hello”

“abc”

“hello911”

Every sting constant is automatically terminated with a special character **‘’** called the**null character** which represents the end of the string.

For example, “hello” will represent “hello” in the memory.

Thus, the size of the string is the total number of characters plus one for the null character.

**Special Symbols**

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

[] () {} , ; : * … = #

**Braces{}:** These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

**Parentheses():** These special symbols are used to indicate function calls and function parameters.

**Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

**VARIABLES**

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types −

| Type | Description |
|---|---|
| char | Typically a single octet(one byte). This is an integer type. |
| int | The most natural size of integer for the machine. |
| float | A single-precision floating point value. |
| double | A double-precision floating point value. |
| void | Represents the absence of type. |

C programming language also allows defining various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc.

### Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only; the compiler needs actual variable definition at the time of linking the program. A variable declaration is useful when multiple files are used.

## OPERATORS AND EXPRESSIONS

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

| S.no | Types of Operators | Description |
|---|---|---|
| 1 | **Arithmetic_operators** | These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus |
| 2 | **Assignment_operators** | These are used to assign the values for the variables in C programs. |
| 3 | **Relational operators** | These operators are used to compare the value of two variables. |
| 4 | **Logical operators** | These operators are used to perform logical |

| | | |
|---|---|---|
| | | operations on the given two variables. |
| 5 | **Bit wise operators** | These operators are used to perform bit operations on given two variables. |
| 6 | **Conditional (ternary) operators** | Conditional operators return one value if condition is true and returns another value is condition is false. |
| 7 | **Increment/decrement operators** | These operators are used to either increase or decrease the value of the variable by one. |
| 8 | **Special operators** | &, *, sizeof( ) and ternary operators. |

**ARITHMETIC OPERATORS IN C**

C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

| S.no | Arithmetic Operators | Operation | Example |
|---|---|---|---|
| 1 | + | Addition | A+B |
| 2 | – | Subtraction | A-B |
| 3 | * | multiplication | A*B |
| 4 | / | Division | A/B |
| 5 | % | Modulus | A%B |

**EXAMPLE PROGRAM FOR C ARITHMETIC OPERATORS**

In this example program, two values “40” and “20” are used to perform arithmetic operations such as addition, subtraction, multiplication, division, modulus and output is displayed for each operation.

```
#include <stdio.h>
 int main()
{
int a=40,b=20, add,sub,mul,div,mod;
add = a+b;
sub = a-b;
mul = a*b;
div = a/b;
mod = a%b;
printf("Addition of a, b is : %d\n", add);
printf("Subtraction of a, b is : %d\n", sub);
printf("Multiplication of a, b is : %d\n", mul);
printf("Division of a, b is : %d\n", div);
printf("Modulus of a, b is : %d\n", mod);
}
```

**OUTPUT:**

```
Addition of a, b is : 60
Subtraction of a, b is : 20
Multiplication of a, b is : 800
Division of a, b is : 2
Modulus of a, b is : 0
```

**ASSIGNMENT OPERATORS IN C**

In C programs, values for the variables are assigned using assignment operators.

For example, if the value "10" is to be assigned for the variable "sum", it can be assigned as "sum = 10;"

Other assignment operators in C language are given below.

| Operators | | Example | Explanation |
|---|---|---|---|
| Simple assignment operator | = | sum = 10 | 10 is assigned to variable sum |
| Compound assignment operators | += | sum += 10 | This is same as sum = sum + 10 |
| | -= | sum -= 10 | This is same as sum = sum – 10 |
| | *= | sum *= 10 | This is same as sum = sum * 10 |
| | /+ | sum /= 10 | This is same as sum = sum / 10 |
| | %= | sum %= 10 | This is same as sum = sum % 10 |
| | &= | sum&=10 | This is same as sum = sum & 10 |
| | ^= | sum ^= 10 | This is same as sum = sum ^ 10 |

**EXAMPLE PROGRAM FOR C ASSIGNMENT OPERATORS:**

In this program, values from 0 – 9 are summed up and total "45" is displayed as output. Assignment operators such as "=" and "+=" are used in this program to assign the values and to sum up the values.

```
# include <stdio.h>
 int main()
{
int Total=0,i;
for(i=0;i<10;i++)
{
Total+=i; // This is same as Total = Toatal+i
}
printf("Total = %d", Total);
}
```

**OUTPUT:**

```
Total = 45
```

**RELATIONAL OPERATORS IN C**

Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

| S.no | Operators | Example | Description |
|---|---|---|---|
| 1 | > | x > y | x is greater than y |
| 2 | < | x < y | x is less than y |
| 3 | >= | x >= y | x is greater than or equal to y |
| 4 | <= | x <= y | x is less than or equal to y |

| | | | |
|---|---|---|---|
| 5 | == | x == y | x is equal to y |
| 6 | != | x != y | x is not equal to y |

**EXAMPLE PROGRAM FOR RELATIONAL OPERATORS IN C**

In this program, relational operator (==) is used to compare 2 values whether they are equal are not.

If both values are equal, output is displayed as " values are equal". Else, output is displayed as "values are not equal".

Note: double equal sign (==) should be used to compare 2 values. We should not single equal sign (=).

```
#include <stdio.h>

 int main()

{

int m=40,n=20;

if (m == n)

{

printf("m and n are equal");

}

else

{

printf("m and n are not equal");

}

}
```

**OUTPUT:**

```
m and n are not equal
```

## LOGICAL OPERATORS IN C

These operators are used to perform logical operations on the given expressions.

There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

| S.no | Operators | Name | Example | Description |
|---|---|---|---|---|
| 1 | && | logical AND | (x>5)&&(y<5) | It returns true when both conditions are true |
| 2 | \|\| | logical OR | (x>=10)\|\|(y>=10) | It returns true when at-least one of the condition is true |
| 3 | ! | logical NOT | !((x>5)&&(y<5)) | It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false |



## EXAMPLE PROGRAM FOR LOGICAL OPERATORS IN C:

```
#include <stdio.h>

 int main()

{
```

```
int m=40,n=20;
int o=20,p=30;
if (m>n && m !=0)
{
printf("&& Operator : Both conditions are true\n");
}
if (o>p || p!=20)
{
printf("|| Operator : Only one condition is true\n");
}
if (!(m>n && m !=0))
{
printf("! Operator : Both conditions are true\n");
}
else
{
printf("! Operator : Both conditions are true. " \
"But, status is inverted as false\n");
}
}
```

**OUTPUT:**

```
&& Operator : Both conditions are true
|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is inverted as false
```

In this program, operators (&&, || and !) are used to perform logical operations on the given expressions.

**&& operator** – "if clause" becomes true only when both conditions (m>n and m! =0) is true. Else, it becomes false.

**|| Operator** – "if clause" becomes true when any one of the condition (o>p || p!=20) is true. It becomes false when none of the condition is true.

**! Operator** – It is used to reverses the state of the operand.

If the conditions (m>n && m!=0) is true, true (1) is returned. This value is inverted by "!" operator.

So, "! (m>n and m! =0)" returns false (0).

**BIT WISE OPERATORS IN C**

These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.

Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

**TRUTH TABLE FOR BIT WISE OPERATION BIT WISE OPERATORS**

| x | y | x\|y | x & y | x ^ y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

| Operator_symbol | Operator_name |
|---|---|
| & | Bitwise_AND |
| \| | Bitwise OR |
| ~ | Bitwise_NOT |
| ^ | XOR |
| << | Left Shift |
| >> | Right Shift |

Consider x=40 and y=80. Binary form of these values are given below.

x = 00101000

y= 01010000

All bit wise operations for x and y are given below.
x&y = 00000000 (binary) = 0 (decimal)
x|y = 01111000 (binary) = 120 (decimal)
~x = 11111111111111111111111111 11111111111111111111111111010111
.. ..= -41 (decimal)
x^y = 01111000 (binary) = 120 (decimal)
x << 1 = 01010000 (binary) = 80 (decimal)
x >> 1 = 00010100 (binary) = 20 (decimal)

**Note:**
**Bit wise NOT:** Value of 40 in binary is00000000000000000000000000000000000000000000000001010000000000. So, all 0's are converted into 1's in bit wise NOT operation.
**Bit wise left shift and right shift :** In left shift operation "x << 1 ", 1 means that the bits will be left shifted by one place. If we use it as "x << 2 ", then, it means that the bits will be left shifted by 2 places.

**EXAMPLE PROGRAM FOR BIT WISE OPERATORS IN C**
In this example program, bit wise operations are performed as shown above and output is displayed in decimal format.

```
#include <stdio.h>

int main()

{

int m = 40,n = 80,AND_opr,OR_opr,XOR_opr,NOT_opr ;

AND_opr = (m&n);

OR_opr = (m|n);

NOT_opr = (~m);

XOR_opr = (m^n);

printf("AND_opr value = %d\n",AND_opr );

printf("OR_opr value = %d\n",OR_opr );

printf("NOT_opr value = %d\n",NOT_opr );
```

```
printf("XOR_opr value = %d\n",XOR_opr );
printf("left_shift value = %d\n", m << 1);
printf("right_shift value = %d\n", m >> 1);
}
```

**OUTPUT:**

```
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
XOR_opr value = 120
left_shift value = 80
right_shift value = 20
```

**CONDITIONAL OR TERNARY OPERATORS IN C**

Conditional operators return one value if condition is true and returns another value is condition is false.

This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

Example : (A > 100 ? 0 : 1);

In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

**EXAMPLE PROGRAM FOR CONDITIONAL/TERNARY OPERATORS IN C**

```
#include <stdio.h>
int main()
{
int x=1, y ;
y = ( x ==1 ? 2 : 0 ) ;
printf("x value is %d\n", x);
printf("y value is %d", y);
```

```
}
```

**OUTPUT:**

```
x value is 1
y value is 2
```

**C – Increment/decrement Operators**

**PREVNEXT**

Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

Syntax:

Increment operator: ++var_name ;( or) var_name++;

Decrement operator: – -var_name; (or) var_name – -;

Example:

Increment operator : ++ i ; i ++ ;

Decrement operator : – – i ; i – – ;

**EXAMPLE PROGRAM FOR INCREMENT OPERATORS IN C**

In this program, value of "i" is incremented one by one from 1 up to 9 using "i++" operator and output is displayed as "1 2 3 4 5 6 7 8 9".

```
//Example for increment operators

#include <stdio.h>

int main()

{

    int i=1;

    while(i<10)

    {

      printf("%d ",i);

       i++;

    }

}
```

**OUTPUT:**

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

**EXAMPLE PROGRAM FOR DECREMENT OPERATORS IN C**

In this program, value of "I" is decremented one by one from 20 up to 11 using "i–" operator and output is displayed as "20 19 18 17 16 15 14 13 12 11".

```
//Example for decrement operators
#include <stdio.h>
int main()
{
   int i=20;
   while(i>10)
   {
       printf("%d ",i);
       i--;
   }
}
```

**OUTPUT:**

```
20 19 18 17 16 15 14 13 12 11
```

**DIFFERENCE BETWEEN PRE/POST INCREMENT & DECREMENT OPERATORS IN C**

Below table will explain the difference between pre/post increment and decrement operators in C.

| S.no | Operator type | Operator | Description |
|---|---|---|---|
| 1 | Pre increment | ++i | Value of i is |

| | | | |
|---|---|---|---|
| | | | incremented before assigning it to variable i. |
| 2 | Post–increment | i++ | Value of i is incremented after assigning it to variable i. |
| 3 | Pre decrement | — –i | Value of i is decremented before assigning it to variable i. |
| 4 | Post_decrement | i– — | Value of i is decremented after assigning it to variable i. |

**EXAMPLE PROGRAM FOR PRE – INCREMENT OPERATORS IN C**

```
//Example for increment operators

#include <stdio.h>

int main()

{

        int i=0;

        while(++i < 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

**OUTPUT:**

```
1 2 3 4
```

Step 1 : In above program, value of "i" is incremented from 0 to 1 using pre-increment operator.
Step 2 : This incremented value "1" is compared with 5 in while expression.
Step 3 : Then, this incremented value "1" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3 4".

**EXAMPLE PROGRAM FOR POST – INCREMENT OPERATORS IN C**

```
#include <stdio.h>

int main()

{

        int i=0;

        while(i++ < 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

**OUTPUT:**

```
1 2 3 4 5
```

Step 1 : In this program, value of i "0" is compared with 5 in while expression.
Step 2 : Then, value of "i" is incremented from 0 to 1 using post-increment operator.
Step 3 : Then, this incremented value "1" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3 4 5".

**EXAMPLE PROGRAM FOR PRE – DECREMENT OPERATORS IN C**

```
#include <stdio.h>

int main()

{

        int i=10;

        while(--i > 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

**OUTPUT:**

```
9 8 7 6
```

Step 1 : In above program, value of "i" is decremented from 10 to 9 using pre-decrement operator.
Step 2 : This decremented value "9" is compared with 5 in while expression.
Step 3 : Then, this decremented value "9" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7 6".

**EXAMPLE PROGRAM FOR POST – DECREMENT OPERATORS IN C:**

```
#include <stdio.h>

int main()

{

        int i=10;

        while(i-- > 5 )

        {
```

```
        printf("%d ",i);
    }
    return 0;
}
```

**OUTPUT:**

```
9 8 7 6 5
```

Step 1 : In this program, value of i "10" is compared with 5 in while expression.
Step 2 : Then, value of "i" is decremented from 10 to 9 using post-decrement operator.
Step 3 : Then, this decremented value "9" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7 6 5".

**SPECIAL OPERATORS IN C:**

Below are some of special operators that C language offers.

| S.no | Operators | Description |
|---|---|---|
| 1 | & | This is used to get the address of the variable.<br>Example : &a will give address of a. |
| 2 | * | This is used as pointer to a variable.<br>Example : * a where, * is pointer to the variable a. |
| 3 | Sizeof () | This gives the size of the variable.<br>Example : size of (char) will give us 1. |

**EXAMPLE PROGRAM FOR & AND * OPERATORS IN C**

In this program, "&" symbol is used to get the address of the variable and "*" symbol is used to get the value of the variable that the pointer is pointing to. Please refer **C – pointer** topic to know more about pointers.

```
#include <stdio.h>

int main()

{

int *ptr, q;

q = 50;

/* address of q is assigned to ptr */

ptr = &q;

/* display q's value using ptr variable */

printf("%d", *ptr);

return 0;

}
```

**OUTPUT:**

```
50
```

**EXAMPLE PROGRAM FOR SIZEOF() OPERATOR IN C**

sizeof() operator is used to find the memory space allocated for each C data types.

```
#include <stdio.h>

#include <limits.h>

 int main()

{

int a;

char b;
```

```
float c;
double d;
printf("Storage size for int data type:%d \n",sizeof(a));
printf("Storage size for char data type:%d \n",sizeof(b));
printf("Storage size for float data type:%d \n",sizeof(c));
printf("Storage size for double data type:%d\n",sizeof(d));
return 0;
}
```

**OUTPUT:**

```
Storage size for int data type:4
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8
```

**EXPRESSIONS**

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of mathematical expressions written in proper syntax of C are

Note: C does not have any operator for exponentiation.

**C OPERATOR PRECEDENCE AND ASSOCIATIVITY**

C operators in order of *precedence* (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

| Operator | Description | Associativity |
|---|---|---|
| ( )<br>[ ]<br>.<br>-><br>++ -- | Parentheses (function call) (see Note 1)<br>Brackets (array subscript)<br>Member selection via object name<br>Member selection via pointer<br>Postfix increment/decrement (see Note 2) | left-to-right |

| | | |
|---|---|---|
| ++ --<br>+ -<br>! ~<br>(*type*)<br>*<br>&<br>sizeof | Prefix increment/decrement<br>Unary plus/minus<br>Logical negation/bitwise complement<br>Cast (convert value to temporary value of *type*)<br>Dereference<br>Address (of operand)<br>Determine size in bytes on this implementation | right-to-left |
| * / % | Multiplication/division/modulus | left-to-right |
| + - | Addition/subtraction | left-to-right |
| << >> | Bitwise shift left, Bitwise shift right | left-to-right |
| < <=<br>> >= | Relational less than/less than or equal to<br>Relational greater than/greater than or equal to | left-to-right |
| == != | Relational is equal to/is not equal to | left-to-right |
| & | Bitwise AND | left-to-right |
| ^ | Bitwise exclusive OR | left-to-right |
| \| | Bitwise inclusive OR | left-to-right |
| && | Logical AND | left-to-right |
| \|\| | Logical OR | left-to-right |
| ? : | Ternary conditional | right-to-left |
| =<br>+= -=<br>*= /=<br>%= &=<br>^= \|=<br><<= >>= | Assignment<br>Addition/subtraction assignment<br>Multiplication/division assignment<br>Modulus/bitwise AND assignment<br>Bitwise exclusive/inclusive OR assignment<br>Bitwise shift left/right assignment | right-to-left |
| , | Comma (separate expressions) | left-to-right |

**Note 1:**

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

**Note 2:**

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.*, **z++** evaluates to **z**) and **z** only incremented after all else is done.

**EVALUATION OF EXPRESSION**

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

**High priority:** * / %

**Low priority:** + -

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered.

Suppose, we have an arithmetic expression as:

x = 9 – 12 / 3 + 3 *2 - 1

This expression is evaluated in two left to right passes as:

**First Pass**

Step 1: x = 9-4 + 3 * 2 – 1

Step 2: x = 9 – 4 + 6 – 1

**Second Pass**

Step 1: x = 5 + 6 – 1

Step 2: x = 11 – 1

Step 3: x = 10

But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

x = 9 – 12 / (3 + 3) * (2 – 1)

When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passes as:

**First Pass**

Step 1: x = 9 – 12 / 6 * (2 – 1)

Step 2: x= 9 – 12 / 6 * 1

**Second Pass**

Step 1: x= 9 – 2 * 1

Step 2: x = 9 – 2

**Third Pass**

Step 3: x= 7

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

For example, we have an expression as:

x = 9 – ((12 / 3) + 3 * 2) – 1

The expression is now evaluated as:

**First Pass:**

Step 1: x = 9 – (4 + 3 * 2) – 1

Step 2: x= 9 – (4 + 6) – 1

Step 3: x= 9 – 10 -1

**Second Pass**

Step 1: x= - 1 – 1

Step 2: x = -2

Note: The number of evaluation steps is equal to the number of operators in the arithmetic expression.

## TYPE CONVERSION IN EXPRESSIONS

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion.

Type conversion in c can be classified into the following two types:

**Implicit Type Conversion**

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as **implicit type conversion** or **type promotion**.

The compiler converts all operands into the data type of the largest operand.

The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

If either of the operand is of type long double, then others will be converted to long double and result will be long double.

Else, if either of the operand is double, then others are converted to double.

Else, if either of the operand is float, then others are converted to float.
Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
Else, if one of the operand is long int, and the other is unsigned int, then
if a long int can represent all values of an unsigned int, the unsigned int is converted to long int.
otherwise, both operands are converted to unsigned long int.
Else, if either operand is long int then other will be converted to long int.
Else, if either operand is unsigned int then others will be converted to unsigned int.
It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.
Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

**Explicit Type Conversion**

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.
The explicit type conversion is also known as **type casting**.
Type casting in c is done in the following form:

**(data_type)expression;**

where, *data_type* is any valid c data type, and *expression* may be constant, variable or expression.
For example, x=(int)a+b*d;
The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:
All integer types to be converted to float.
All float types to be converted to double.
All character types to be converted to integer.

**FORMATTED INPUT AND OUTPUT**

The C Programming Language is also called the Mother of languages. The C language was developed by Dennis Ritchie between 1969 and 1973 and is a second and third generation of languages. The C language provides both low and high level features it provides both the power of low-level languages and the flexibility and simplicity of high-level languages.

C provides standard functions scanf() and printf(), for performing formatted input and output. These functions accept, as parameters, a format specification string and a list of variables.

The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.

Now to discuss formatted output in functions.

**Formatted Output**

The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.

**Syntax:**

printf (format, data1, data2,……..);

In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.

**Example:**

printf ("%c", data1);

The character specified after % is called a conversion character because it allows one data type to be converted to another type and printed.

See the following table conversion character and their meanings.

| **Conversion Character** | **Meaning** |
|---|---|
| d | The data is converted to decimal (integer) |
| c | The data is taken as a character. |
| s | The data is a string and character from the string , are printed until a NULL, character is reached. |

| f | The data is output as float or double with a default Precision 6. |
|---|---|
| **Symbols** | **Meaning** |
| \n | For new line (linefeed return) |
| \t | For tab space (equivalent of 8 spaces) |

**Example**

printf (“%c\n”,data1);

The format specification string may also have text.

**Example**

printf (“Character is:”%c\n”, data1);

The text "Character is:" is printed out along with the value of data1.

**Example with program**

```
#include<stdio.h>
#include<conio.h>
Main()
{
Char alphabh="A";
int number1= 55;
float number2=22.34;
printf(“char= %c\n”,alphabh);
printf(“int= %d\n”,number1);
printf(“float= %f\n”,number2);
getch();
clrscr();
retrun 0;
}
```

Output Here…

**char** =A

**int**= 55

flaot=22.340000

**What is the output of the statement?**

printf("Integer is: %d; Alphabet is:%c\n",number1, alpha);

Where number1 contains 44 and alpha contains "Krishna Singh".

Give the answer below.

Between the character % and the conversion character, there may be:

- A minus sign: Denoting left adjustment of the data.
- A digit: Specifying the minimum width in which the data is to be output, if the data has a larger number of characters then the specified width occupied by the output is larger. If the data consists of fewer characters then the specified width, it is padded to the right or to the left (if minus sign is not specified) with blanks. If the digit is prefixed with a zero, the padding is done with zeros instead of blanks.
- A period: Separating the width from the next digit.
- A digit following the period: specifying the precision (number of decimal places for numeric data) or the maximum number of characters to be output.
- Letter 1: To indicate that the data item is a long integer and not an int.

| **Format specification string** | **Data** | **Output** |
|---|---|---|
| \|%2d\| | 9 | \|9\| |
| \|%2d\| | 123 | \|123\| |
| \|%03d\| | 9 | \|009\| |
| \|%-2d\| | 7 | \|7\| |
| \|%5.3d\| | 2 | \|002\| |
| \|%3.1d\| | 15 | \|15\| |
| \|%3.5d\| | 15 | \|0015\| |
| \|%5s\| | "Output sting" | \|Output string\| |
| \|%15s\| | "Output sting" | \|Output string\| |
| \|%-15s\| | "Output sting" | \|Output string\| |
| \|%15.5s\| | "Output sting" | \|Output string\| |
| \|%.5s\| | "Output sting" | \|Output\| |

| | | |
|---|---|---|
| \|%15.5s\| | "Output sting" | \|Output\| |
| \|%f\| | 87.65 | \|87.650000\| |
| \|%.4.1s\| | 87.65 | \|87.71\| |

Example based on the conversion character:

```
#include<stdio.h>
#include<conio.h>
main()
{
Int num=65;
printf("Value of num is : %d\n:, num);
printf("Character equivalent of %d is %c\n", num , num);
getch();
clrscr();
rerurn o;
}
```

Output Here…

char =A

int= 55

flaot=22.340000

**Formatted Input**

The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().

**Syntax**

scanf (format, num1, num2,……);

The function scnaf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,….).

For Example:

scanf(" %c %d",&Name, &Roll No);

**Note**: the data names are listed as &Name and &Roll No instead of Name and Roll No

respectively. This is how data names are specified in a scnaf() function. In case of string type data names, the data name is not preceded by the character &.

**Example with program**

Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.

Solve here:

```
#include<stdio.h>
#include<conio.h>
main()
{
Int e_num;
Float e_wt;
printf (“Enter the Element No. and Weight of a Proton\n”);
scanf (“%d %f”,&e_num, &e_wt);
printf (“The Element No.is:”,e_num);
printf (“The Weight of a Proton is: %f\n”, e_wt);
getch();
return 0;
}
```

## UNIT-II

## CONTROL STRUCTURES, ARRAYS AND STRINGS

### DECISION STATEMENTS

**If statement:**

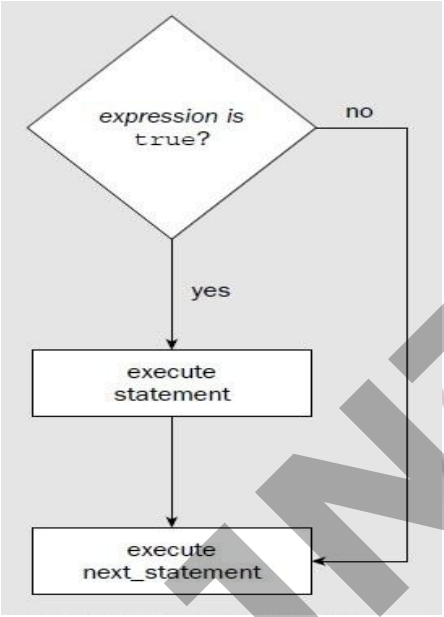
**Syntax :**

**if**(expression)

  statement1;

**Explanation :**

- Expression is Boolean Expression
- It may have true or false value




**Meaning of If Statement :**

- It Checks whether the given Expression is Boolean or not !!
- If Expression is True Then it executes the statement otherwise jumps to next_instruction

**Sample Program Code :**

```
void main()
{
int a=5,b=6,c;
  c = a + b ;
```

```
 if (c==11)
     printf("Execute me 1");

 printf("Execute me 2");
}
```

**Output :**

Execute me 1

If Statement :

```
if(conditional)
{
   Statement No 1
   Statement No 2
   Statement No 3
   .
   .
   .
   Statement No N
}
```

Note :

More than One Conditions can be Written inside If statement.

1. Opening and Closing Braces are required only when "Code" after if statement occupies multiple lines.

```
if(conditional)
   Statement No 1
Statement No 2
Statement No 3
```

In the above example only Statement 1 is a part of if Statement.

1. Code will be executed if condition statement is True.
2. Non-Zero Number Inside if means **"TRUE Condition"**

```
if(100)
   printf("True Condition");
```

**if-else Statement :**

We can use if-else statement in c programming so that we can check any condition and depending on the outcome of the condition we can follow appropriate path. We have true path as well as false path.

Syntax :

```
if(expression)
  {
  statement1;
  statement2;
  }
 else
  {
  statement1;
  statement2;
  }

next_statement;
```

Explanation :

If expression is True then Statement1 and Statement2 are executed

Otherwise Statement3 and Statement4 are executed.

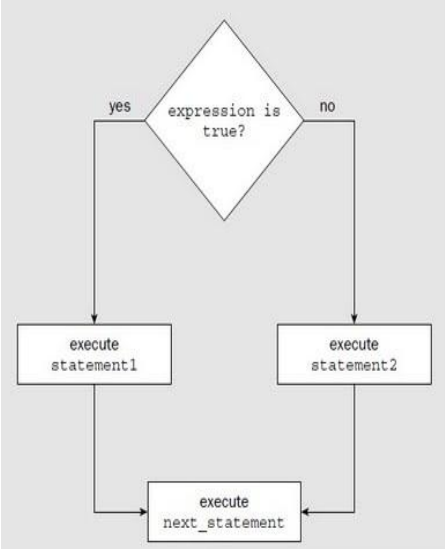
Sample Program on if-else Statement :

```
void main()
{
int marks=50;
 if(marks>=40)
  {
  printf("Student is Pass");
  }
 else
  {
  printf("Student is Fail");
  }
}
```

**Output :**

Student is Pass

Flowchart : If Else Statement




Consider Example 1 with Explanation:

Consider Following Example –

```
int num = 20;

if(num == 20)
   {
   printf("True Block");
   }
else
   {
   printf("False Block");
   }
```

If part Executed if Condition Statement is True.

```
if(num == 20)
   {
   printf("True Block");
   }
```

True Block will be executed if condition is True.

Else Part executed if Condition Statement is False.

```
else
   {
   printf("False Block");
   }
```

Consider Example 2 with Explanation :

More than One Conditions can be Written inside If statement.

```
int num1 = 20;
int num2 = 40;

if(num1 == 20 && num2 == 40)
   {
   printf("True Block");
   }
```

Opening and Closing Braces are required only when “Code” after if statement occupies multiple lines. Code will be executed if condition statement is True. Non-Zero Number Inside if means **“TRUE Condition”**

If-Else Statement :

```
if(conditional)
{
//True code
}
else
{
//False code
}
```

Note :

Consider Following Example –

```
int num = 20;

if(num == 20)
   {
   printf("True Block");
   }
else
   {
   printf("False Block");
   }
```

If part Executed if Condition Statement is True.

```
if(num == 20)
   {
   printf("True Block");
   }
```

True Block will be executed if condition is True.

Else Part executed if Condition Statement is False.

```
else
   {
   printf("False Block");
   }
```

More than One Conditions can be Written inside If statement.

```
int num1 = 20;
int num2 = 40;

if(num1 == 20 && num2 == 40)
   {
   printf("True Block");
   }
```

Opening and Closing Braces are required only when "Code" after if statement occupies multiple lines.

Code will be executed if condition statement is True.

Non-Zero Number Inside if means **"TRUE Condition"**

**Switch statement**

Why we should use Switch Case?

- One of the classic problem encountered in **nested if-else / else-if ladder**is called **problem of Confusion**.
- It occurs when no matching else is available for if .
- As the number of **alternatives increases the Complexity of program increases drastically**.
- To overcome this , C Provide a **multi-way decision statement** called '**Switch Statement**'

See how difficult is this scenario?

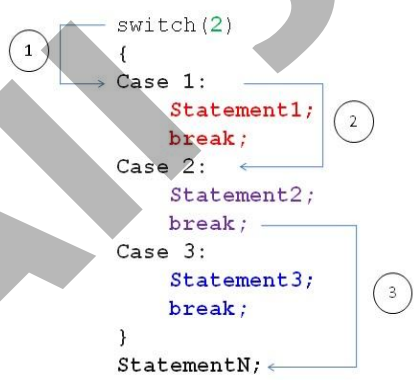
```
if(Condition 1)
  Statement 1
else
  {
  Statement 2
  if(condition 2)
    {
    if(condition 3)
```

```
      statement 3
    else
      if(condition 4)
        {
        statement 4
        }
   }
 else
  {
  statement 5
  }
 }
```

First Look of Switch Case

```
switch(expression)
{
case value1 :
   body1
   break;

case value2 :
   body2
   break;

case value3 :
   body3
   break;

default :
   default-body
   break;
}
next-statement;
```

Flow Diagram :




*Steps are Shown in Circles.

How it works?

- Switch case checks the value of expression/variable against the list of case values and when the match is found , **the block of statement associated with that case is executed**
- Expression should be Integer **Expression / Character**
- **Break statement takes** control out of the case.
- Break Statement is **Optional**.

```
#include<stdio.h>
void main()
{
int roll = 3 ;
switch ( roll )
{
case 1:
printf ( " I am Pankaj ");
break;
case 2:
printf ( " I am Nikhil ");
break;
case 3:
printf ( " I am John ");
break;
default :
printf ( "No student found");
break;
}
}
```

As explained earlier –

3 is assigned to integer variable '**roll**'

On line 5 switch case decides – "**We have to execute block of code specified in 3rd case**".

Switch Case executes code from top to bottom.

It will now enter into first Case [i.e case 1:]

It will validate **Case number** with variable **Roll**.

If no match found then it will jump to Next Case..

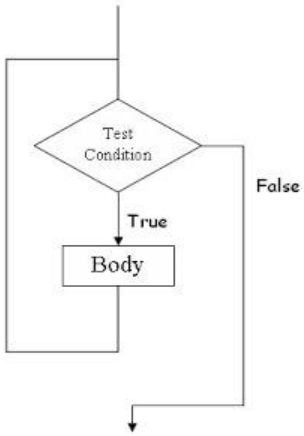
When it finds matching case it will execute block of code specified in that case.

**LOOP CONTROL STATEMENTS**

**While statement:**

While Loop Syntax:

```
initialization;
while(condition)
{
----------
----------
---------
incrementation;
}
```




Note :

For Single Line of Code – Opening and Closing braces are not needed.

while(1) is used for Infinite Loop

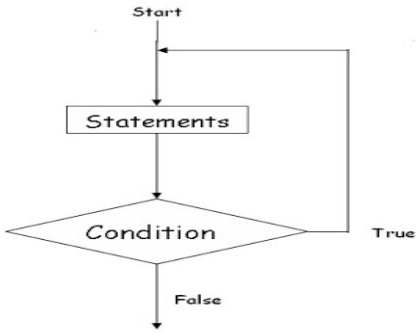
Initialization , Incrementation and Condition steps are on different Line.

While Loop is also Entry Controlled Loop.[i.e conditions are checked if found true then and then only code is executed ]

**Do while:**

Do-While Loop Syntax :

```
initialization;
do
{
-------------
-------------
-------------
-------------
incrementation;
}while(condition);
```




Note :

It is Exit Controlled Loop.

Initialization , Incrementation and Condition steps are on different Line.

It is also called Bottom Tested [i.e Condition is tested at bottom and Body has to execute at least once ]

**For statement:**

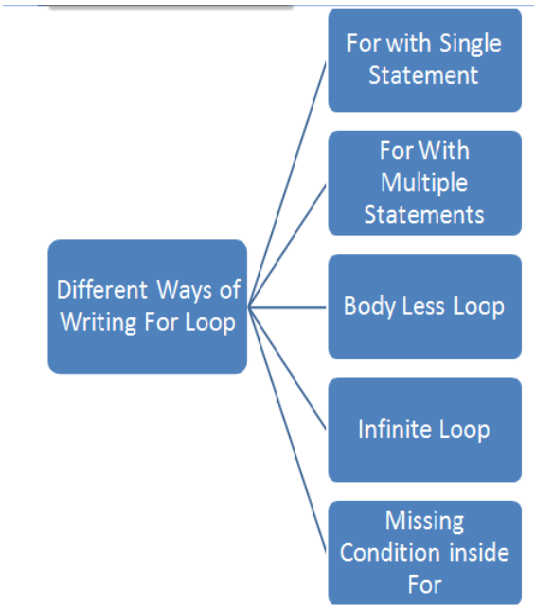
We have already seen the basics of Looping Statement in C. C Language provides us different kind of looping statements such as For loop, while loop and do-while loop. In this chapter we will be learning different flavors of for loop statement.

Different Ways of Using For Loop in C Programming

In order to do certain actions multiple times, we use loop control statements.

For loop can be implemented in different verities of using for loop –

- Single Statement inside For Loop
- Multiple Statements inside For Loop
- No Statement inside For Loop
- Semicolon at the end of For Loop
- Multiple Initialization Statement inside For
- Missing Initialization in For Loop
- Missing Increment/Decrement Statement
- Infinite For Loop
- Condition with no Conditional Operator.




*Different Ways of Writing For Loop in C Programming Language*

Way 1 : Single Statement inside For Loop

```
for(i=0;i<5;i++)
  printf("Hello");
```

Above code snippet will print Hello word 5 times.

We have single statement inside for loop body.

No need to wrap printf inside opening and closing curly block.

Curly Block is Optional.

Way 2 : Multiple Statements inside For Loop

```
for(i=0;i<5;i++)
   {
   printf("Statement 1");
   printf("Statement 2");
   printf("Statement 3");

   if(condition)
      {
      --------
      --------
      }
   }
```

If we have block of code that is to be executed multiple times then we can use curly braces to wrap multiple statement in for loop.

Way 3 : No Statement inside For Loop

```
for(i=0;i<5;i++)
  {

  }
```

this is bodyless for loop. It is used to increment value of "i".This verity of for loop is not used generally.

At the end of above for loop value of i will be 5.

Way 4 : Semicolon at the end of For Loop

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

Generally beginners thought that , we will get compile error if we write semicolon at the end of for loop.

This is perfectly legal statement in C Programming.

This statement is similar to bodyless for loop. (Way 3)

Way 5 : Multiple Initialization Statement inside For

```
for(i=0,j=0;i<5;i++)
  {
  statement1;
  statement2;
  statement3;
  }
```

Multiple initialization statements must be seperated by Comma in for loop.

Way 6 : Missing Increment/Decrement Statement

```
for(i=0;i<5;)
  {
  statement1;
  statement2;
  statement3;
  i++;
  }
```

however we have to explicitly alter the value i in the loop body.

Way 7 : Missing Initialization in For Loop

```
i = 0;
```

```
for(;i<5;i++)
  {
  statement1;
  statement2;
  statement3;
  }
```

we have to set value of 'i' before entering in the loop otherwise it will take garbage value of 'i'.

Way 8 : Infinite For Loop

```
i = 0;
for(;;)
  {
  statement1;
  statement2;
  statement3;

  if(breaking condition)
    break;

  i++;
  }
```

Infinite for loop must have breaking condition in order to break for loop. otherwise it will cause overflow of stack.

Summary of Different Ways of Implementing For Loop

| **Form** | **Comment** |
|---|---|
| for ( i=0 ; i < 10 ; i++ )<br>Statement1; | **Single** Statement |
| for ( i=0 ;i <10; i++)<br>{<br>Statement1;<br>Statement2;<br>Statement3;<br>} | **Multiple** Statements within for |
| for ( i=0 ; i < 10;i++) ; | For Loop with no Body (**Carefully Look at the Semicolon**) |
| for | **Multiple initialization** & Multiple |

| | |
|---|---|
| (i=0,j=0;i<100;i++,j++)<br>Statement1; | **Update** Statements Separated by Comma |
| for ( ; i<10 ; i++) | **Initialization not used** |
| for ( ; i<10 ; ) | **Initialization & Update not used** |
| for ( ; ; ) | **Infinite** Loop, Never Terminates |

**JUMP STATEMENTS:**

**Break statement**
Break Statement Simply Terminate Loop and takes control out of the loop.

**Break in For Loop :**

```
for(initialization ; condition ; incrementation)
{
Statement1;
Statement2;
break;
}
```

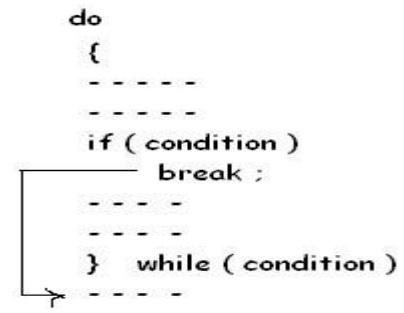
**Break in While Loop :**

```
initialization ;
while(condition)
{
Statement1;
Statement2;
incrementation
break;
}
```

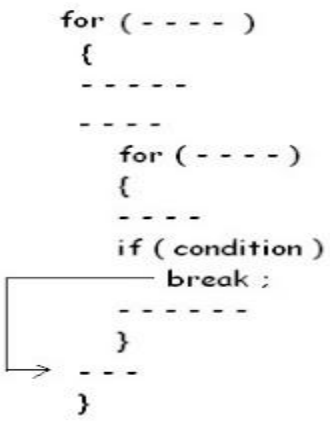
**Break Statement in Do-While :**

```
initialization ;
do
{
Statement1;
Statement2;
incrementation
break;
}while(condition);
```

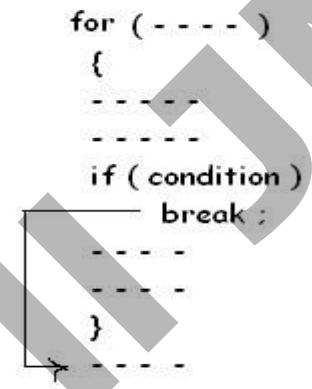
**Way 1 : Do-While Loop**



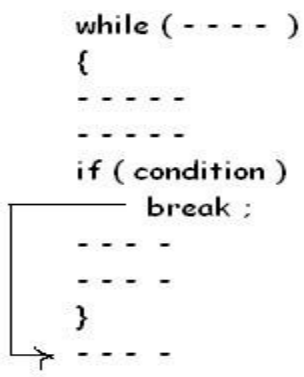

**Way 2 : Nested for**




**Way 3 : For Loop**




**Way 4 : While Loop**




**Continue statement:**

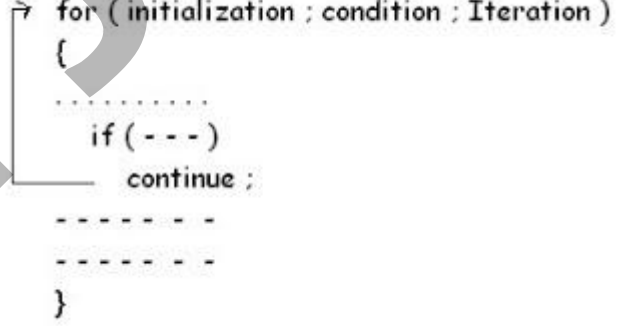
```
loop
{
  continue;
  //code
}
```

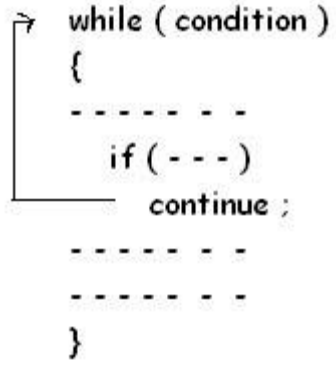
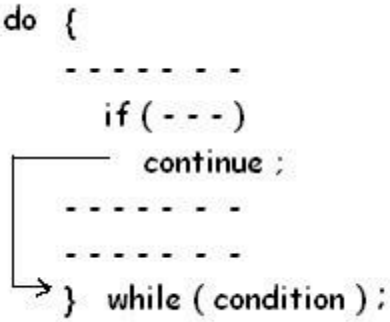
Note :

It is used for skipping part of Loop.

Continue causes the remaining code inside a loop block to be skipped and causes execution to jump to the top of the loop block



| Loop | Use of Continue !! |
|---|---|
| for |  |

| | |
|---|---|
| while |  |
| do-while |  |

**Goto statement:**

goto label;

-------

-------

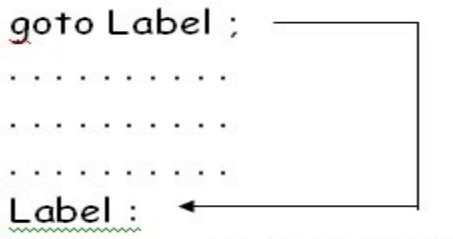
label :

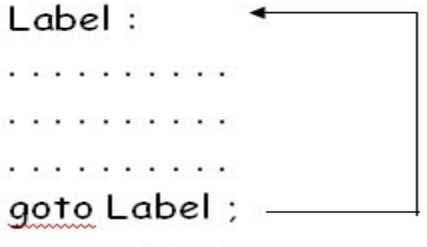
Whenever goto keyword encountered then it causes the program to continue on the line , so long as it is in the scope .

Types of Goto

Forward

Backward




**ARRAYS:**

What is an array?

An array is a collection of similar datatype that are used to allocate memory in a sequential manner.

Syntax : <data type> <array name>[<size of an array>]

Subscript or indexing: A subscript is property of an array that distinguishes all its stored elements because all the elements in an array having the same name (i.e. the array name). so to distinguish these, we use subscripting or indexing option.

e.g. int ar[20];

First element will be: int ar[0];

Second element will be: int ar[1];

Third element will be: int ar[2];

Fourth element will be: int ar[3];

Fifth element will be: int ar[4];

Sixth element will be: int ar[5];

So on……

Last element will be: int ar[19];

- NOTE: An array always starts from 0 indexing.
- Example: int ar[20];

This above array will store 20 integer type values from 0 to 19.

Advantage of an array:

- Multiple elements are stored under a single unit.
- Searching is fast because all the elements are stored in a sequence.

Types of Array

1. Static Array
2. Dynamic Array.

Static Array

An array with fixed size is said to be a static array.

Types of static array:

1. One Dimensional Array
2. Two Dimensional Array.
3. Multi Dimensional Array.

1. One Dimensional Array

An Array of elements is called 1 dimensional, which stores data in column or row form.

Example: int ar[5];

This above array is called one dimensional array because it will store all the elements in column or in row form

2. Two Dimensional Array.

An array of an array is said to be 2 dimensional array , which stores data in column androw form

Example: int ar[4][5];

This above array is called two dimensional array because it will store all the elements in column and in row form

NOTE: In above example of two dimensional array, we have 4 rows and 5 columns.

NOTE: In above example of two dimensional array, we have total of 20 elements.

3. Multi Dimensional Array.

This array does not exist in c and c++.

Dynamic Array.

This type of array also does not exist in c and c++.

Example: Program based upon array:

WAP to store marks in 5 subjects for a student. Display marks in 2nd and 5thsubject.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int ar[5];
int i;
for(i=0;i<5;i++)
{
printf(" \n Enter marks in ",i, "subject");
scanf("%d",&ar[i]);
}
printf("Marks in 2nd subject is: ",ar[1]);
printf("Marks in 5th subject is: ",ar[4]);

}
```

**STRINGS**

What is String?

- A string is a collection of characters.
- A string is also called as an array of characters.
- A String must access by %s access specifier in c and c++.
- A string is always terminated with \0 (Null) character.
- Example of string: "Gaurav"
- A string always recognized in double quotes.
- A string also consider space as a character.

- Example: ” Gaurav Arora”
- The above string contains 12 characters.
- Example: Char ar[20]
- The above example will store 19 character with I null character.

Example: Program based upon String.

WAP to accept a complete string (first name and last name) and display hello message in the output.

```
# include<stdio.h>
#include<conio.h>
#include<string.h>
void main ()
{
char str1[20];
char str2[20];
printf(“Enter First Name”);
scanf(“%s”,&str1);
printf(“Enter last Name”);
scanf(“%s”,&str2);
puts(str1);
puts(str2);
}
```

String Functions in C:

Our c language provides us lot of string functions for manipulating the string.

All the string functions are available in string.h header file.

These String functions are:

1. strlen().
2. strupr().
3. strlwr().
4. strcmp().

5. strcat().
6. strapy().
7. strrev().

1. strlen().

This string function is basically used for the purpose of computing the ength of string.

Example: char str=”Gaurav Arora”;

```
int length= strlen(str);
printf(“The length of the string is =”,str);
```

2. strupr().

This string function is basically used for the purpose of converting the case sensitiveness of the string i.e. it converts string case sensitiveness into uppercase.

Example: char str = “gaurav”

```
strupr(str);
printf(“The uppercase of the string is : %s”,str);
```

3. strlwr ().

This string function is basically used for the purpose of converting the case sensitiveness of the string i.e it converts string case sensitiveness into lowercase.

Example: char str = “gaurav”

```
strlwr(str);
printf(“The Lowercase of the string is :%s ”,str);
```

4. strcmp ().

This string function is basically used for the purpose of comparing two string.

This string function compares two strings character by characters.

Thus it gives result in three cases:

Case 1: if first string > than second string then, result will be true.

Case 2: if first string < than second string then, result will be false.

Case 3: if first string = = to second string then, result will be zero.

Example:

```
char str1= “Gaurav”;
char str2= “Arora”;
char str3=strcmp(str1,str2);
printf(“%s”,str3);
```

5. strcat().

This string function is used for the purpose of concatenating two strings ie.(merging two or more strings)

Example:

```
char str1 = “Gaurav”;
char str2 = “Arora”;
char str3[30];
str3=strcat(str1,str2);
printf(“%s”,str3);
```

6. strcpy()

This string function is basically used for the purpose of copying one string into another string.

```
char str1= “Gaurav”;
char str2[20];
str2 = strcpy(str2,str1);
printf(“%s”,str2);
```

6. strrev()

This string function is basically used for the purpose of reversing the string.

```
char str1= “Gaurav”;
char str2[20];
```

```
str2= strrev(str2,str1);
printf("%s",str2);
```

Example: Program based upon string functions.

WAP to accept a string and perform various operations:

1. To convert string into upper case.
2. To reverse the string .
3. To copy string into another string.
4. To compute length depending upon user choice.

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

void main()
{
char str[20];
char str1[20];
int opt,len;
printf("\n  MAIN MENU");
printf("\n 1. Convert string into upper case");
printf("\n 2. Reverse the string");
printf("\n 3. Copy one string into another string");
printf("\n  4.Compute length of string ");
printf("Enter string ");
scanf("%s", &str);
printf("Enter your choice");
scanf("%d",&opt);
switch(opt)
{
case 1:                         strupr(str);
```

```
                printf("The string in uppercase is :%s ",str);
                break;

case 2:         strrev(str);
                printf("The reverse of string is : %s",str);
                break;
case 3:         strcpy(str1,str);
                printf("New copied string is : %s",str1);
                break;
case 4:         len=strlen(str);
                printf("The length of the string is :  %s",len);
                break;
default:        printf("Ypu have entered a wrong choice.");
}
```

## UNIT-III

## FUNCTIONS AND POINTERS

### FUNCTIONS

A function is itself a block of code which can solve simple or complex task/calculations.

A function performs calculations on the data provided to it is called "parameter" or "argument".

A function always returns single value result.

Types of function:

1. Built in functions(Library functions)

a.) Inputting Functions.

b.) Outputting functions.

2. User defined functions.

a.) fact();

b.) sum();

Parts of a function:

1. Function declaration/Prototype/Syntax.
2. Function Calling.
3. Function Definition.

1.)Function Declaration:

Syntax: <return type > <function name>(<type of argument>)

The declaration of function name, its argument and return type is called function declaration.

2.) Function Calling:

The process of calling a function for processing is called function calling.

Syntax: <var_name>=<function_name>(<list of arguments>).

3.) Function defination:

The process of writing a code for performing any specific task is called function defination.

Syntax:

<return type><function name>(<type of arguments>)

{

<statement-1>

<statement-2>

return(<vlaue>)

}

Example: program based upon function:

WAP to compute cube of a no. using function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int c,n;
int cube(int);
printf("Enter a no.");
scanf("%d",&n);
c=cube(n);
printf("cube of a no. is=%d",c);
}
int cube(int n)
{
c=n*n*n;
return(c);
}
```

WAP to compute factorial of a no. using function:

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int n,f=1;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 for(int i=n;i>=n;i--)
{
 f=f*i;
}
return(f);
}
```

Recursion

Firstly, what is nested function?

When a function invokes another function then it is called nested function.

But,

When a function invokes itself then it is called recursion.

NOTE: In recursion, we must include a terminating condition so that it won't execute to infinite time.

Example: program based upon recursion:

WAP to compute factorial of a no. using Recursion:

```
#include<stdio.h>
#include<conio.h>
void main()
```

```
{
int n,f;
int fact(int)
printf("Enter a no.");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a no. is:=%d",f);
}
int fact(int n)
int f=1;
{
 if(n=0)
return(f);
else
return(n*fact(n-1));
}
```

Passing parameters to a function:

Firstly, what are parameters?

parameters are the values that are passed to a function for processing.

There are 2 types of parameters:

a.) Actual Parameters.

b.) Formal Parameters.

a.) Actual Parameters:

These are the parameters which are used in main() function for function calling.

Syntax: <variable name>=<function name><actual argument>

Example: f=fact(n);

b.) Formal Parameters.

These are the parameters which are used in function defination for processing.

Methods of parameters passing:

1.) Call by reference.

2.) Call by value.

1.) Call by reference:

In this method of parameter passing , original values of variables are passed from calling program to function.

Thus,

Any change made in the function can be reflected back to the calling program.

2.) Call by value.

In this method of parameter passing, duplicate values of parameters are passed from calling program to function defination.

Thus,

Any change made in function would not be reflected back to the calling program.

Example: Program based upon call by value:

```
# include<stdio.h>
# include<conio.h>
void main()
{
int a,b;
a=10;
b=20;
void swap(int,int)
printf("The value of a before swapping=%d",a);
printf("The value of b before swapping=%d",b);
void swap(a,b);
printf("The value of a after swapping=%d",a);
printf("The value of b after swapping=%d",b);
```

```
}
void swap(int x, int y)
{
int t;
t=x;
x=y;
y=t;
}
```

**STORAGE CLASSES**

Every Variable in a program has memory associated with it.

Memory Requirement of Variables is different for different types of variables.

In C, Memory is allocated & released at different places

| **Term** | **Definition** |
|---|---|
| **Scope** | Region or Part of Program in which Variable is accessible |
| **Extent** | Period of time during which memory is associated with variable |
| **Storage Class** | Manner in which memory is allocated by the Compiler for Variable **Different Storage Classes** |

**Storage class of variable Determines following things**

Where the variable is stored

Scope of Variable

Default initial value

Lifetime of variable

**A. Where the variable is stored:**

Storage Class determines the location of variable, where it is declared. Variables declared with auto storage classes are declared inside main memory whereas variables declared with keyword register are stored inside the CPU Register.

**B. Scope of Variable**

Scope of Variable tells compile about the visibility of Variable in the block. Variable may have Block Scope, Local Scope and External Scope. A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

**C. Default Initial Value of the Variable**

Whenever we declare a Variable in C, garbage value is assigned to the variable. Garbage Value may be considered as initial value of the variable. C Programming have different storage classes which has different initial values such as Global Variable have Initial Value as 0 while the Local auto variable have default initial garbage value.

**D. Lifetime of variable**

Lifetime of the = Time Of variable Declaration - Time of Variable Destruction

Suppose we have declared variable inside main function then variable will be destroyed only when the control comes out of the main .i.e end of the program.

**Different Storage Classes:**

Auto Storage Class

Static Storage Class

Extern Storage Class

Register Storage Class

**Automatic (Auto) storage class**

This is default storage class

All variables declared are of type Auto by default

In order to Explicit declaration of variable use 'auto' keyword

auto int num1 ; // Explicit Declaration

**Features:**

| | |
|---|---|
| **Storage** | Memory |
| **Scope** | Local / Block Scope |

| | |
|---|---|
| **Life time** | Exists as long as Control remains in the block |
| **Default initial Value** | Garbage |

**Example**

```
void main()
{
 auto mum = 20 ;
 {
     auto num = 60 ;
          printf("nNum : %d",num);
 }
 printf("nNum : %d",num);
}
```

**Output :**

Num : 60

Num : 20



**Note :**

**Two variables are declared in different blocks , so they are treated as different variables**

**External ( extern ) storage class in C Programming**

Variables of this storage class are “Global variables”

Global Variables are declared outside the function and are accessible to all functions in the program

Generally , External variables are declared again in the function using keyword extern

In order to Explicit declaration of variable use ‘extern’ keyword

extern int num1 ;  // Explicit Declaration

**Features :**

| | |
|---|---|
| **Storage** | Memory |
| **Scope** | Global / File Scope |
| **Life time** | Exists as long as variable is running<br>Retains value within the function |
| **Default initial Value** | Zero |

**Example**

```
int num =  75 ;

void display();

void main()
{
 extern int num ;
          printf("nNum : %d",num);
 display();
}

void display()
{
 extern int num ;
          printf("nNum : %d",num);
}
```

**Output :**

Num : 75

Num : 75

**Note :**

Declaration within the function indicates that the function uses external variable

Functions belonging to same source code , does not require declaration (no need to write extern)

If variable is defined outside the source code , then declaration using extern keyword is required

**Static Storage Class**

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

**Register Storage Class**

register keyword is used to define local variable.

Local variable are stored in register instead of **RAM**.

As variable is stored in register, the **Maximum size of variable = Maximum Size of Register**

unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

This is generally used for **faster access**.

Common use is "**Counter**"

**Syntax**

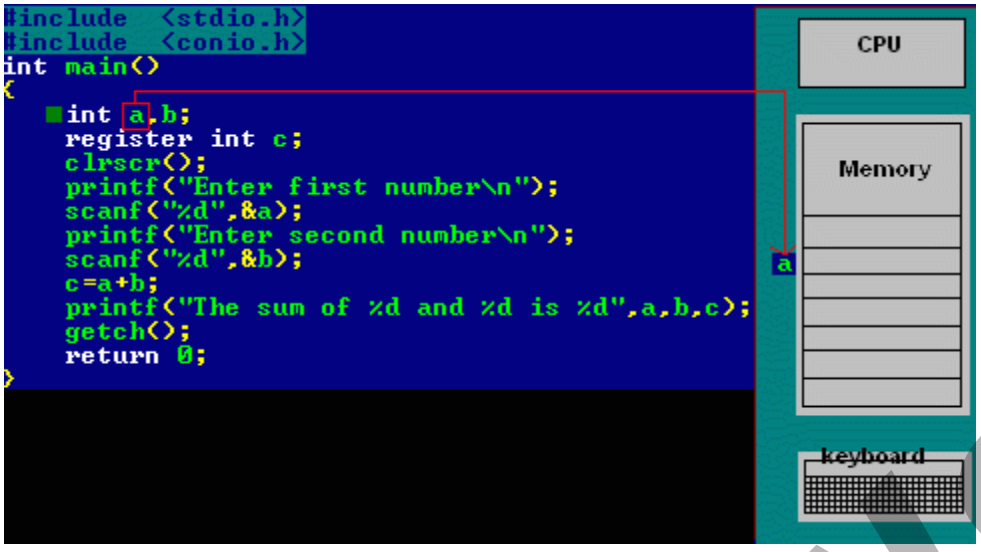
```
{
register int count;
}
```

**Register storage classes example**

```
#include<stdio.h>

int main()
{
int num1,num2;
register int sum;

printf("\nEnter the Number 1 : ");
scanf("%d",&num1);

printf("\nEnter the Number 2 : ");
scanf("%d",&num2);

sum = num1 + num2;

printf("\nSum of Numbers : %d",sum);

return(0);
}
```

**Explanation of program**

Refer below animation which depicts the register storage classes –




In the above program we have declared two variables num1,num2. These two variables are stored in RAM.

Another variable is declared which is stored in register variable.Register variables are stored in the register of the microprocessor.Thus memory access will be faster than other variables.

If we try to declare more register variables then it can treat variables asAuto storage variables as memory of microprocessor is fixed and limited.

**Why we need Register Variable ?**

Whenever we declare any variable inside C Program then memory will be randomly allocated at particular memory location.

We have to keep track of that memory location. We need to access value at that memory location using ampersand operator/Address Operator i.e (&).

If we store same variable in the register memory then we can access that memory location directly without using the Address operator.

Register variable will be accessed faster than the normal variable thus increasing the operation and program execution. Generally we use register variable as Counter.

**Note :** It is not applicable for arrays, structures or pointers.

**Summary of register Storage class**

| **Keyword** | **register** |
|---|---|
| Storage Location | CPU Register |

| **Keyword** | **register** |
|---|---|
| Initial Value | Garbage |
| Life | Local to the block in which variable is declared. |
| Scope | Local to the block. |

**Preprocessor directives**

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Below is the list of preprocessor directives that C language offers.

| **S.no** | **Preprocessor** | **Syntax** | **Description** |
|---|---|---|---|
| 1 | Macro | #define | This macro defines constant value and can be any of the basic data types. |
| 2 | Header file inclusion | #include <file_name> | The source code of the file "file_name" is included in the main program at the specified place |
| 3 | Conditional compilation | #ifdef, #endif, #if, #else, #ifndef | Set of commands are included or excluded in source program before compilation with respect to the |

| | | | |
|---|---|---|---|
| | | | condition |
| 4 | Other directives | #undef, #pragma | #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program |

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C:

#define – This macro defines constant value and can be any of the basic data types.
#include <file_name> – The source code of the file "file_name" is included in the main C program where "#include <file_name>" is mentioned.

```
#include <stdio.h>

#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{
  printf("value of height    : %d \n", height );
  printf("value of number : %f \n", number );
  printf("value of letter : %c \n", letter );
  printf("value of letter_sequence : %s \n", letter_sequence);
  printf("value of backslash_char  : %c \n", backslash_char);

}
```

OUTPUT:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

"#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.

Otherwise, "else" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
  #ifdef RAJU
  printf("RAJU is defined. So, this line will be added in " \
        "this C file\n");
  #else
  printf("RAJU is not defined\n");
  #endif
  return 0;
}
```

OUTPUT:

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

#ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.

Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100
```

```
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
          "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");

  #endif
  return 0;

}
```

OUTPUT:

```
SELVA is not defined. So, now we are going to define here
```

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

"If" clause statement is included in source file if given condition is true.

Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file since " \
        "a \= 100\n");
  #else
  printf("This line will be added in this C file since " \
        "a is not equal to 100\n");
  #endif
  return 0;
}
```

OUTPUT:

```
This line will be added in this C file since a = 100
```

EXAMPLE PROGRAM FOR UNDEF IN C:

This directive undefines existing macro in the program.

```
#include <stdio.h>

#define height 100
void main()
{
  printf("First defined value for height    : %d\n",height);
  #undef height          // undefining variable
  #define height 600     // redefining the same for new value
  printf("value of height after undef \& redefine:%d",height);
}
```

OUTPUT:

```
First defined value for height : 100
value of height after undef & redefine : 600
```

EXAMPLE PROGRAM FOR PRAGMA IN C:

Pragma is used to call a function before and after main function in a C program.

```
#include <stdio.h>

void function1( );
void function2( );

#pragma startup function1
#pragma exit function2

int main( )
{
  printf ( "\n Now we are in main function" ) ;
  return 0;
}

void function1( )
{
  printf("\nFunction1 is called before main function call");
}

void function2( )
{
  printf ( "\nFunction2 is called just before end of " \
        "main function" ) ;"
}
```

OUTPUT:

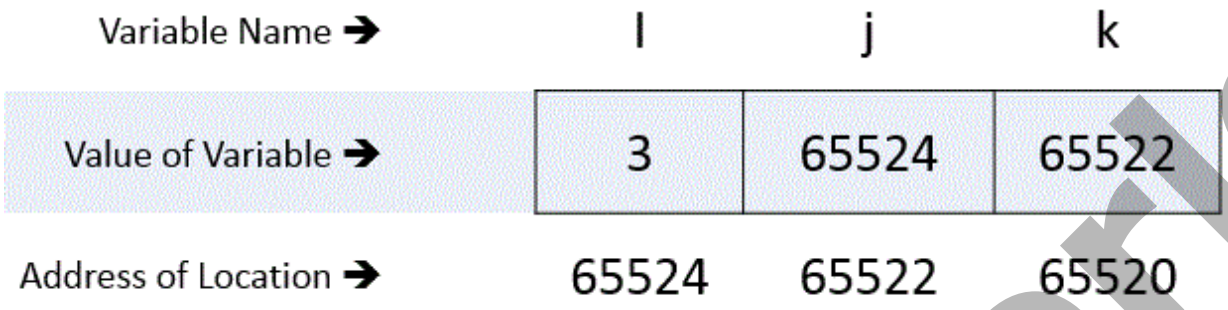
```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```

MORE ON PRAGMA DIRECTIVE IN C:

| S.no | Pragma command | description |
|---|---|---|
| 1 | #Pragma startup <function_name_1> | This directive executes function named “function_name_1” before |
| 2 | #Pragma exit <function_name_2> | This directive executes function named “function_name_2” just before termination of the program. |
| 3 | #pragma warn – rvl | If function doesn’t return a value, then warnings are suppressed by this directive while compiling. |
| 4 | #pragma warn – par | If function doesn’t use passed function parameter , then warnings are suppressed |
| 5 | #pragma warn – rch | If a non reachable code is written inside a program, such warnings are suppressed by this directive. |

**POINTERS**

Pointer Overview




Consider above Diagram which clearly shows pointer concept in c programming –

**i** is the name given for particular memory location of ordinary variable.

Let us consider it's Corresponding address be 65624 and the Value stored in variable **'i'** is 5

The address of the variable **'i'** is stored in another integer variable whose name is **'j'** and which is having corresponding address 65522

thus we can say that –

j = &i;

i.e

j = Address of i

Here j is not ordinary variable , It is special variable and called pointer variable as it stores the address of the another ordinary variable. We can summarize it like –

| Variable Name | Variable Value | Variable Address |
|---|---|---|
| i | 5 | 65524 |
| j | 65524 | 65522 |

B. C Pointer Basic Example:

```
#include <stdio.h>

int main()
{
  int *ptr, i;
  i = 11;

  /* address of i is assigned to ptr */
```

```
  ptr = &i;

  /* show i's value using ptr variable */
  printf("Value of i : %d", *ptr);

  return 0;
}
```

See Output and Download »

You will get value of i = 11 in the above program.

C. Pointer Declaration Tips :

1. Pointer is declared with preceding * :

int *ptr;  //Here ptr is Integer Pointer Variable

int ptr;   //Here ptr is Normal  Integer Variable

2. Whitespace while Writing Pointer :

pointer variable name and asterisk can contain whitespace because whitespace is ignored by compiler.

int *ptr;

int       * ptr;

int *          ptr;

All the above syntax are legal and valid. We can insert any number of spaces or blanks inside declaration. We can also split the declaration on multiple lines.

D. Key points for Pointer :

Unline ordinary variables pointer is special type of variable which stores the address of ordinary variable.

Pointer can only store the whole or integer number because address of any type of variable is considered as integer.

It is good to initialize the pointer immediately after declaration

& symbol is used to get address of variable

* symbol is used to get value from the address given by pointer.

E. Pointer Summary :

Pointer is Special Variable used to Reference and de-reference memory. (*Will be covered in upcoming chapter)

When we declare integer pointer then we can only store address of integer variable into that pointer.

Similarly if we declare character pointer then only the address of character variable is stored into the pointer variable.

| Pointer storing the address of following DT | Pointer is called as |
|---|---|
| Integer | Integer Pointer |
| Character | Character Pointer |
| Double | Double Pointer |
| Float | Float Pointer |

Pointer is a variable which stores the address of another variable

Since Pointer is also a kind of variable , thus pointer itself will be stored at different memory location.

2 Types of Variables :

Simple Variable that stores a value such as integer,float,character

Complex Variable that stores address of simple variable i.e pointer variables

Simple Pointer Example #1 :

```
#include<stdio.h>

int main()
{
int a = 3;
int *ptr;
ptr = &a;

return(0);
}
```

Explanation of Example :

| Point | Variable 'a' | Variable 'ptr' |
|---|---|---|
| Name of Variable | a | ptr |

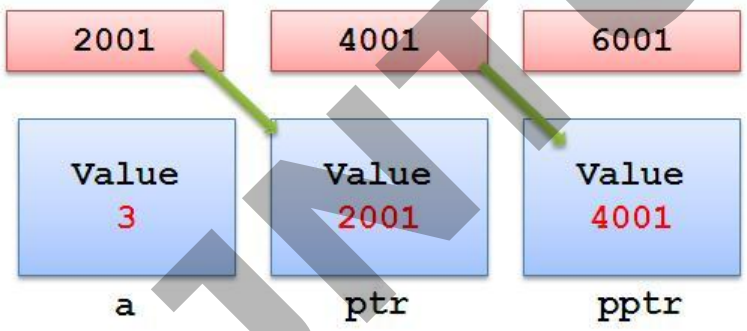
| Point | Variable 'a' | Variable 'ptr' |
|---|---|---|
| Type of Value that it holds | Integer | Address of Integer 'a' |
| Value Stored | 3 | 2001 |
| Address of Variable | 2001 (Assumption) | 4001 (Assumption) |

Simple Pointer Example #2 :

```
#include<stdio.h>

int main()
{
int a = 3;
int *ptr,**pptr;
ptr = &a;
pptr = &ptr;
return(0);
}
```

Explanation of Example

With reference to above program –




We have following associated points –

| Point | Variable 'a' | Variable 'ptr' | Variable 'pptr' |
|---|---|---|---|
| Name of Variable | a | ptr | pptr |
| Type of Value that it holds | Integer | Address of 'a' | Address of 'ptr' |
| Value Stored | 3 | 2001 | 4001 |

| Point | Variable 'a' | Variable 'ptr' | Variable 'pptr' |
|---|---|---|---|
| Address of Variable | 2001 | 4001 | 6001 |

Pointer address operator in C Programming

Pointer address operator is denoted by '&' symbol

When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

lets take an example –

&n - It gives an address on variable n

Working of address operator

```
#include<stdio.h>
void main()
{
int n = 10;
printf("\nValue of n is : %d",n);
printf("\nValue of &n is : %u",&n);
}
```

**Output :**

Value of n is : 10

Value of &n is : 1002

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with %u

printf("\nValue of &n is : %u",&n);

Understanding address operator

Consider the following program –

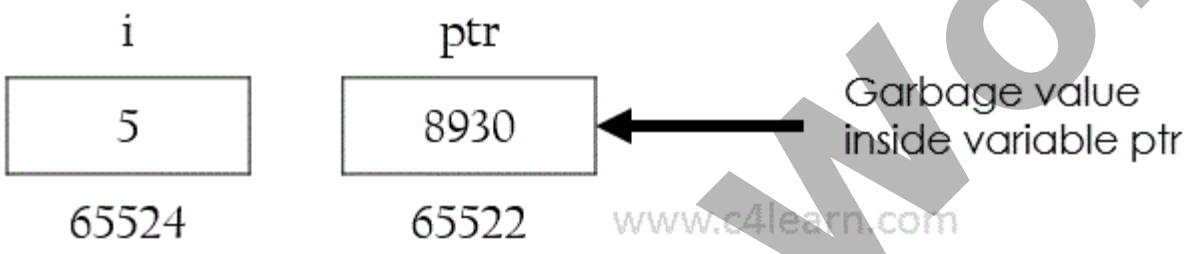
```
#include<stdio.h>
int main()
{
int i = 5;
int *ptr;
```

```
ptr = &i;

printf("\nAddress of i    : %u",&i);
printf("\nValue of ptr is : %u",ptr);

return(0);
}
```

After declaration memory map will be like this –

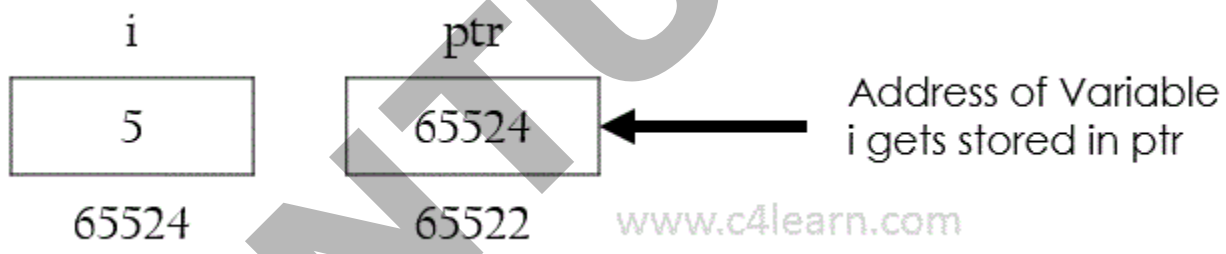
int i = 5;

int *ptr;




after Assigning the address of variable to pointer , i.e after the execution of this statement –

ptr = &i;




Invalid Use of pointer address operator

Address of literals

In C programming using address operator over literal will throw an error. We cannot use address operator on the literal to get the address of the literal.

&75

Only variables have an address associated with them, constant entity does not have corresponding address. Similarly we cannot use address operator over character literal –

&('a')

Character ‘a’ is literal, so we cannot use address operator.

Address of expressions

(a+b) will evaluate addition of values present in variables and output of (a+b)is nothing but Literal, so we cannot use Address operator

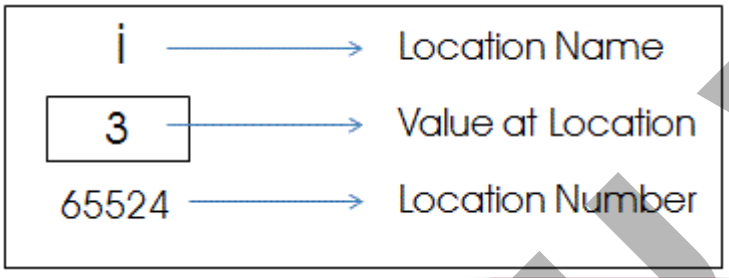
&(a+b)

Memory Organization for Pointer Variable:

When we use variable in program then Compiler keeps some memory for that variable depending on the **data type**

The address given to the variable is Unique with that variable name

When Program execution starts the **variable name** is automatically translated into the corresponding **address**.




Explanation :

Pointer Variable is nothing but a memory address which holds another address .

In the above program "i" is name given for memory location for human understanding , but compiler is unable to recognize "i" . Compiler knows only address.

In the next chapter we will be learning , Memory requirement for storing pointer variable.

Syntax for Pointer Declaration in C :

data_type *<pointer_name>;

Explanation :

data_type

Type of variable that the pointer **points to**

OR data type whose address is stored in **pointer_name**

Asterisk(*)

Asterisk is called as **Indirection Operator**

It is also called as **Value at address Operator**

It Indicates **Variable declared is of Pointer type**

pointer_name

Must be any **Valid C identifier**

Must follow all Rules of Variable name declaration

Ways of Declaring Pointer Variable:

[box] * can appears anywhere between Pointer_name and Data Type

```
int *p;
int *       p;
int    *  p;
```

Example of Declaring Integer Pointer:

```
int n = 20;
int *ptr;
```

Example of Declaring Character Pointer:

```
char ch = 'A';
char *cptr;
```

Example of Declaring Float Pointer:

```
float fvar = 3.14;
float *fptr;
```

How to Initialize Pointer in C Programming?

```
pointer = &variable;
```

Above is the syntax for initializing pointer variable in C.

Initialization of Pointer can be done using following 4 Steps :

Declare a Pointer Variable and Note down the Data Type.

Declare another Variable with Same Data Type as that of Pointer Variable.

Initialize Ordinary Variable and assign some value to it.

Now Initialize pointer by assigning the address of ordinary variable to pointer variable.

below example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{

int a;      // Step 1
int *ptr;    // Step 2
```

```
a = 10;      // Step 3
ptr = &a;    // Step 4

return(0);
}
```

Explanation of Above Program :

Pointer should not be used before initialization.

"ptr" is pointer variable used to store the address of the variable.

Stores address of the variable **'a'** .

Now "ptr" will contain the address of the variable "a" .

Note :

[box]Pointers are always initialized before using it in the program[/box]

Example : Initializing Integer Pointer

```
#include<stdio.h>
int main()
{
int a = 10;
int *ptr;

ptr = &a;
printf("\nValue of ptr : %u",ptr);

return(0);
}
```

**Output :**

Value of ptr : 4001

**Pointer arithematic**

Incrementing Pointer:

Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.

Incrementing Pointer Variable Depends Upon data type of the Pointer variable

Formula : ( After incrementing )

new value = current address + i * size_of(data type)

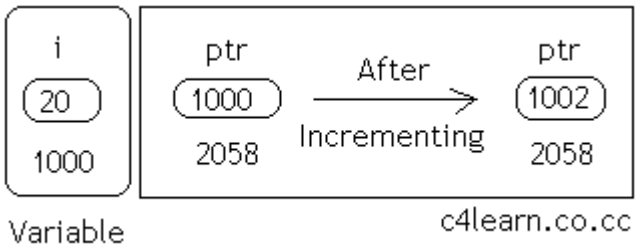
Three Rules should be used to increment pointer –

Address + 1 = Address

Address++ = Address

++Address = Address

**Pictorial Representation :**




| Data Type | Older Address stored in pointer | Next Address stored in pointer after incrementing (ptr++) |
|---|---|---|
| int | 1000 | 1002 |
| float | 1000 | 1004 |
| char | 1000 | 1001 |

Explanation : Incremeting Pointer

Incrementing a pointer to an integer data will cause its **value to be incremented by 2** .

This differs from compiler to compiler as memory required to store integer **vary compiler to compiler**

[box]**Note to Remember :** Increment and Decrement Operations on pointer should be used when we have Continues memory (in Array).[/box]

Live Example 1 : Increment Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;

ptr=ptr+1;
printf("New Value of ptr : %u",ptr);
```

```
return 0;
}
```

**Output :**

New Value of ptr : 1002

Live Example 2 : Increment Double Pointer

```
#include<stdio.h>

int main(){

double *ptr=(double *)1000;

ptr=ptr+1;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

**Output :**

New Value of ptr : 1004

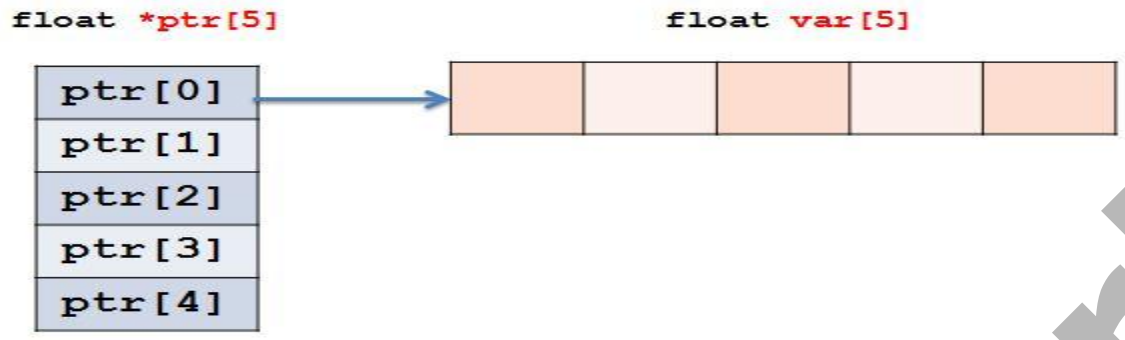
Live Example 3 : Array of Pointer

```
#include<stdio.h>

int main(){

float var[5]={1.1f,2.2f,3.3f};
float(*ptr)[5];

ptr=&var;
printf("Value inside ptr : %u",ptr);

ptr=ptr+1;
printf("Value inside ptr : %u",ptr);

return 0;
}
```

**Output :**

Value inside ptr : 1000

Value inside ptr : 1020




Explanation :

Address of ptr[0] = 1000

We are storing Address of float array to ptr[0]. –

Address of ptr[1]

= Address of ptr[0] + (Size of Data Type)*(Size of Array)

= 1000 + (4 bytes) * (5)

= 1020

**Address of Var[0]…Var[4] :**

Address of var[0] = 1000

Address of var[1] = 1004

Address of var[2] = 1008

Address of var[3] = 1012

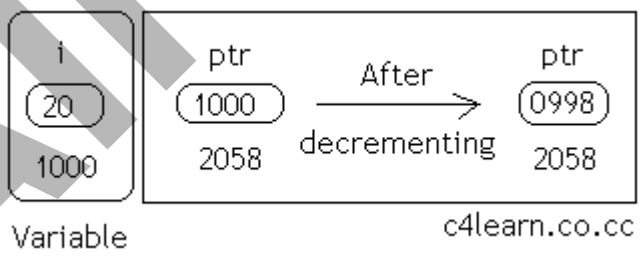
Address of var[4] = 1016

Formula : ( After decrementing )

new_address = (current address) - i * size_of(data type)

[box]Decrementation of Pointer Variable Depends Upon : data type of the Pointer variable[/box]

Example :



| Data Type | Older Address stored in pointer | Next Address stored in pointer after incrementing (ptr–) |
|---|---|---|
| int | 1000 | 0998 |
| float | 1000 | 0996 |
| char | 1000 | 0999 |

Explanation:

**Decrementing a pointer** to an integer data will cause its value to be decremented by 2

This differs from compiler to compiler as memory required to store integer vary **compiler to compiler**

Pointer Program: Difference between two integer Pointers

```
#include<stdio.h>

int main(){

float *ptr1=(float *)1000;
float *ptr2=(float *)2000;

printf("\nDifference : %d",ptr2-ptr1);

return 0;
}
```

**Output :**

Difference : 250

Explanation :

Ptr1 and Ptr2 are two pointers which holds memory address of Float Variable.

Ptr2-Ptr1 will gives us number of floating point numbers that can be stored.

ptr2 - ptr1 = (2000 - 1000) / sizeof(float)

= 1000 / 4

= 250

Live Example 2:

```
#include<stdio.h>

struct var{
```

```
    char cvar;
    int ivar;
    float fvar;
};

int main(){

struct var *ptr1,*ptr2;

ptr1 = (struct var *)1000;
ptr2 = (struct var *)2000;

printf("Difference= %d",ptr2-ptr1);

return 0;
}
```

Output :

Difference = 142

Explanation :

ptr2-ptr1 = (2000 - 1000)/Sizeof(struct var)

= 1000 / (1+2+4)

= 1000 / 7

= 142

Adding integer value with Pointer

In C Programming we can add any integer number to Pointer variable. It is perfectly legal in c programming to add integer to pointer variable.

In order to compute the final value we need to use following formulae :

final value = (address) + (number * size of data type)

Consider the following example –

```
int *ptr , n;
ptr = &n ;
ptr = ptr + 3;
```

Live Example 1 : Increment Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;
```

```
ptr=ptr+3;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

New Value of ptr : 1006

Explanation of Program :

In the above program –

int *ptr=(int *)1000;

this line will store 1000 in the pointer variable considering 1000 is memory location for any of the integer variable.

Formula :

ptr = ptr + 3 * (sizeof(integer))

= 1000 + 3 * (2)

= 1000 + 6

= 1006

Similarly if we have written above statement like this –

float *ptr=(float *)1000;

then result may be

ptr = ptr + 3 * (sizeof(float))

= 1000 + 3 * (4)

= 1000 + 12

= 1012

Suppose we have subtracted “n” from pointer of any data type having initial addess as “init_address” then after subtraction we can write –

ptr = initial_address - n * (sizeof(data_type))

Subtracting integer value with Pointer

int *ptr , n;

ptr = &n ;

ptr = ptr - 3;

Live Example 1 : Decrement Integer Pointer

```
#include<stdio.h>

int main(){

int *ptr=(int *)1000;

ptr=ptr-3;
printf("New Value of ptr : %u",ptr);

return 0;
}
```

Output :

New Value of ptr : 994

Formula :

```
ptr = ptr   - 3 * (sizeof(integer))
   = 1000  - 3 * (2)
   = 1000  - 6
   = 994
```

Summary :

Pointer - Pointer = Integer

Pointer - Integer = Pointer

Differencing Pointer in C Programming Language :

Differencing Means **Subtracting two Pointers**.

Subtraction gives the Total number of objects between them .

Subtraction indicates “How apart the two Pointers are ?”

C Program to Compute Difference Between Pointers :

```
#include<stdio.h>

int main()
{
```

```
int num , *ptr1 ,*ptr2 ;

ptr1 = &num ;
ptr2 = ptr1 + 2 ;

printf("%d",ptr2 - ptr1);

return(0);
}
```

Output :

2

ptr1 stores the **address of Variable** num

Value of ptr2 is incremented by **4 bytes**

Differencing two Pointers

Important Observations :

Suppose the Address of Variable num = 1000.

| Statement | Value of Ptr1 | Value of Ptr2 |
|---|---|---|
| int num , *ptr1 ,*ptr2 ; | Garbage | Garbage |
| ptr1 = &num ; | 1000 | Garbage |
| ptr2 = ptr1 + 2 ; | 1000 | 1004 |
| ptr2 - ptr1 | 1000 | 1004 |

Computation of Ptr2 – Ptr1 :

Remember the following formula while computing the difference between two pointers –

Final Result = (ptr2 - ptr1) / Size of Data Type

Step 1 : Compute Mathematical Difference (Numerical Difference)

ptr2 - ptr1 = Value of Ptr2 - Value of Ptr1

= 1004 - 1000

= 4

Step 2 : Finding Actual Difference (Technical Difference)

Final Result = 4 / Size of Integer

= 4 / 2

= 2

Numerically Subtraction ( ptr2-ptr1 ) differs by 4

As both are Integers they are numerically Differed by 4 and Technically by 2 objects

Suppose Both pointers of float the they will be differed numerically by 8 and Technically by 2 objects

Consider the below statement and refer the following table –

int num = ptr2 - ptr1;

and

| If Two Pointers are of Following Data Type | Numerical Difference | Technical Difference |
|---|---|---|
| Integer | 2 | 1 |
| Float | 4 | 1 |
| Character | 1 | 1 |

Comparison between two Pointers :

**Pointer comparison is Valid** only if the **two pointers are Pointing to same array**

All Relational Operators can be used for comparing pointers of **same type**

**All Equality and Inequality Operators** can be used with all Pointer types

Pointers **cannot be Divided or Multiplied**

Point 1 : Pointer Comparison

```
#include<stdio.h>

int main()
{
int *ptr1,*ptr2;

ptr1 = (int *)1000;
ptr2 = (int *)2000;

if(ptr2 > ptr1)
   printf("Ptr2 is far from ptr1");

return(0);
}
```

Pointer Comparison of Different Data Types :

```
#include<stdio.h>

int main()
{
int *ptr1;
float *ptr2;

ptr1 = (int *)1000;
ptr2 = (float *)2000;

if(ptr2 > ptr1)
  printf("Ptr2 is far from ptr1");

return(0);
}
```

Explanation :

**Two Pointers of different data types can be compared** .

In the above program we have compared two pointers of different data types.

It is perfectly **legal in C Programming**.

[box]As we know Pointers can store Address of any data type, address of the data type is "Integer" so we can compare address of any two pointers although they are of different data types.[/box]

Following operations on pointers :

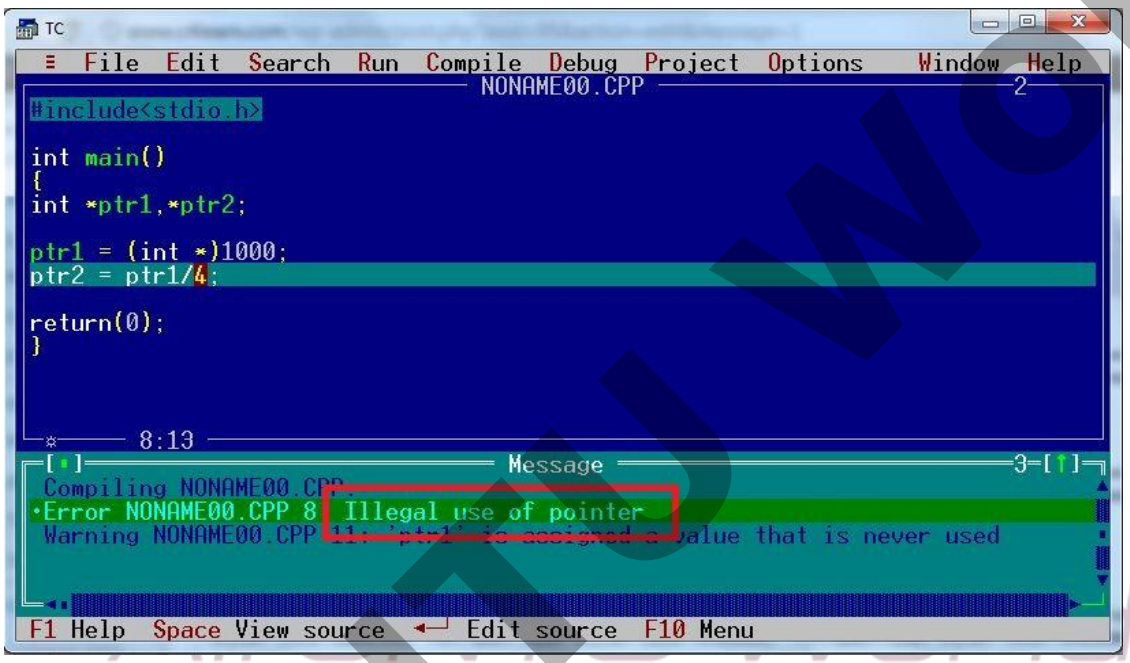
| | |
|---|---|
| > | Greater Than |
| < | Less Than |
| >= | Greater Than And Equal To |
| <= | Less Than And Equal To |
| == | Equals |
| != | Not Equal |

Divide and Multiply Operations :

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

```
int main()
{
int *ptr1,*ptr2;

ptr1 = (int *)1000;
ptr2 = ptr1/4;

return(0);
}
```

**Output :**






Pointer to pointer

Pointer to Pointer in C Programming

**Declaration : Double Pointer**

int **ptr2ptr;

**Consider the Following Example :**

int num = 45 , *ptr , **ptr2ptr ;

ptr = #

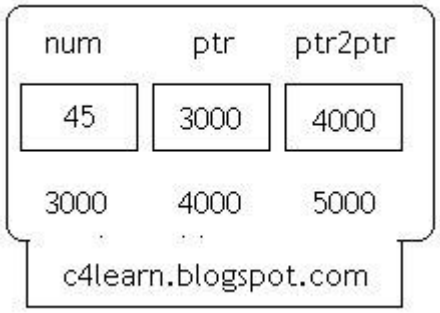
ptr2ptr = &ptr;

What is Pointer to Pointer ?

Double (**) is used to denote the **double Pointer**

Pointer Stores the address of the Variable

Double Pointer **Stores the address of the Pointer Variable**




| Statement | What will be the Output ? |
|---|---|
| *ptr | 45 |
| **ptr2ptr | 45 |
| ptr | &n |
| ptr2ptr | &ptr |

**Notes :**

Conceptually we can have Triple ….. n pointers

Example : *****n,****b can be another example

**Live Example :**

```
#include<stdio.h>

int main()
{
int num = 45 , *ptr , **ptr2ptr ;
ptr     = &num;
ptr2ptr = &ptr;

printf("%d",**ptr2ptr);
```

```
return(0);
}
```
**Output :**

45

## UNIT-IV

## STRUCTURES AND UNIONS

### INTRODUCTION TO STRUCTURE

As we know that Array is collection of the elements of same type , but many time we have to store the elements of the different data types.

Suppose Student record is to be stored, then for storing the record we have to group together all the information such as Roll, name, Percent which may be of different data types.

Ideally Structure is collection of different variables under single name.

Basically Structure is for storing the complicated data.

A structure is a convenient way of grouping several pieces of related information together.

**Definition of Structure in C**

Structure is composition of the different variables of different data types, grouped under same name.

```
typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
  } student;
```

**Some Important Definitions of Structures**

Each member declared in Structure is called **member.**

```
char name[64];
char course[128];
int age;
int year;
```

are some examples of members.

Name given to structure is called as **tag**

Structure **member** may be of **different data type** including **user defined data-type** also

```
typedef struct {
        char name[64];
        char course[128];
        book b1;
        int year;
  } student;
```

Here book is user defined data type.

**Declaring Structure Variable in C**

In C we can group some of the user defined or primitive data types together and form another compact way of storing complicated information is called as Structure. Let us see how to declare structure in c programming language –

**Syntax of Structure in C Programming**

```
struct tag
{
   data_type1 member1;
   data_type2 member2;
   data_type3 member3;
};
```

**Structure Alternate Syntax**

```
struct <structure_name>
{
    structure_Element1;
    structure_Element2;
```

```
    structure_Element3;
    ...
    ...
};
```

Some Important Points Regarding Structure in C Programming:

**Struct** keyword is used to declare structure.

**Members of structure** are enclosed within opening and closing braces.

**Declaration** of Structure reserves **no space**.

It is nothing but the " **Template / Map / Shape** " of the structure .

Memory is created, very first time when the **variable is created** /**Instance** is created.

Different Ways of Declaring Structure Variable:

Way 1 : Immediately after Structure Template

```
struct date
{
    int date;
    char month[20];
    int year;
}today;

// 'today' is name of Structure variable
```

Way 2 : Declare Variables using struct Keyword

```
struct date
{
    int date;
    char month[20];
```

```
    int year;
};
```

**struct** date today;

where "**date**" is name of structure and "**today**" is name of variable.

Way 3 : Declaring Multiple Structure Variables

```
struct Book
{
    int pages;
    char name[20];
    int year;

}book1,book2,book3;
```

**C Structure Initialization**

When we declare a structure, memory is not allocated for un-initialized variable.

Let us discuss very familiar example of structure student , we can initialize structure variable in different ways –

Way 1 : Declare and Initialize

```
struct student
{
   char name[20];
   int roll;
   float marks;
}std1 = { "Pritesh",67,78.3 };
```

In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

std1 = { "Pritesh",67,78.3 }

This is the code for initializing structure variable in C programming

Way 2 : Declaring and Initializing Multiple Variables

```
struct student
{
  char name[20];
  int roll;
  float marks;
}

std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

```
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

Way 3 : Initializing Single member

```
struct student
{
  int mark1;
  int mark2;
  int mark3;
} sub1={67};
```

Though there are three members of structure,only one is initialized , Then remaining two members are initialized with Zero. If there are variables of other data type then their initial values will be –

| Data Type | Default value if not initialized |
|---|---|
| integer | 0 |
| float | 0.00 |
| char | NULL |

Way 4 : Initializing inside main

```
struct student
{
   int mark1;
   int mark2;
   int mark3;
};

void main()
{
struct student s1 = {89,54,65};
- - - - --
- - - - --
- - - - --
};
```

When we declare a structure then memory won't be allocated for the structure. i.e only writing below declaration statement will never allocate memory

```
struct student
{
```

```
    int mark1;
    int mark2;
    int mark3;
};
```

We need to initialize structure variable to allocate some memory to the structure.

**struct** student s1 = {89,54,65};

Some Structure Declarations and It's Meaning :

```
struct
 {
    int length;
    char *name;
}*ptr;
```

Suppose we initialize these two structure members with following values –

length = 30;

*name  = "programming";

Now Consider Following Declarations one by One –

| Member | Value | Address |
|---|---|---|
| length | 30 | 3000 |
| name | programming | 3002 |

Example 1 : Incrementing Member

++ptr->length

"++" Operator is **pre-increment operator**.

Above Statement will increase the value of "**length**"

Example 2 : Incrementing Member

(++ptr)->length

Content of the length is fetched and then ptr is incremented.

**Consider above Structure and Look at the Following Table:-**

| Expression | Meaning |
|---|---|
| ++ptr->length | Increment the value of length |
| (++ptr)->length | Increment ptr before accessing length |
| (ptr++)->length | Increment ptr after accessing length |
| *ptr->name | Fetch Content of name |
| *ptr->name++ | Incrementing ptr after Fetching the value |
| (*ptr->name)++ | Increments whatever str points to |
| *ptr++->name | Incrementing ptr after accessing whatever str points to |

**Accessing Structure Members**

Array elements are accessed using the Subscript variable, Similarly Structure members are accessed using dot [.] operator.

(.) is called as "Structure member Operator".

Use this Operator in between **"Structure name"** & **"member name"**

Live Example :

#include<stdio.h>

```
struct Vehicle
   {
   int wheels;
   char vname[20];
   char color[10];
}v1 = {4,"Nano","Red"};

int main()
{
printf("Vehicle No of Wheels : %d",v1.wheels);
printf("Vehicle Name          : %s",v1.vname);
printf("Vehicle Color         : %s",v1.color);
return(0);
}
```

Output :

Vehicle No of Wheels : 4

Vehicle Name : Nano

Vehicle Color : Red

Note :

Dot operator has Highest Priority than unary, arithmetic, relational, logical Operators

**Initializing Array of Structure in C Programming**

Array elements are stored in consecutive memory Location.

Like Array , Array of Structure can be initialized at compile time.

Way1 : Initializing After Declaring Structure Array :

```
struct Book
{
    char bname[20];
    int pages;
    char author[20];
    float price;
}b1[3] = {
        {"Let us C",700,"YPK",300.00},
        {"Wings of Fire",500,"APJ Abdul Kalam",350.00},
        {"Complete C",1200,"Herbt Schildt",450.00}
       };
```

Explanation :

As soon as after declaration of structure we initialize structure with the pre-defined values. For each structure variable we specify set of values in curly braces. Suppose we have 3 Array Elements then we have to initialize each array element individually and all individual sets are combined to form single set.

```
{"Let us C",700,"YPK",300.00}
```

Above set of values are used to initialize first element of the array. Similarly –

```
{"Wings of Fire",500,"APJ Abdul Kalam",350.00}
```

is used to initialize second element of the array.

Way 2 : Initializing in Main

```
struct Book
  {
    char bname[20];
    int pages;
    char author[20];
```

```
    float price;
};
void main()
{
struct Book b1[3] = {
        {"Let us C",700,"YPK",300.00},
        {"Wings of Fire",500,"Abdul Kalam",350.00},
        {"Complete C",1200,"Herbt Schildt",450.00}
    };
}
```

**Some Observations and Important Points:**

Tip #1 : All Structure Members need not be initialized

```
#include<stdio.h>
struct Book
{
    char bname[20];
    int pages;
    char author[20];
    float price;
}b1[3] = {
        {"Book1",700,"YPK"},
        {"Book2",500,"AAK",350.00},
        {"Book3",120,"HST",450.00}
        };
```

```
void main()
{

printf("\nBook Name    : %s",b1[0].bname);
printf("\nBook Pages   : %d",b1[0].pages);
printf("\nBook Author  : %s",b1[0].author);
printf("\nBook Price   : %f",b1[0].price);

}
```

Output :

Book Name : Book1

Book Pages : 700

Book Author : YPK

Book Price : 0.000000

Explanation :

In this example , While initializing first element of the array we have not specified the price of book 1.It is not mandatory to provide initialization for all the values. Suppose we have 5 structure elements and we provide initial values for first two element then we cannot provide initial values to remaining elements.

```
{"Book1",700,,90.00}
```

above initialization is illegal and can cause compile time error.

Tip #2 : Default Initial Value

```
struct Book
{
    char bname[20];
    int pages;
```

```
    char author[20];
    float price;
}b1[3] = {
        {},
        {"Book2",500,"AAK",350.00},
        {"Book3",120,"HST",450.00}
       };
```

Output :

Book Name :

Book Pages : 0

Book Author :

Book Price : 0.000000

It is clear from above output , Default values for different data types.

| Data Type | Default Initialization Value |
|---|---|
| Integer | 0 |
| Float | 0.0000 |
| Character | Blank |

**Passing Array of Structure to Function in C Programming**

Array of Structure can be passed **to function as a Parameter**.

Function can also return Structure as **return type**.

Structure can be passed as follow

Live Example :

```
#include<stdio.h>
#include<conio.h>
//-------------------------------------
struct Example
{
  int num1;
  int num2;
}s[3];
//-------------------------------------
void accept(struct Example sptr[],int n)
{
  int i;
  for(i=0;i<n;i++)
  {
  printf("\nEnter num1 : ");
  scanf("%d",&sptr[i].num1);
  printf("\nEnter num2 : ");
  scanf("%d",&sptr[i].num2);
  }
}
//-------------------------------------
void print(struct Example sptr[],int n)
{
  int i;
  for(i=0;i<n;i++)
```

```
  {
  printf("\nNum1 : %d",sptr[i].num1);
  printf("\nNum2 : %d",sptr[i].num2);
  }
}
//-------------------------------------
void main()
{
int i;
clrscr();
accept(s,3);
print(s,3);
getch();
}
```

Output :

```
Enter num1 : 10
Enter num2 : 20
Enter num1 : 30
Enter num2 : 40
Enter num1 : 50
Enter num2 : 60
Num1 : 10
Num2 : 20
Num1 : 30
Num2 : 40
```

Num1 : 50

Num2 : 60

Explanation :

Inside main structure and size of structure array is passed.

When reference (i.e ampersand) is not specified in main , so this passing is simple pass by value.

Elements can be accessed by using dot [.] operator

**Pointer Within Structure in C Programming:**

Structure may contain the **Pointer variable as member**.

Pointers are used to store the address of memory location.

They can be **de-referenced** by **'*'** operator.

Example :

```
struct Sample
{
  int  *ptr;  //Stores address of integer Variable
  char *name; //Stores address of Character String
}s1;
```

**s1** is structure variable which is used to access the **"structure members"**.

```
s1.ptr  = &num;
s1.name = "Pritesh"
```

Here **num** is any variable but it's address is stored in the Structure member ptr (**Pointer to Integer**)

Similarly Starting address of the String "Pritesh" is stored in structure variable name(**Pointer to Character array**)

Whenever we need to print the content of variable **num** , we are dereferancing the pointer variable num.

```
printf("Content of Num : %d ",*s1.ptr);
printf("Name : %s",s1.name);
```

Live Example : Pointer Within Structure

```
#include<stdio.h>

struct Student
{
  int  *ptr;  //Stores address of integer Variable
  char *name; //Stores address of Character String
}s1;

int main()
{

int roll = 20;
s1.ptr   = &roll;
s1.name  = "Pritesh";

printf("\nRoll Number of Student : %d",*s1.ptr);
printf("\nName of Student        : %s",s1.name);

return(0);
}
```

Output :

Roll Number of Student : 20

Name of Student : Pritesh

Some Important Observations :

printf("\nRoll Number of Student : %d",*s1.ptr);

We have stored the address of variable ‘roll’ in a pointer member of structure thus we can access value of pointer member directly using de-reference operator.

printf("\nName of Student : %s",s1.name);

Similarly we have stored the base address of string to pointer variable ‘name’. In order to de-reference a string we never use de-reference operator.

**Array of Structure :**

Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Example :

```
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}Book[100];
```

Explanation :

Here Book structure is used to Store the information of one Book.

In case if we need to store the Information of 100 books then Array of Structure is used.

b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books.

book[3] is shown Below

| | Name | Pages | Price |
|---|---|---|---|
| Book[0] | | | |
| Book[1] | | | |
| Book[2] | | c4learn.blogspot.com | |

Accessing Pages field of Second Book :

Book[1].pages

Live Example :

```
#include <stdio.h>


struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}book[3];


int main(int argc, char *argv[])
{
int i;


for(i=0;i<3;i++)
   {
   printf("\nEnter the Name of Book    : ");
   gets(book[i].bname);
   printf("\nEnter the Number of Pages : ");
```

```
    scanf("%d",book[i].pages);
    printf("\nEnter the Price of Book   : ");
    scanf("%f",book[i].price);
    }
printf("\n--------- Book Details ------------ ");
    for(i=0;i<3;i++)
    {
    printf("\nName of Book    : %s",book[i].bname);
    printf("\nNumber of Pages : %d",book[i].pages);
    printf("\nPrice of Book   : %f",book[i].price);
    }
return 0;
}
```

Output of the Structure Example:

```
Enter the Name of Book    : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book    : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book    : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500

--------- Book Details ------------
```

Name of Book : ABC

Number of Pages : 100

Price of Book : 200

Name of Book : EFG

Number of Pages : 200

Price of Book : 300

Name of Book : HIJ

Number of Pages : 300

Price of Book : 500

Union in C Programming :

In C Programming we have came across Structures. Unions are similar tostructure syntactically.Syntax of both is almost similar. Let us discuss some important points one by one –

Note #1 : **Union and Structure are Almost Similar**

```
union stud
{
  int roll;
  char name[4];
  int marks;
}s1;
```

```
struct stud
{
  int roll;
  char name[4];
  int marks;
}s1;
```

If we look at the two examples then we can say that both structure and union are same except Keyword.

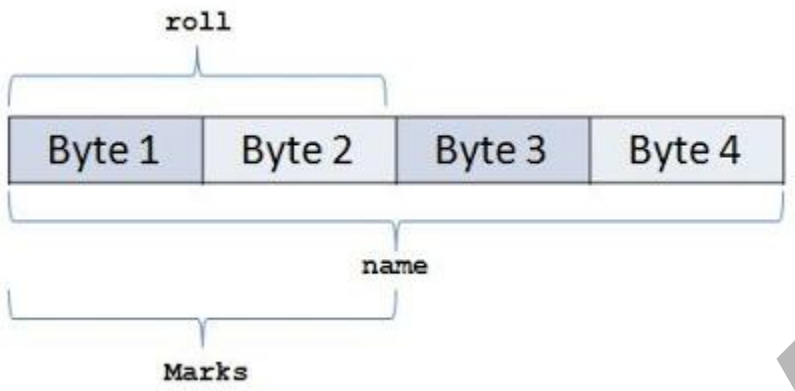
Note #2 : Multiple Members are Collected Together Under Same Name

```
int roll;
char name[4];
```

**int** marks;

We have collected three variables of different data type under same name together.

Note #3 : All Union Members Occupy Same Memory Area




For the union maximum memory allocated will be equal to the data member with maximum size. In the example character array 'name' have maximum size thus maximum memory of the union will be 4 Bytes.

Maximum Memory of Union = Maximum Memory of Union

Data Member

Note #4 : Only one Member will be active at a time.

Suppose we are accessing one of the data member of union then we cannot access other data member since we can access single data member of union because each data member shares same memory. By Using Union we can **Save Lot of Valuable Space**

Simple Example:

**union** u

{

**char** a;

**int** b;

}

How to Declare Union in C ?

Union is similar to that of Structure. Syntax of both are same but major difference between structure and union is '**memory storage**'.

In structures, **each member has its own storage location**, whereas all the members of union use the same location. Union contains many members of different types,

Union can handle **only one member at a time**.

Syntax :

```
union tag
{
  union_member1;
  union_member2;
  union_member3;
  ..
  ..
  ..
  union_memberN;
}instance;
```

Note :

Unions are Declared in the same way as a Structure.Only "**struct Keyword**" is replaced with **union**

Sample Declaration of Union :

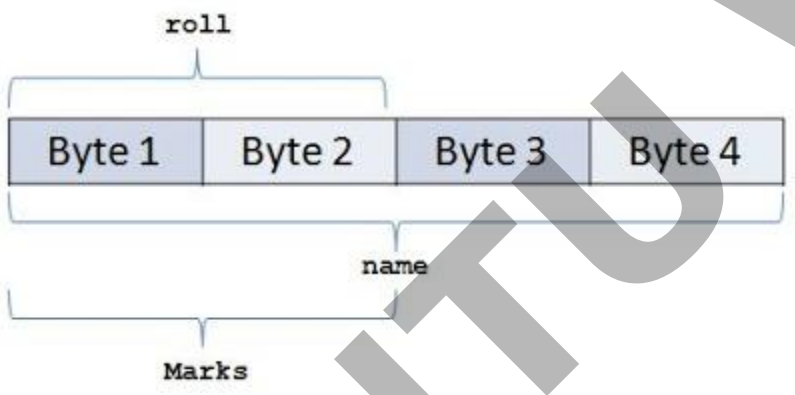
```
union stud
{
  int roll;
  char name[4];
  int marks;
}s1;<
```

```
union stud
{
int roll;
char name[4];
int marks;
}s1;
```

www.c4learn.com

| Member | Memory Required |
|---|---|
| Roll | 2 |
| Name | 4 |
| Marks | 2 |

How Memory is Allocated ?




So From the Above fig. We can Conclude –

Union Members that compose a union, **all share the same storage area within the computers memory**

Each member within a structure is assigned its own **unique storage area**

Thus unions are used to observe memory.

Unions are useful for **application involving multiple members**, where values need not be assigned to all the members at any one time.

**C Programming accessing union members**

While accessing union, we can have access to single data member at a time. we can access single union member using following two Operators –

Using DOT Operator

Using ARROW Operator

Accessing union members DOT operator

In order to access the member of the union we are using the dot operator. DOT operator is used inside printf and scanf statement to get/set value from/of union member location.

**Syntax :**

variable_name.member

consider the below union, when we declare a variable of union type then we will be accessing union members using dot operator.

**union** emp

{

**int** id;

**char** name[20];

}e1;

id can be Accessed by – union_variable.member

| **Syntax** | **Explanation** |
|---|---|
| e1.id | Access id field of union |
| e1.name | Access name field of union |

**Accessing union members Arrow operator**

Instead of maintain the union variable suppose we store union at particular address then we can access the members of the union using pointer to the union and arrow operator.

**union** emp

{

**int** id;

**char** name[20];

}*e1;

id can be Accessed by – union_variable->member

| Syntax | Explanation |
|---|---|
| e1->id | Access id field of union |
| e1->name | Access name field of union |

C Programs

Program #1 : Using dot operator

```
#include <stdio.h>
union emp
{
  int id;
  char name[20];
}e1;
int main(int argc, char *argv[])
{
    e1.id = 10;
    printf("\nID   : %d",e1.id);
    strcpy(e1.name,"Pritesh");
    printf("\nName : %s",e1.name);
    return 0;
}
```

**Output :**

ID   : 10

Name : Pritesh

Program #2 : Accessing same memory

```
#include <stdio.h>
union emp
{
  int id;
  char name[20];
}e1;
int main(int argc, char *argv[])
{
    e1.id = 10;
    strcpy(e1.name,"Pritesh");
    printf("\nID   : %d",e1.id);
    printf("\nName : %s",e1.name);
    return 0;
}
```

**Output :**

```
ID   : 1953067600
Name : Pritesh
```

As we already discussed in the previous article of union basics, we have seen how memory is shared by all union fields. In the above example –

Total memory for union = max(sizeof(id),sizeof(name))

= sizeof(name)

= 20 bytes

Firstly we have utilized first two bytes out of 20 bytes for storing integer value. After execution of statement again same memory is overridden by character array so while printing the ID value, garbage value gets printed

Program #3 : Using arrow operator

```
#include <stdio.h>
union emp
{
  int id;
  char name[20];
}*e1;
int main(int argc, char *argv[])
{
    e1->id = 10;
    printf("\nID   : %d",e1->id);
    strcpy(e1->name,"Pritesh");
    printf("\nName : %s",e1->name);
    return 0;
}
```

**Output :**

```
ID   : 10
Name : Pritesh
```

Bit fiels:

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

```
struct {
   unsigned int widthValidated : 1;
   unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept −

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

/* define simple structure */
struct {
   unsigned int widthValidated;
   unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
   unsigned int widthValidated : 1;
```

```
   unsigned int heightValidated : 1;
} status2;
 int main( ) {
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

**Bit Field Declaration**

The declaration of a bit-field has the following form inside a structure −

```
struct {
   type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field −

| Elements | Description |
|---|---|
| type | An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| member_name | The name of the bit-field. |
| width | The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. |

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

```
struct {
   unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example −

```
#include <stdio.h>
#include <string.h>
struct {
   unsigned int age : 3;
} Age;
int main( ) {
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );
   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );
   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );
   return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
```

Age.age : 0

**Typedef:**

The C programming language provides a keyword called **typedef**, which you can use to give a type, a new name. Following is an example to define a term**BYTE** for one-byte numbers −

typedef unsigned char BYTE;

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.

BYTE  b1, b2;

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

typedef unsigned char byte;

You can use **typedef** to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

```
#include <stdio.h>

#include <string.h>

 typedef struct Books {

   char title[50];

   char author[50];

   char subject[100];

   int book_id;

} Book;

 int main( ) {

   Book book;

    strcpy( book.title, "C Programming");

   strcpy( book.author, "Nuha Ali");

   strcpy( book.subject, "C Programming Tutorial");
```

```
   book.book_id = 6495407;
    printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```

typedef vs #define

**#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences −

**typedef** is limited to giving symbolic names to types only where as**#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.

**typedef** interpretation is performed by the compiler whereas **#define**statements are processed by the pre-processor.

The following example shows how to use #define in a program −

```
#include <stdio.h>
 #define TRUE  1
#define FALSE 0
 int main( ) {
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of TRUE : 1
Value of FALSE : 0
```

**Enumerated data type:**

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword **enum** is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, ***type_name*** is the name of enumerated data type or tag. And ***value1***,***value2***,....,***valueN*** are values of type ***type_name***.

By default, ***value1*** will be equal to 0, ***value2*** will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements
enum suit{
   club=0;
   diamonds=10;
   hearts=20;
   spades=3;
};
```

**Declaration of enumerated variable**

Above code defines the type of the data but, no any variable is created. Variable of type **enum** can be created as:

```
enum boolean{
   false;
   true;
```

```
};
enum boolean check;
```

Here, a variable check is declared which is of type **enum boolean**.

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main(){
   enum week today;
   today=wednesday;
   printf("%d day",today+1);
   return 0;
  }
```

Output

```
4 day
```

You can write any program in C language without the help of enumerations but, enumerations helps in writing clear codes and simplify programming.

**Dynamic memory allocation**

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not has any technique to allocated memory dynamically, there are 4 library functions under **"stdlib.h"** for dynamic memory allocation.

| Function | Use of Function |
|---|---|

| Function | Use of Function |
|---|---|
| malloc() | Allocates requested size of bytes and returns a pointer first byte of allocated space |
| calloc() | Allocates space for an array elements, initializes to zero and then returns a pointer to memory |
| free() | dellocate the previously allocated space |
| realloc() | Change the size of previously allocated space |

malloc()

The name malloc stands for "memory allocation". The function **malloc()**reserves a block of memory of specified size and return a pointer of type **void**which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size)

Here, ***ptr*** is pointer of cast-type. The **malloc()** function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr=(int*)malloc(100*sizeof(int));

This statement will allocate either 200 or 400 according to size of **int** 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr=(cast-type*)calloc(n,element-size);

This statement will allocate contiguous space in memory for an array of ***n***elements. For example:

ptr=(float*)calloc(25,sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement cause the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>

#include <stdlib.h>

int main(){

   int n,i,*ptr,sum=0;

   printf("Enter number of elements: ");

   scanf("%d",&n);

   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc

   if(ptr==NULL)

   {

      printf("Error! memory not allocated.");

      exit(0);

   }

   printf("Enter elements of array: ");

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

   {
```

```
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
```

```
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, ***ptr*** is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
```

```
    for(i=0;i<n2;++i)
        printf("%u\t",ptr+i);
    return 0;
}
```

## UNIT-IV

## FILES

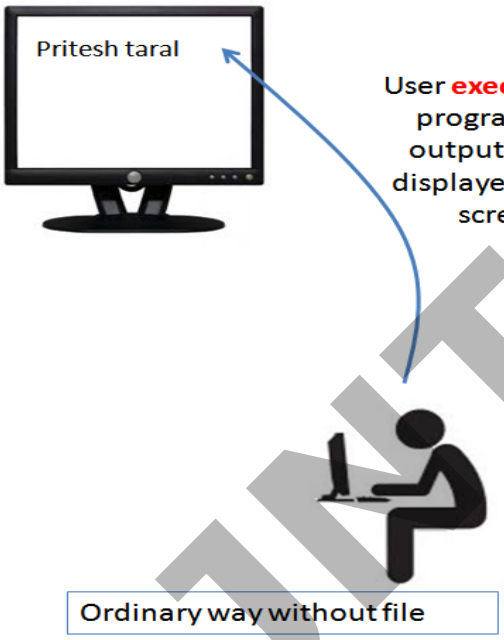
### DRAWBACKS OF TRADITIONAL I/O SYSTEM

Until now we are using Console Oriented I/O functions.

"Console Application" means an application that has a text-based interface. (black screen window))

Most applications require a large amount of data , if this data is entered through console then it will be quite time consuming task

Main drawback of using Traditional I/O :- data is temporary (and will not be available during re-execution )




Consider example –

We have written C Program to accept person detail from user and we are going to print these details back to the screen.

Now consider another scenario, suppose we want to print same data that we have entered previously.

We cannot save data which was entered on the console before.

Now we are storing data entered (during first run) into text file and when we need this data back (during 2nd run), we are going to read file.

**Introduction to file handling in C**

New way of dealing with data is file handling.

Data is stored onto the disk and can be retrieve whenever require.

Output of the program may be stored onto the disk

In C we have many functions that deals with file handling

A file is a collection of bytes stored on a secondary storage device(generally a disk)

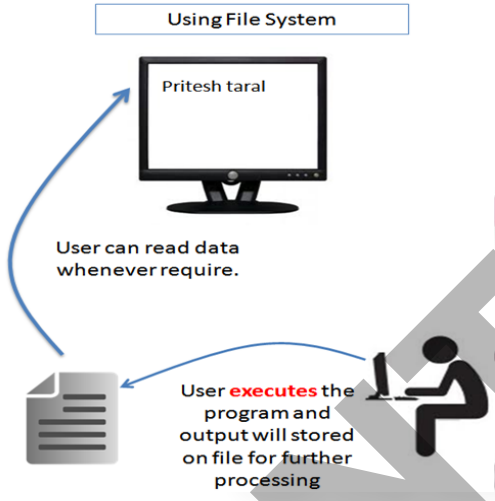
Collection of byte may be interpreted as –

Single character

Single Word

Single Line

Complete Structure.




**File I/O Streams in C Programming Language**

In C all input and output is done with streams

Stream is nothing but the sequence of bytes of data

A sequence of bytes flowing into program is called input stream

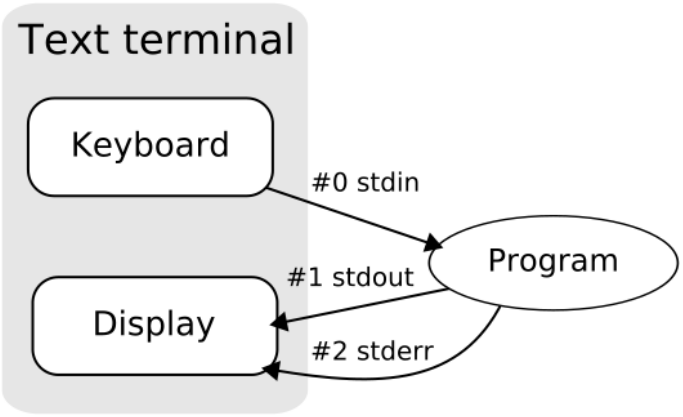
A sequence of bytes flowing out of the program is called output stream

Use of Stream make I/O machine independent.

Predefined Streams :

| | |
|---|---|
| stdin | Standard Input |
| stdout | Standard Output |

| | |
|---|---|
| stdin | Standard Input |
| stderr | Standard Error |




**Standard Input Stream Device**

stdin stands for (Standard Input)

Keyboard is standard input device .

Standard input is data (Often Text) going into a program.

The program requests data transfers by use of the read operation.

Not all programs require input.

**Standard Output Stream Device**

stdout stands for (Standard Output)

Screen(Monitor) is standard output device .

Standard output is data (Often Text) going out from a program.

The program sends data to output device by using write operation.

Difference Between Std. Input and Output Stream Devices :

| Point | Std i/p Stream Device | Standard o/p Stream Device |
|---|---|---|
| Stands For | Standard Input | Standard Output |
| Example | Keyboard | Screen/Monitor |

| Point | Std i/p Stream Device | Standard o/p Stream Device |
|---|---|---|
| Data Flow | Data (Often Text) going into a program | data (Often Text) going out from a program |
| Operation | Read Operation | Write Operation |

Some Important Summary :

| Point | Input Stream | Output Stream |
|---|---|---|
| Standard Device 1 | Keyboard | Screen |
| Standard Device 2 | Scanner | Printer |
| IO Function | scanf and gets | printf and puts |
| IO Operation | Read | Write |
| Data | Data goes from stream | data comes into stream |

**Text file Format in C Programming**

Text File is Also Called as "Flat File".

Text File Format is Basic File Format in C Programming.

Text File is simple Sequence of ASCII Characters.

Each Line is Characterized by EOL Character (End of Line).

```
1000233   Miralda          John
1000234   Faley            Nick
1000235   Baylog           Cathy
1000236   Gallardo         Mike
1000237   Christian        Daniel
1000238   Baufield         Daniel
1000239   Frazier          Robert
1000240   Garrido          Edward
1000241   Williams         Zachary
1000242   Morel            David
          Padilla          Damian
1000244   Rosenberg        Wayne
1000245   Blanchard        Phong S
1000246   Wiggins          David
1000247   Miller           Jeffrey
1000248   Coon             Terry
1000249   Chretien         Walter
1000250   Myers            Timothy

1000233   Miralda          John
1000234   Faley            Nick
1000235   Baylog           Cathy
```

**Text File Formats**

Text File have .txt Extension.

Text File Format have Little contains very little formatting .

The precise definition of the .txt format is not specified, but typicallymatches the format accepted by the system terminal or simple text editor.

Files with the .txt extension can easily be read or opened by any program that reads text and, for that reason, are considered universal (or platform independent).

Text Format Contain Mostly English Characters

**What are Binary Files**

Binary Files Contain Information Coded Mostly in Binary Format.

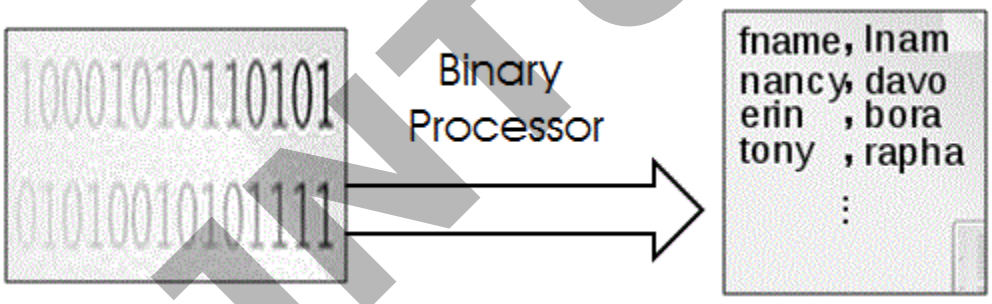
Binary Files are **difficult to read for human**.

Binary Files can be processed by **certain applications or processors**.

Only **Binary File Processors can understood Complex Formatting**Information Stored in Binary Format.

Humans can read binary **files only after processing**.

All Executable Files are **Binary Files**.




Explanation :

As shown in fig. Binary file is stored in Binary Format (in 0/1). This Binary file is difficult to read for humans. So generally Binary file is given as input to the Binary file Processor. Processor will convert binary file into equivalent readable file.

**Some Examples of the Binary files :**

Executable Files

Database files

Before opening the file we must understand the basic concept of file in C Programming , Types of File. If we want to display some message on the console from the file then we must open it in read mode.

**Opening and Defining FILE in C Programming**

Before storing data onto the secondary storage , firstly we must specify following things –

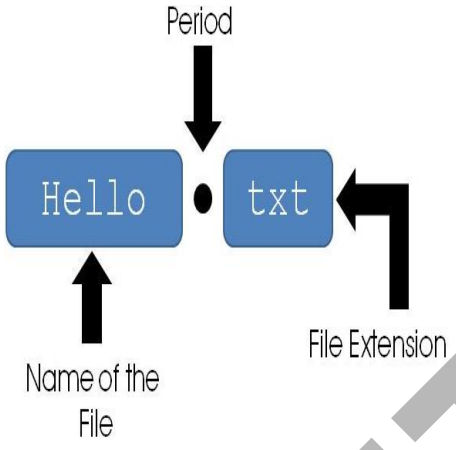
File name

Data Structure

Perpose / Mode

Very first task in File handling is to open file

File name : Specifies Name of the File




File name consists of **two fields**

First field is **name field** and second field is of **extension field**

**Extension field is optional**

Both File name and extension are separated by period or dot.

**Data Structure**

Data structure of file is defined as FILE in the library of standard I/O functions

In short we have to declare the pointer variable of type FILE

**Mode of FILE opening**

In C Programming we can open file in different modes such as reading mode,writing mode and appending mode depending on purpose of handling file.

Following are the different Opening modes of File :

| Opening Mode | Purpose | Previous Data |
|---|---|---|
| Reading | File will be opened just for reading purpose | Retained |
| Writing | File will be opened just for writing purpose | Flushed |
| Appending | File will be opened for appending some thing in file | Retained |

**Different Steps to Open File**

Step 1 : Declaring FILE pointer

Firstly we need pointer variable which can point to file. below is the syntax fordeclaring the file pointer.

FILE *fp;

Step 2 : Opening file hello.txt

fp = fopen ("filename","mode");

Live Example : Opening the File and Defining the File

```
#include<stdio.h>

int main()
{
FILE *fp;
char ch;

fp = fopen("INPUT.txt","r") // Open file in Read mode

fclose(fp); // Close File after Reading

return(0);
}
```

If we want to open file in different mode then following syntax will be used –

| Reading Mode | fp = fopen("hello.txt","r"); |
|---|---|
| Writing Mode | fp = fopen("hello.txt","w"); |
| Append Mode | fp = fopen("hello.txt","a"); |

Opening the File : Yet Another Live Example

```
#include<stdio.h>

void main()
{
 FILE *fp;
 char ch;
 fp = fopen("INPUT.txt","r"); // Open file in Read mode

  while(1)
  {
  ch = fgetc(fp); // Read a Character
    if(ch == EOF ) // Check for End of File
      break ;

  printf("%c",ch);
  }
 fclose(fp); // Close File after Reading
}
```

**File Opening Mode Chart**

| Mode | Meaning | fopen Returns if FILE- | |
|---|---|---|---|
| | | **Exists** | **Not Exists** |
| r | Reading | – | NULL |
| w | Writing | Over write on Existing | Create New File |
| a | Append | – | Create New File |
| r+ | Reading + Writing | New data is written at the beginning overwriting existing data | Create New File |
| w+ | Reading + Writing | Over write on Existing | Create New File |
| a+ | Reading + | New data is appended at the end of file | Create New |

| | | | |
|---|---|---|---|
| | Appending | | File |

Explanation :

File can be opened in **basic 3 modes** : Reading Mode, Writing Mode, Appending Mode

If File is not present on the path specified then **New File can be created using Write and Append Mode**.

Generally we used to open **following types of file in C** –

| File Type | Extension |
|---|---|
| C Source File | .c |
| Text File | .txt |
| Data File | .dat |

Writing on the file will **overwrite previous content**

EOF and feof function >> stdio.h >> File Handling in C

Syntax :

int feof(FILE *stream);

What it does?

Macro tests if end-of-file has been reached on a stream.

feof is a macro that tests the given stream for an end-of-file indicator.

Once the indicator is set, read operations on the file return the indicatoruntil rewind is called, or the file is closed.

The end-of-file indicator is reset with each input operation.

**Ways of Detecting End of File**

A ] In Text File :

Special Character EOF denotes the end of File

As soon as Character is read,End of the File can be detected

EOF is defined in stdio.h

Equivalent value of EOF is -1

Printing Value of EOF :

```
void main()
{
printf("%d", EOF);
}
```

B ] In Binary File :

feof function is used to detect the end of file

It can be used in text file

feof Returns TRUE if end of file is reached

Syntax :

```
int feof(FILE *fp);
```

Ways of Writing feof Function :

Way 1 : In if statement :

```
if( feof(fptr) == 1 ) // as if(1) is TRUE
printf("End of File");
```

Way 2 : In While Loop

```
while(!feof(fptr))
{
--- - --
--- - --
}
```

**C - Command Line Arguments**

main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,

argc

argv[]

where,

argc – Number of arguments in the command line including program name
argv[] – This is carrying all the arguments

In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.

For example, when we compile a program (test.c), we get executable file in the name "test".

Now, we run the executable "test" along with 4 arguments in command line like below.

**./test this is a program**

Where,

argc = 5
argv[0] = "test"
argv[1] = "this"
argv[2] = "is"
argv[3] = "a"
argv[4] = "program"
argv[5] = NULL

EXAMPLE PROGRAM FOR ARGC () AND ARGV() FUNCTIONS IN C:

```
#include <stdio.h>

#include <stdlib.h>

int main(int argc, char *argv[])   //  command line arguments

{

if(argc!=5)

{

  printf("Arguments passed through command line " \

       "not equal to 5");

  return 1;

}
```

```
    printf("\n Program name  : %s \n", argv[0]);
    printf("1st arg  : %s \n", argv[1]);
    printf("2nd arg  : %s \n", argv[2]);
    printf("3rd arg  : %s \n", argv[3]);
    printf("4th arg  : %s \n", argv[4]);
    printf("5th arg  : %s \n", argv[5]);

return 0;
}
```

OUTPUT:

```
Program name : test
1st arg : this
2nd arg : is
3rd arg : a
4th arg : program
5th arg : (null)
```