COMPUTER PROGRAMMING Using C

LECTURE NOTES

Branch : Computer Science & Engineering

Prepared by : Prashant Singh Yaav

Lecturer Computer Science & Engineering

MAHAMYA POLYTECHNIC OF INFORMATION TECHNOLOGY,SALEMPUR HATHRAS

UNIT-I

INTRODUCTION TO COMPUTERS

COMPUTER SYSTEMS

―AComputer 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

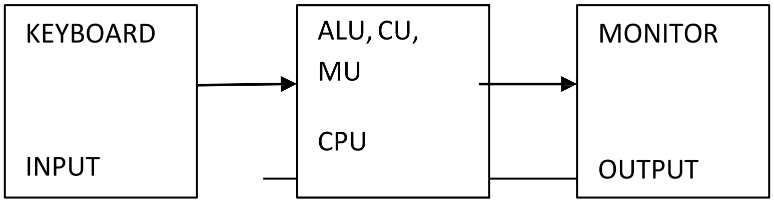
c) Software

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.

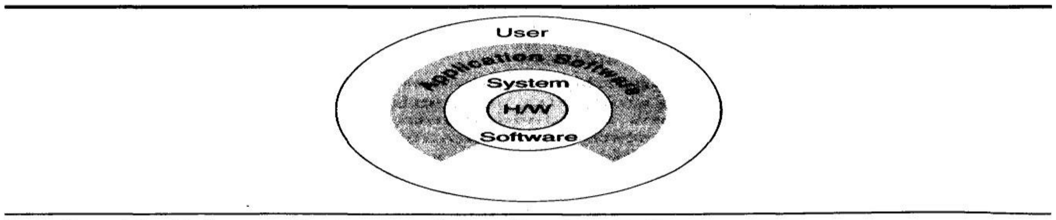
Application software

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.

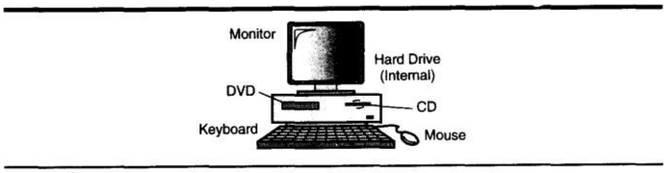
The following are the different kinds of computing environments available

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

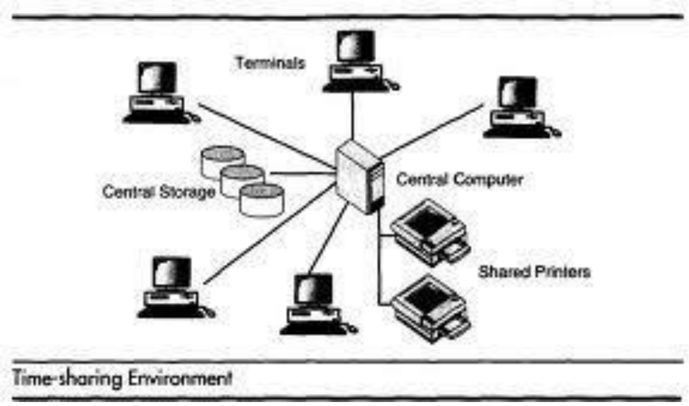


Personal Computing Environment

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.

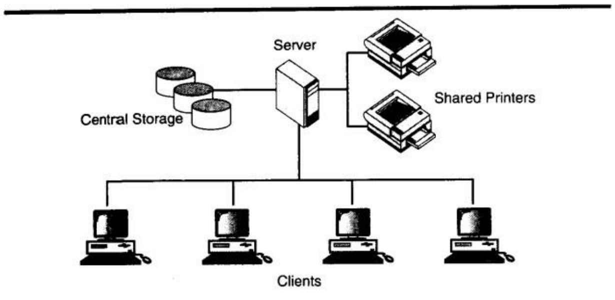
The complete processing is done by the central computer. The computer which ask for processing are only dumb terminals.



Time-sharing Environment

Client/Server Environment

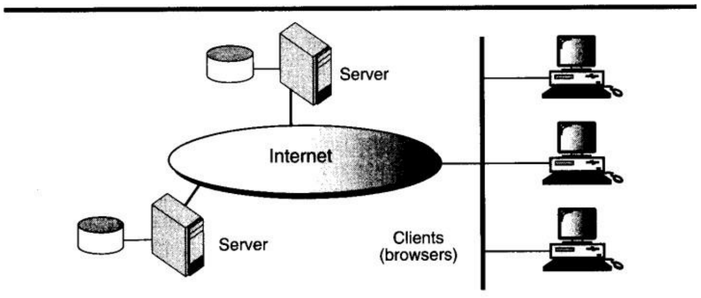
A Client/Server Computing involves the processing between two machines. A client Machine is the one which requests processing. Server Machine is the one which offers the processing. Hence the client is Capable enough to do processing. A portion of processing is done by client and the core(important)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



The Client/Server Environment

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.



Distributed Computing

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:

4) Machine dependent
5) Programming is very difficult
6) Difficult to understand
7) Difficult to write bug free programs
8) 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:

5) Machine Dependent Language
6) Requires translator
7) Difficult to learn and write programs
8) Slow development time
9) 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:

9) Needs translator
10) Requires high execution time
11) Poor control on hardware
12) 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

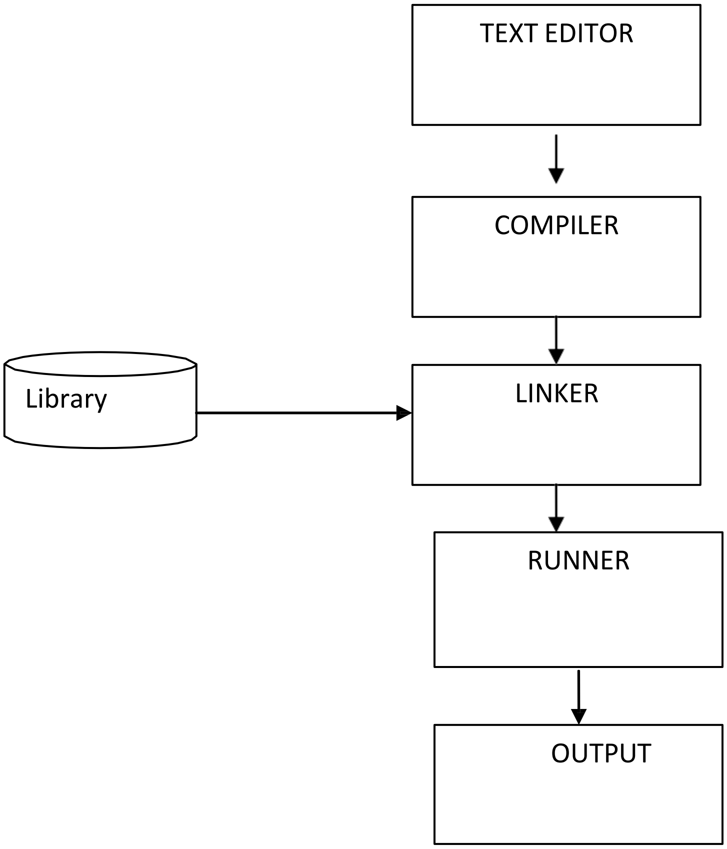
COMPILER	INTERPRETER
A Compiler is used to compile an entire program and an executable program is generated through the object program	An interpreter is used to translate each line of the program code immediately as it is entered

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.

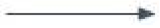
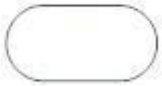
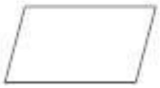
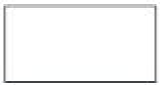
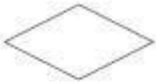
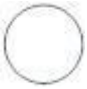
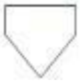
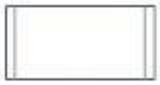
```
step 1 : start
step 2 : input number
step 3 : rem=number mod 2
step 4 : if rem=0 then
             print "number even"
           else
             print "number odd"
           endif
step 5 : stop
```

FLOWCHART

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

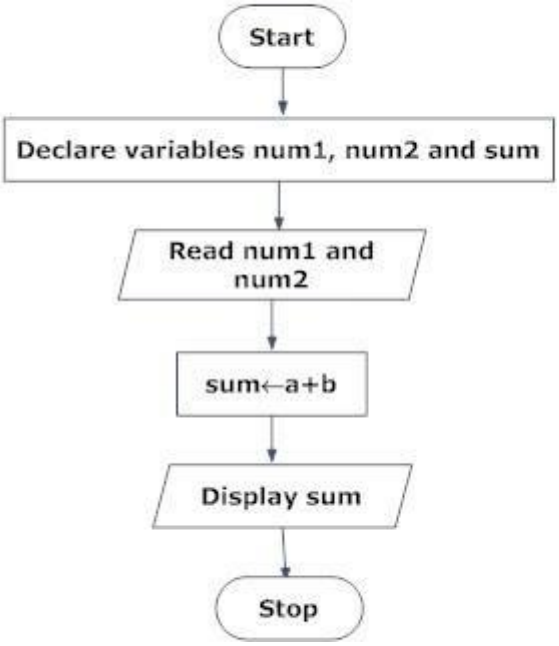
Symbols Used In Flowchart

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

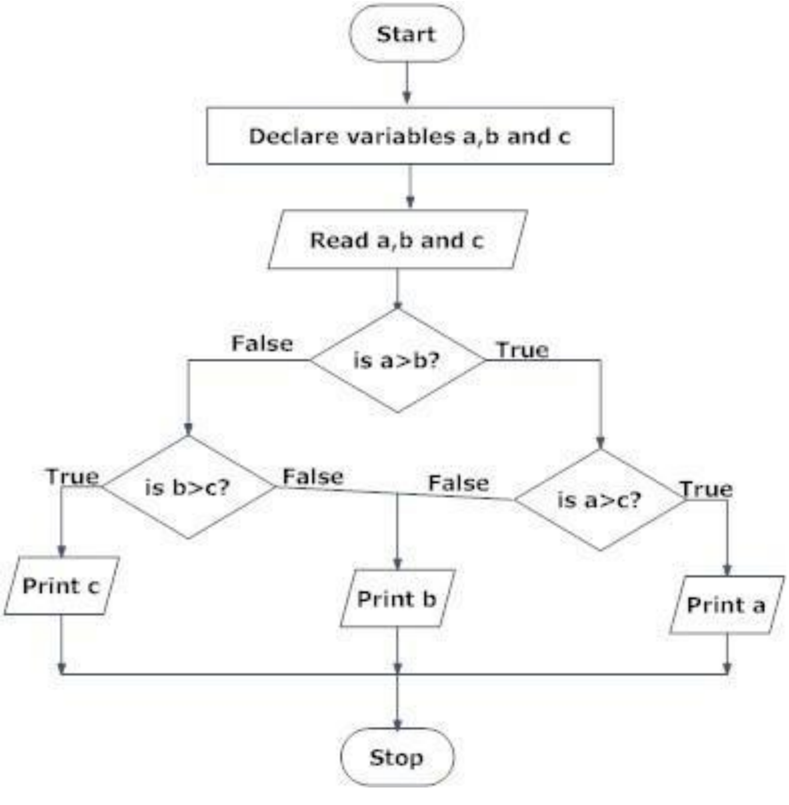
Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for airthmetic operations and data-manipulations.
	Desicion	Used to represent 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.

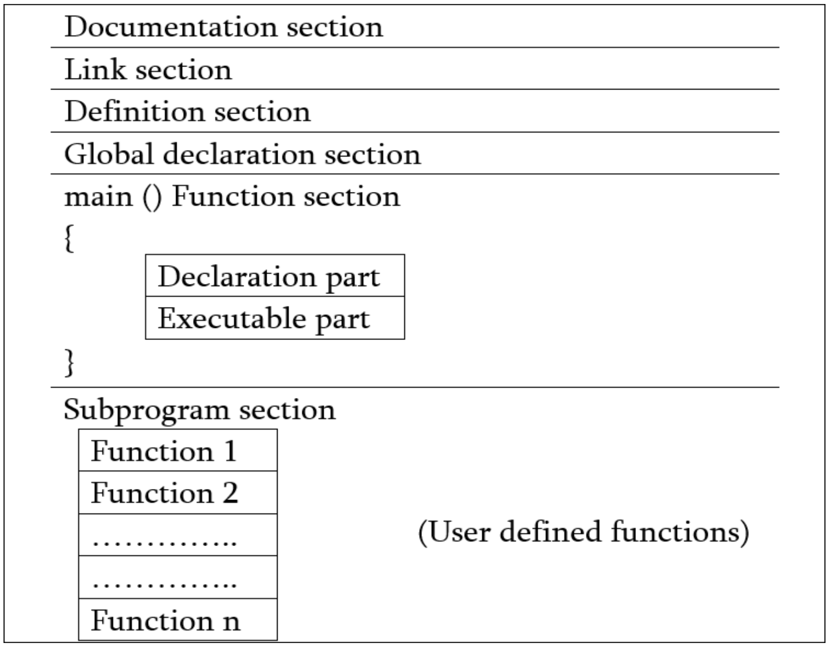
Many of the important ideas of C stem from the language **BCPL,** developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-

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



Documentation section

Link section

Definition section

Global declaration section

main () Function section

{

Declaration part

Executable part

}

Subprogram section

Function 1

Function 2

…………..

…………..

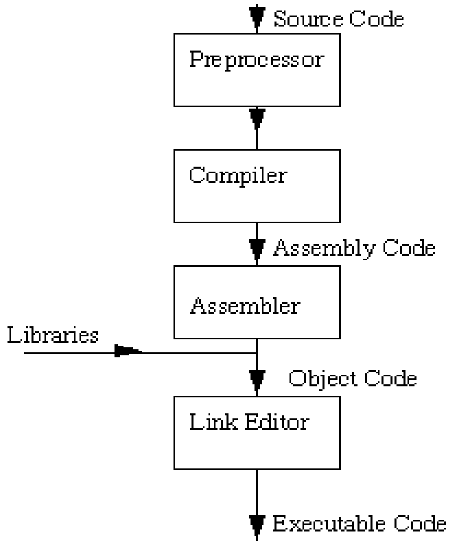
Function n

(User defined functions)

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 = 111111111111111111111111 111111111111111111111111111111010111

.. ..= -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 is0000000000000000000000000000000000000000000000000010100000000000. 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. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. 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
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	

++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) 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
< <= > >=	Relational less than/less than or equal to 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
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment 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 (―%d‖,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

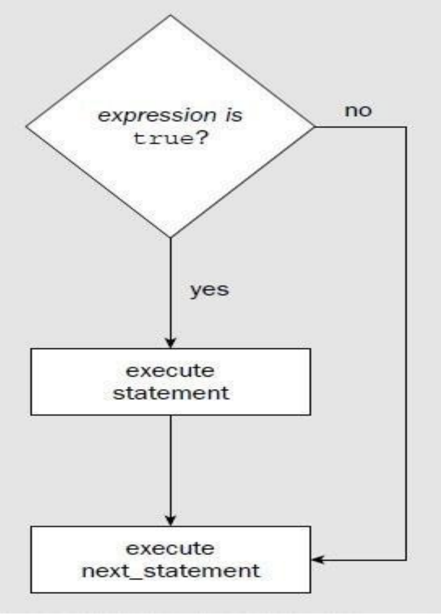
If statement:

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.

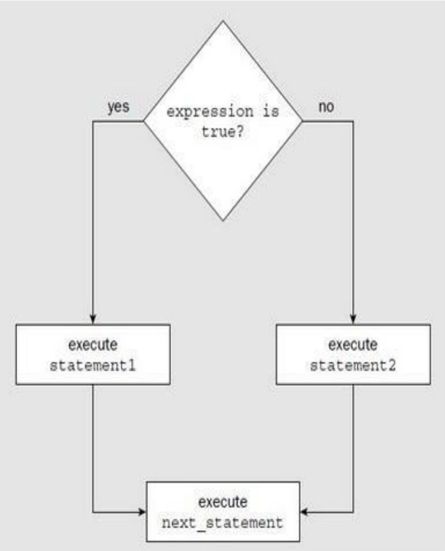
Sample Program on if-else Statement :

```
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?

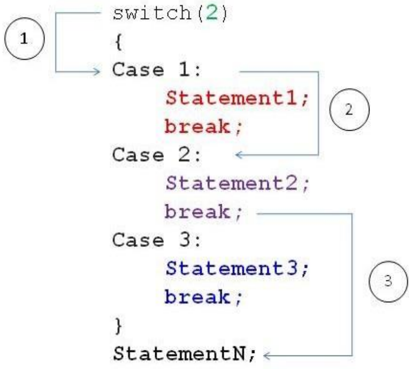
```
if(Condition 1)
   Statement 1
else
   {
   Statement 2
   if(condition 2)
      {
      if(condition 3)
```

```
        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 :



```
switch(2)
{
Case 1:
      Statement1;
      break;
Case 2:
      Statement2;
      break;
Case 3:
      Statement3;
      break;
}
StatementN;
```

*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..

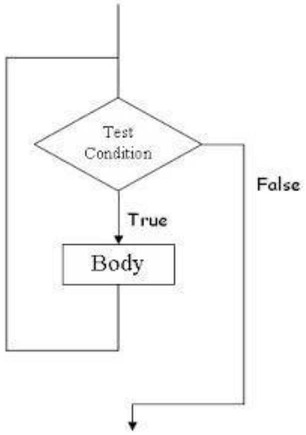
When it finds matching case it will execute block of code specified in that case.

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

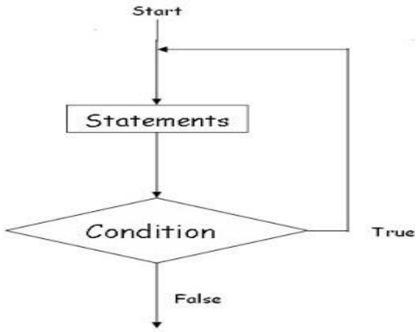
Initialization , Incrementation and Condition steps are on different Line.

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:

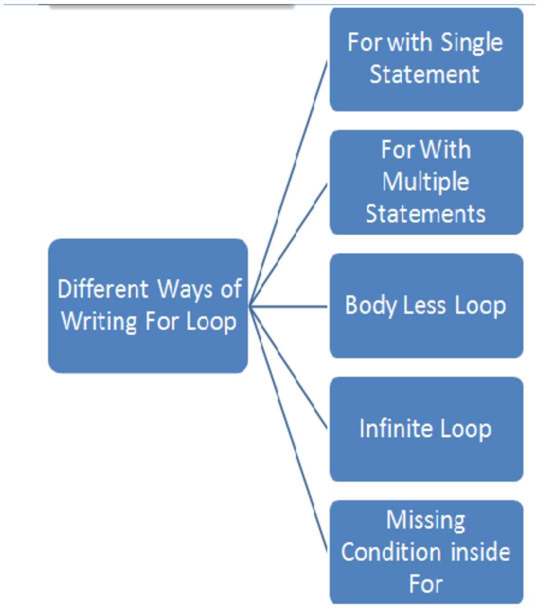
We have already seen the basics of Looping Statement in C. C Language provides us different kind of looping statements such as For loop, while loop and do-while loop. In this chapter we will be learning different flavors of for loop statement.

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
For with Single Statement
For With Multiple Statements
Body Less Loop
Infinite Loop
Missing Condition inside For

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++)
Statement1;
	Single Statement

for (i=0 ;i <10; i++)
{
Statement1;
Statement2;
Statement3;
}
	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++) 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.

<u>Break in For Loop :</u>

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

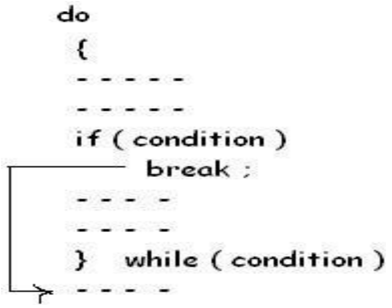
<u>Break in While Loop :</u>

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

<u>Break Statement in Do-While :</u>

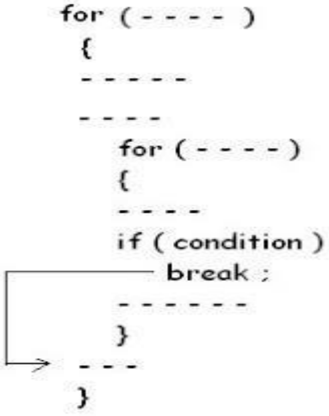
```
initialization ;
do
{
Statement1;
Statement2;
incrementation
break;
}while(condition);
```

Way 1 : Do-While Loop



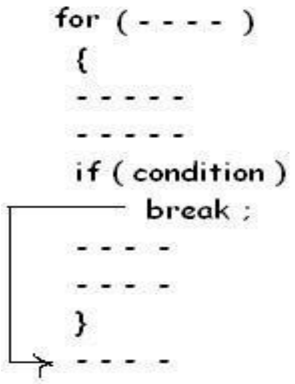
```
do
 {
 - - - - -
 - - - - -
 if ( condition )
      break ;
 - - - -
 - - - -
 }   while ( condition )
 - - - -
```

Way 2 : Nested for



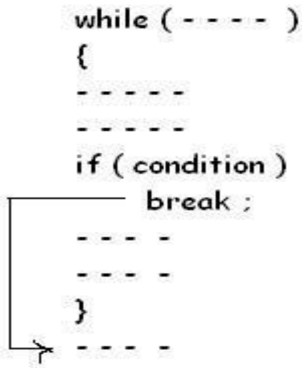
```
for ( - - - - )
 {
 - - - - -
 - - - -
    for ( - - - - )
    {
    - - - -
    if ( condition )
        break ;
    - - - - - -
    }
  - - -
 }
```

Way 3 : For Loop



```
for ( - - - - )
 {
 - - - - -
 - - - - -
 if ( condition )
      break ;
 - - - -
 - - - -
 }
 - - - -
```

Way 4 : While Loop



```
while ( - - - -  )
{
- - - - -
- - - - -
if ( condition )
   break ;
- - -  -
- - -  -
}
- - -  -
```

Continue statement:

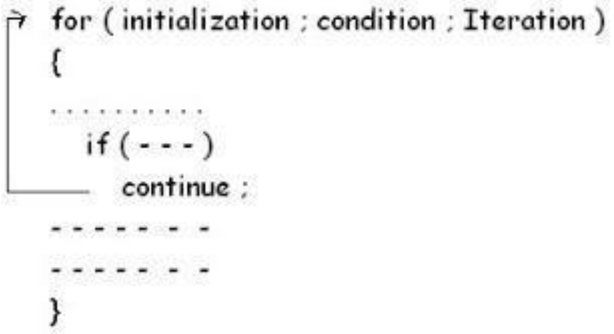
```
loop
{
  continue;
  //code
}
```

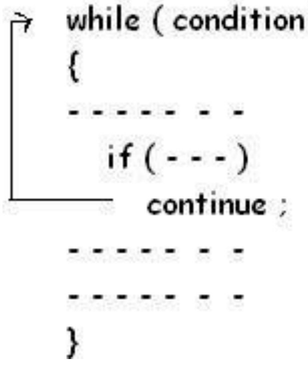
Note :

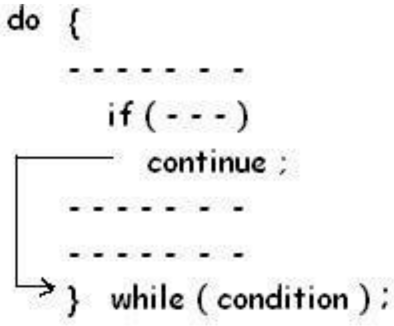
It is used for skipping part of Loop.

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	
	for (initialization ; condition ; Iteration)
	{
	. . . .
	  if (- - -)
	    continue ;
	- - - - - - -
	- - - - - - -
	}

while	
	while (condition)
	{
	- - - - - - - -
	if (- - -)
	continue ;
	- - - - - - - -
	- - - - - - - -
	}

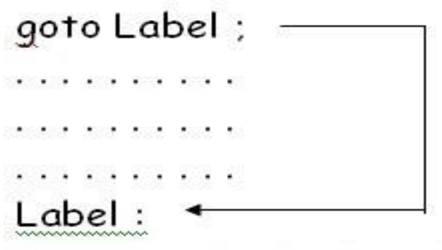
do-while	 do { - - - - - - - - if (- - -) continue ; - - - - - - - - - - - - - - - - } while (condition) ;

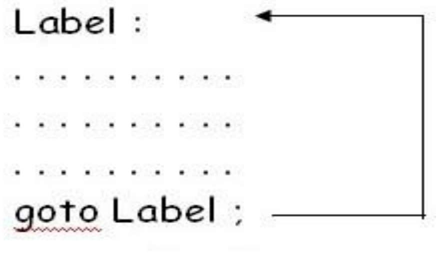
Goto statement:

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 .

<u>Types of Goto</u>
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 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 StorageClass

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 StorageClass

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

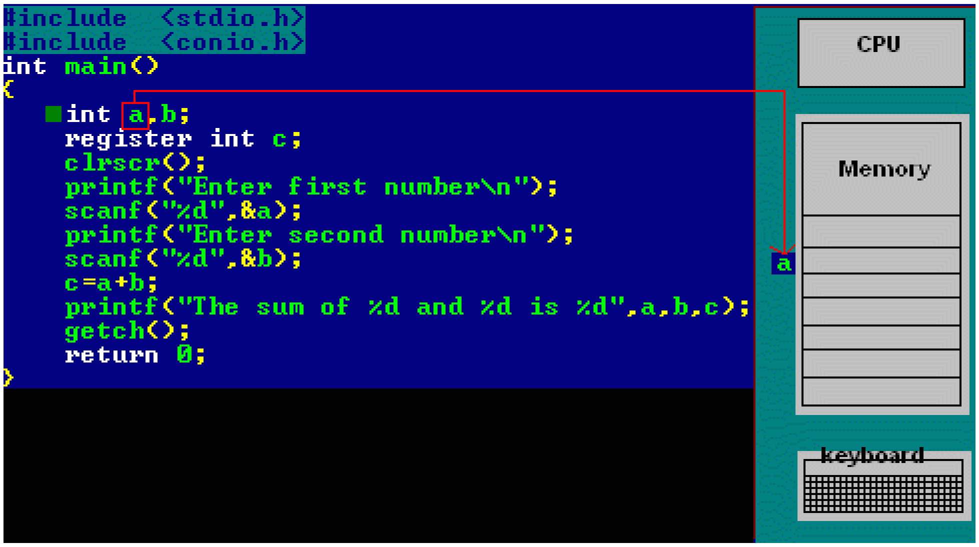
```
{
register int count;
}
```

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 –



```
#include  <stdio.h>
#include  <conio.h>
int main()
{
    int a,b;
    register int c;
    clrscr();
    printf("Enter first number\n");
    scanf("%d",&a);
    printf("Enter second number\n");
    scanf("%d",&b);
    c=a+b;
    printf("The sum of %d and %d is %d",a,b,c);
    getch();
    return 0;
}
```

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:

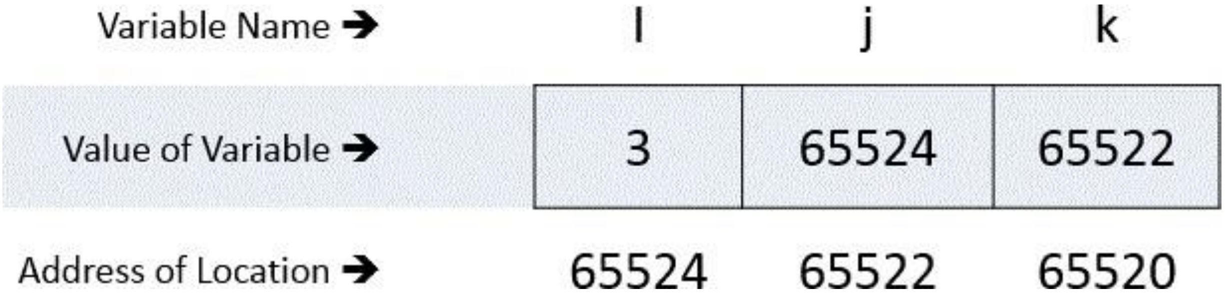
```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```

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



Variable Name ➔	I	j	k
Value of Variable ➔	3	65524	65522
Address of Location ➔	65524	65522	65520

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

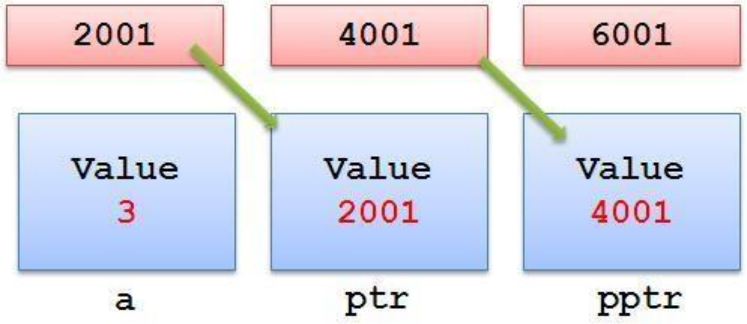
Point	Variable 'a'	Variable 'ptr'
Type of Value that it holds	Integer	Address of Integer 'a'
Value Stored	3	2001
Address of Variable	2001 (Assumption)	4001 (Assumption)

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);
```

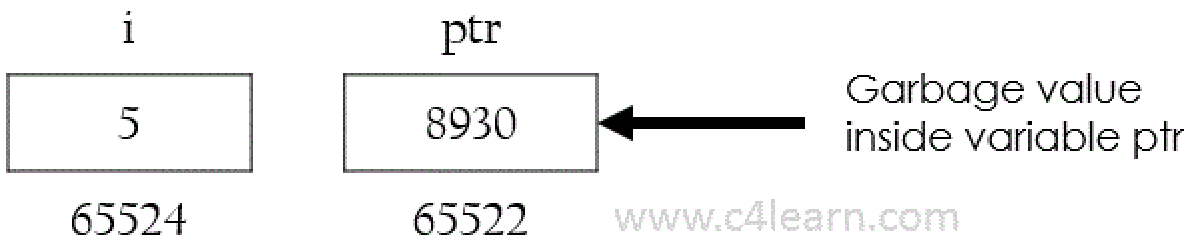
Understanding address operator

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);
}
```

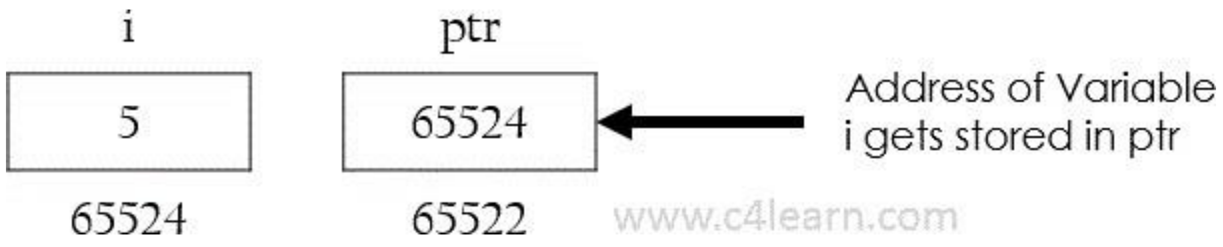
After declaration memory map will be like this –

```
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

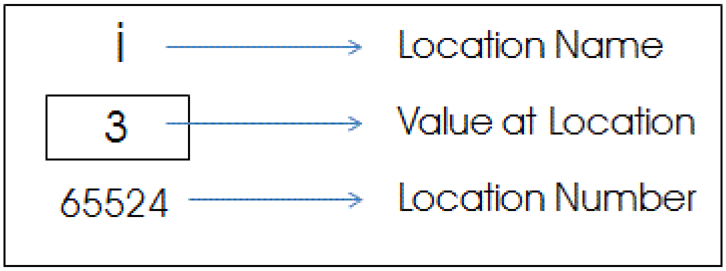
&(a+b)

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)

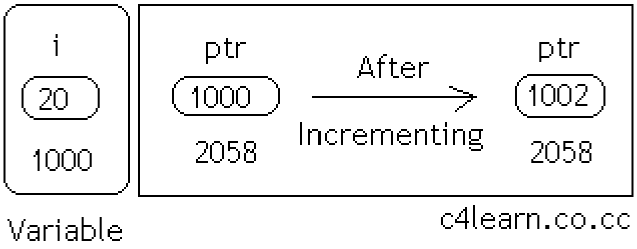
Three Rules should be used to increment pointer –

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 :

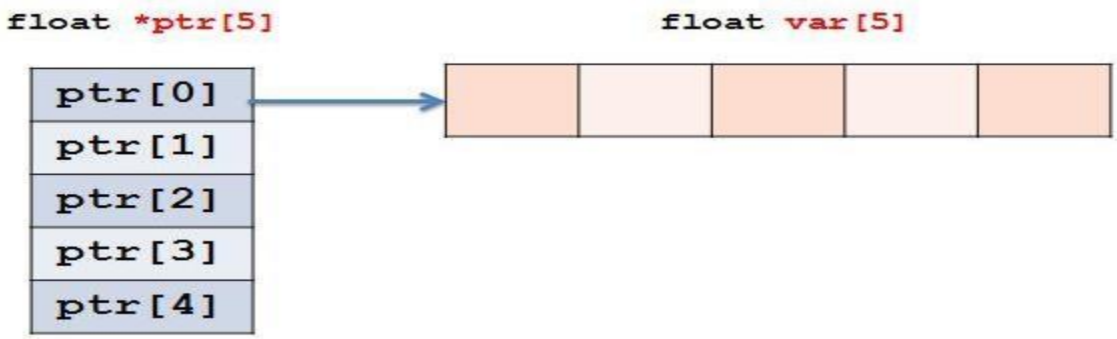
```
New Value of ptr : 1004
```

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

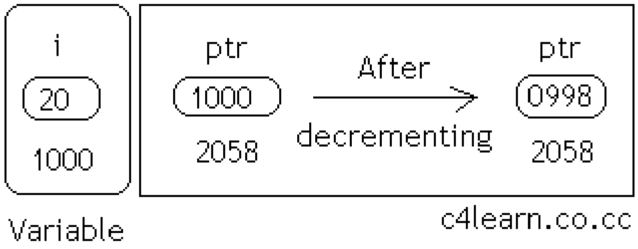
Address of var[4] = 1016

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 :

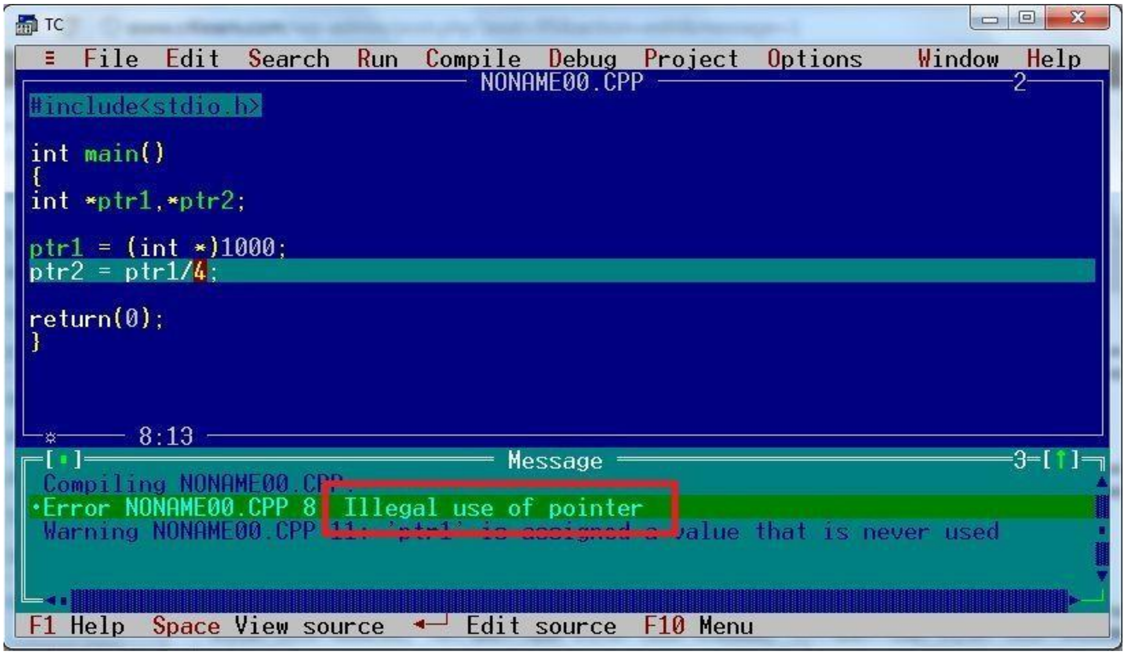
>	Greater Than
<	Less Than
>=	Greater Than And Equal To
<=	Less Than And Equal To
==	Equals
!=	Not Equal

Divide and Multiply Operations :

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

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

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

return(0);
}
```

Output :



TC
≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.CPP
2
8:13
[■]
Message
3=[↑]
F1 Help Space View source ←┘ Edit source F10 Menu

```
#include<stdio.h>

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

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

return(0);
}
```

```
 Compiling NONAME00.CPP:
•Error NONAME00.CPP 8: Illegal use of pointer
 Warning NONAME00.CPP 11: 'ptr1' is assigned a value that is never used
```

Pointer to pointer

Pointer to Pointer in C Programming

Declaration : Double Pointer

int **ptr2ptr;

Consider the Following Example :

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

ptr = &num;

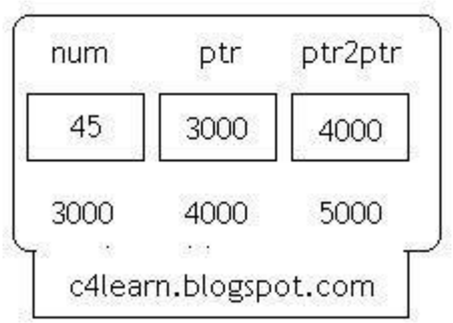
ptr2ptr = &ptr;

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 –

<u>Way 1 : Declare and Initialize</u>

```
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};
```

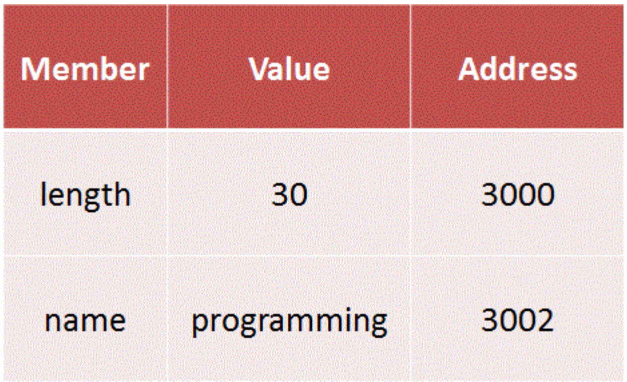
Some Structure Declarations and It‘s Meaning :

```
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 ―Priteshǁ 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];
```

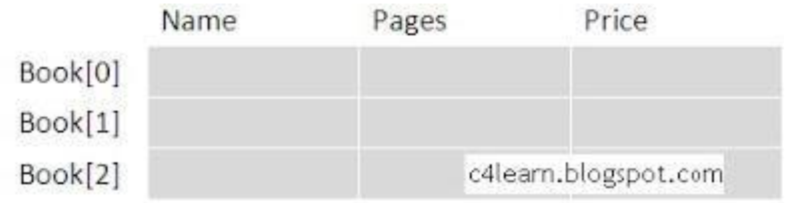
Explanation :

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.

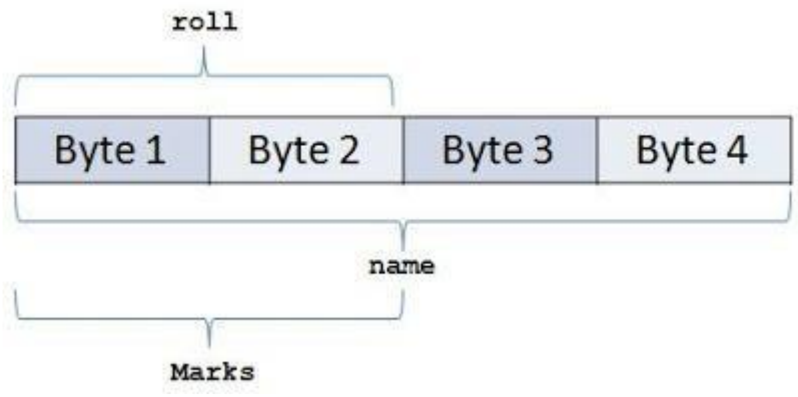
Note #2 : Multiple Members are Collected Together Under Same Name

```
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 :

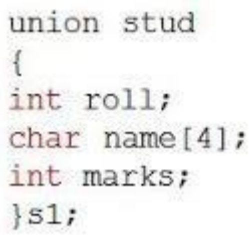
```
union tag

{

   union_member1;
   union_member2;

   union_member3;

   ..

   ..

   ..

   union_memberN;

}instance;
```

Note :

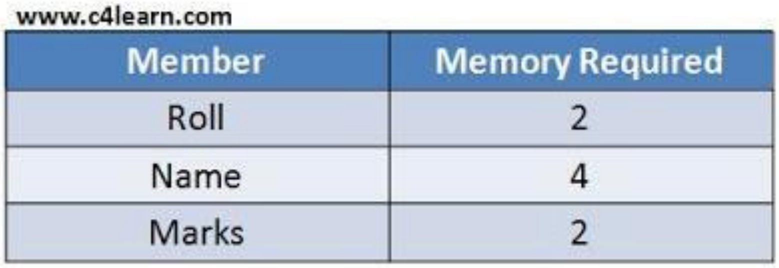
Unions are Declared in the same way as a Structure.Only **—struct Keyword‖** is replaced with **union**

Sample Declaration of Union :

```
union stud

{

   int roll;

   char name[4];

   int marks;

}s1;<
```



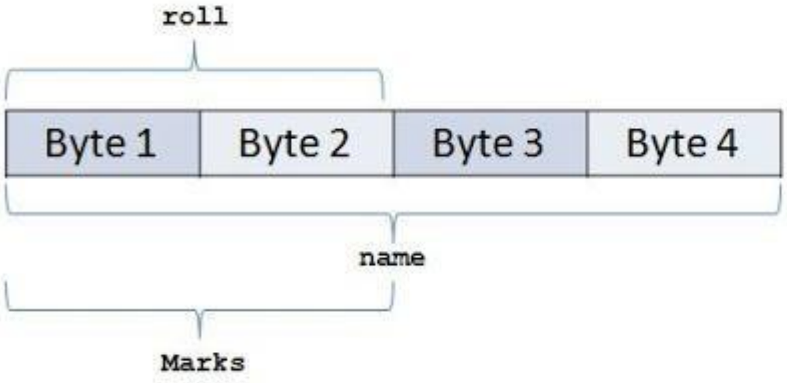
```
union stud
{
int roll;
char name[4];
int marks;
}s1;
```



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;

}
```

<u>Output :</u>

```
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

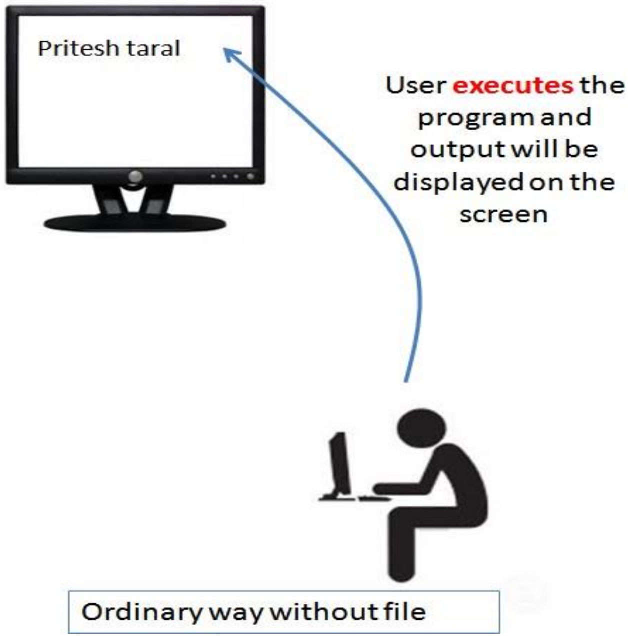
DRAWBACKS OF TRADITIONAL I/O SYSTEM

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)


Pritesh taral
User executes the program and output will be displayed on the screen
Ordinary way without file

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)

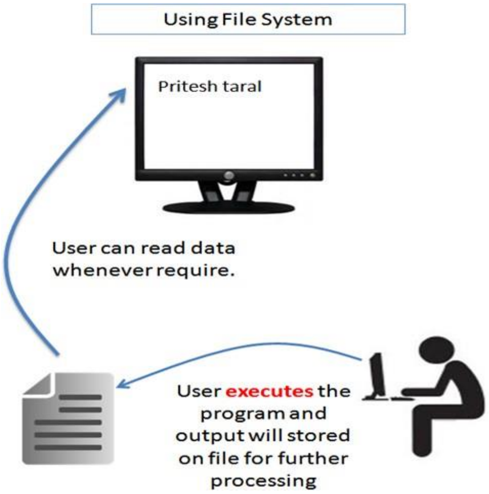
Collection of byte may be interpreted as –

Single character

Single Word

Single Line

Complete Structure.


Using File System
Pritesh taral
User can read data whenever require.
User executes the program and output will stored on file for further processing

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

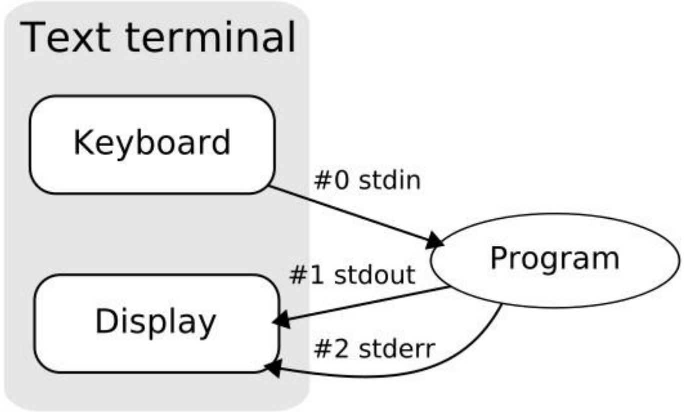
A sequence of bytes flowing out of the program is called output stream

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

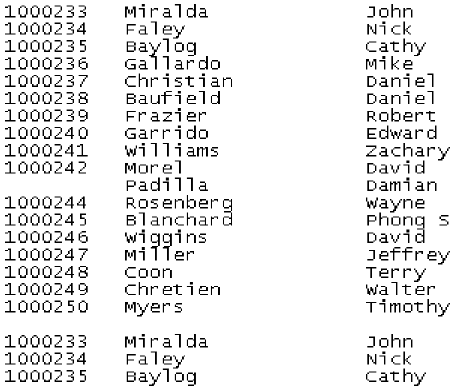
Text file Format in C Programming

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 FileFormats

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.

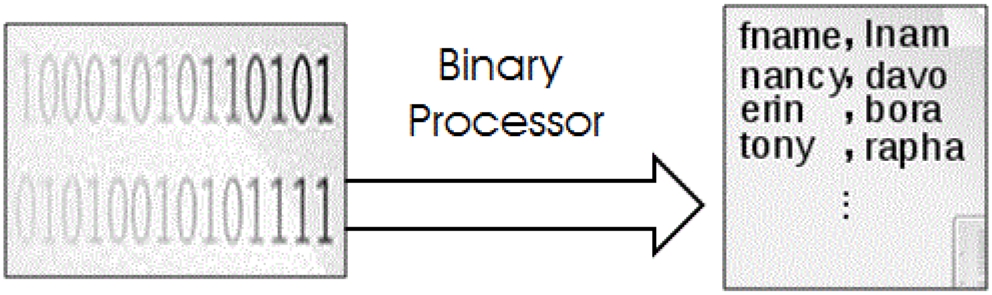
Binary Files are **difficult to read for human**.

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 –

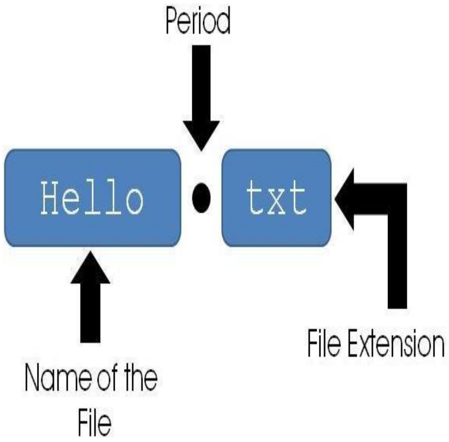
File name

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)
```

COMPUTER PROGRAMMING Using C

LECTURE NOTES

Branch : Computer Science & Engineering

Prepared by : Prashant Singh Yaav

Lecturer Computer Science & Engineering

MAHAMYA POLYTECHNIC OF INFORMATION TECHNOLOGY,SALEMPUR HATHRAS

Lecture Note: 1

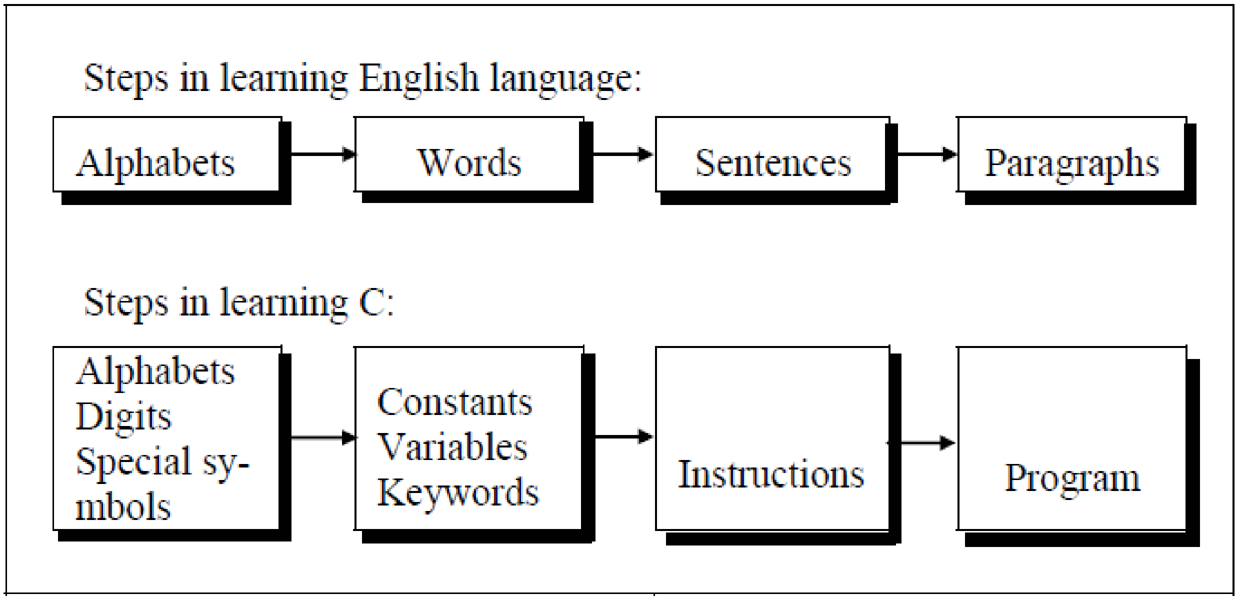
Introduction to **C**

C is a programming language developed at AT & T’s Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called ***Programming in C***, and the title that covered ANSI C was called ***Programming in ANSI C***. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is **reliable**, **simple** and **easy** to use. often heard today is – “C has been already superceded by languages like C++, C# and Java.

Program

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an **instruction**. A group of instructions would be combined later on to form a **program**. So



a computer ***program*** is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's ***instruction set.*** And the approach or method that is used to solve the problem is known as an ***algorithm***.

So for as programming language concern these are of two types.

1) Low level language

2) High level language

Low level language:

Low level languages are **machine level** and **assembly level language**. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The **assembly language** is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

High level language:

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Three types of translator are there:

Compiler

Interpreter

Assembler

Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

Integrated Development Environments (IDE)

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows us to easily manage large software programs, edit files in windows, and compile, link, run, and debug programs.

On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++.

<u>Lecture Note: 2</u>

Structure of C Language program

1) Comment line

2) Preprocessor directive

3) Global variable declaration

4) main function( )

```
        {
          Local variables;
  Statements;
  }
  User defined function
 }
}
```

Comment line

It indicates the purpose of the program. It is represented as

/*……………………………..*/

Comment line is used for increasing the readability of the program. It is useful in explaining the program and generally used for documentation. It is enclosed within the decimeters. Comment line can be single or multiple line but should not be nested. It can be anywhere in the program except inside string constant & character constant.

Preprocessor Directive:

#include<stdio.h> tells the compiler to include information about the standard input/output library. It is also used in symbolic constant such as #define PI 3.14(value). The stdio.h (standard input output header file) contains definition &declaration of system defined function such as printf( ), scanf( ), pow( ) etc. Generally printf() function used to display and scanf() function used to read value

Global Declaration:

This is the section where variable are declared globally so that it can be access by all the functions used in the program. And it is generally declared outside the function :

main()

It is the user defined function and every function has one main() function from where actually program is started and it is encloses within the pair of curly braces.

The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax :

```
  main()
{
……..
……..
……..
}
```

The main( ) function return value when it declared by data type as

```
 int main( )
{
return 0
}
```

The main function does not return any value when void (means null/empty) as

```
void main(void ) or void main()

{

 printf (“C language”);

}
```

Output: C language

The program execution start with opening braces and end with closing brace.

And in between the two braces declaration part as well as executable part is mentioned. And at the end of each line, the semi-colon is given which indicates statement termination.

/*First c program with return statement*/

```
#include <stdio.h>

int main (void)

{

printf ("welcome to c Programming language.\n");

return 0;

}
```

Output: welcome to c programming language.

Steps for Compiling and executing the Programs

A compiler is a software program that analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on a particular computer system. Figure below shows the steps that are involved in entering, compiling, and executing a

computer program developed in the C programming language and the typical Unix commands that would be entered from the command line.

Step 1: The program that is to be compiled is first typed into a *file* on the computer system. There are various conventions that are used for naming files, typically be any name provided the last two characters are **".c"** or file with extension .c. So, the file name **prog1.c** might be a valid filename for a C program. A text editor is usually used to enter the C program into a file. For example, vi is a popular text editor used on Unix systems. The program that is entered into the file is known as the ***source program*** because it represents the original form of the program expressed in the C language.

Step 2: After the source program has been entered into a file, then proceed to have it compiled. The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under Unix, the command to initiate program compilation is called **cc**. If we are using the popular GNU C compiler, the command we use is **gcc**.

Typing the line

```
gcc prog1.c or cc prog1.c
```

In the first step of the compilation process, the compiler examines each program

statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, they are reported to the user and the compilation process ends right there. The errors then have to be corrected in the source program (with the use of an editor), and the compilation process must be restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (**syntactic error**), or due to the use of a variable that is not "defined" (**semantic error**).

Step 3: When all the syntactic and semantic errors have been removed from the program, the compiler then proceeds to take each statement of the program and translate it into a “lower” form that is equivalent to assembly language program needed to perform the identical task.

Step 4: After the program has been translated the next step in the compilation process is to translate the assembly language statements into actual machine instructions. The assembler takes each assembly language statement and converts it into a binary format known as ***object code***, which is then written into another file on the system. This file has the same name as the source file under Unix, with the last letter an “**o**” (**for *object***) instead of a “**c**”.

Step 5: After the program has been translated into object code, it is ready to be ***linked.*** This process is once again performed automatically whenever the cc or gcc command is issued under Unix. The purpose of the linking phase is to get the program into a final form for execution on the computer.

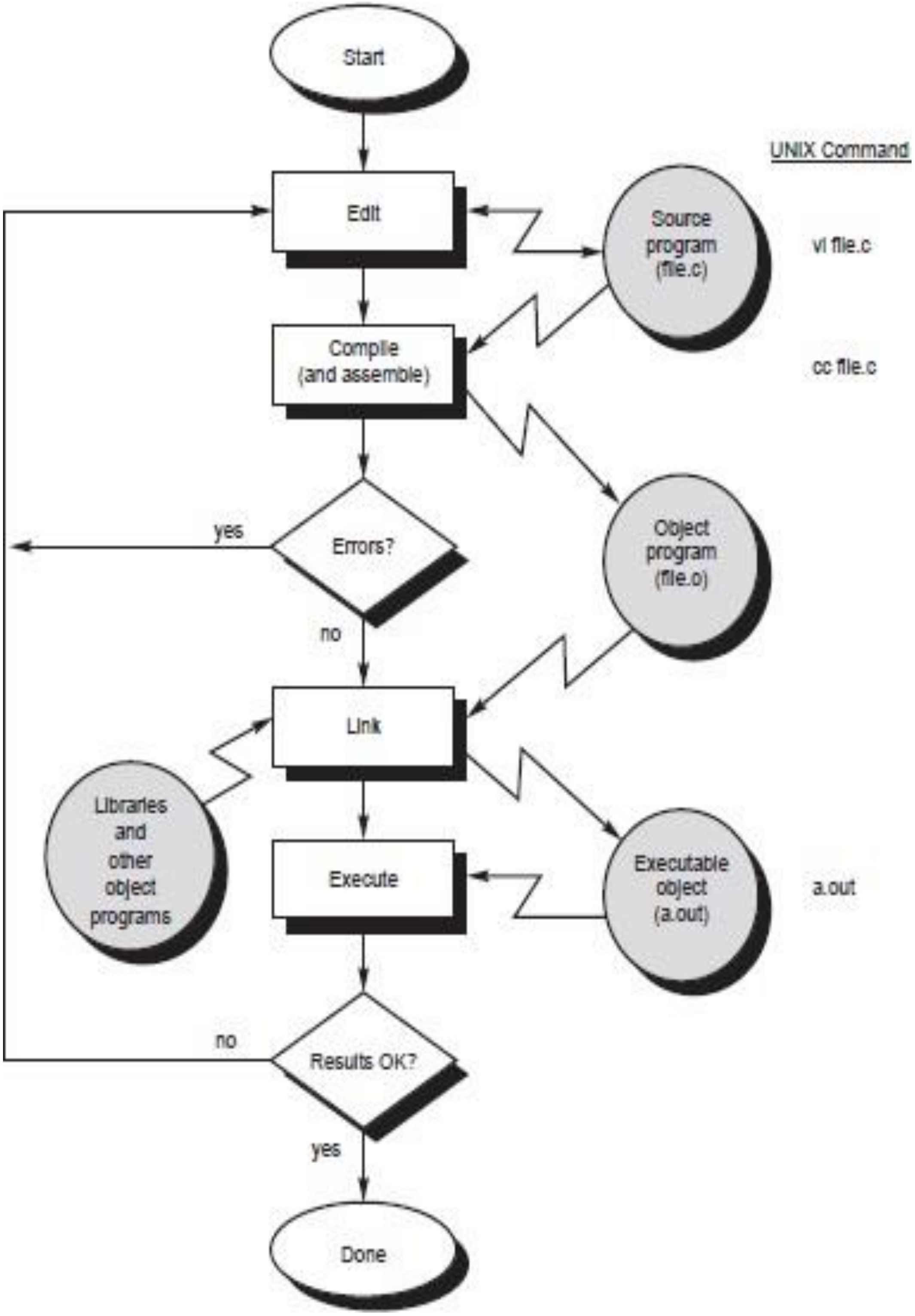
If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system’s program ***library*** are also searched and linked together with the object program during this phase.

The process of compiling and linking a program is often called ***building.***

The final linked file, which is in an executable ***object*** code format, is stored in another file on the system, ready to be run or ***executed.*** Under Unix, this file is called **a.out** by default. Under Windows, the executable file usually has the same name as the source file, with the c extension replaced by an exe extension.

Step 6: To subsequently execute the program, the command **a.out** has the effect of *loading* the program called **a.out** into the computer's memory and initiating its execution.

When the program is executed, each of the statements of the program is sequentially executed in turn. If the program requests any data from the user, known as ***input***, the program temporarily suspends its execution so that the input can be entered. Or, the program might simply wait for an ***event***, such as a mouse being clicked, to occur. Results that are displayed by the program, known as ***output,*** appear in a window, sometimes called the ***console***. If the program does not produce the desired results, it is necessary to go back and reanalyze the program's logic. This is known as the ***debugging phase***, during which an attempt is made to remove all the known problems or ***bugs*** from the program. To do this, it will most likely be necessary to make changes to original source program.



/* Simple program to add two numbers…………………….*/

```
#include <stdio.h>

int main (void)

{

int v1, v2, sum;                    //v1,v2,sum are variables and int is data type declared

v1 = 150;

v2 = 25;

sum = v1 + v2;
printf ("The sum of %i and %i is= %i\n", v1, v2, sum);

return 0;

}
```

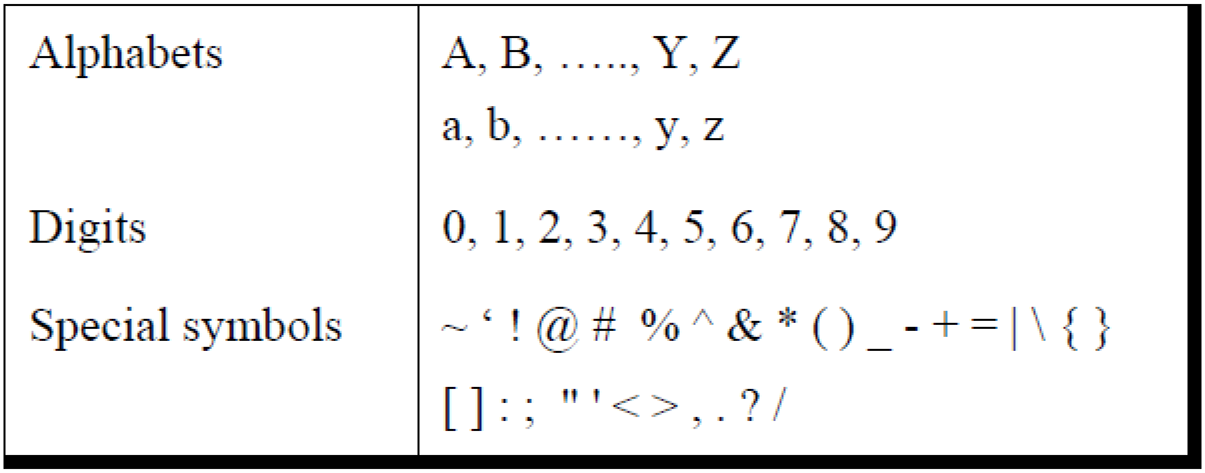
Output:

```
  The sum of 150 and 25 is=175
```

Lecture Note: 3

Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are



<table>
<tr><td>Alphabets</td><td>A, B, ….., Y, Z
a, b, ……, y, z</td></tr>
<tr><td>Digits</td><td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td></tr>
<tr><td>Special symbols</td><td>~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { }
[ ] : ; " ' < > , . ? /</td></tr>
</table>

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1)name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.

2) first characters should be alphabet or underscore

3) name should not be a keyword

4)since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

5) identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

Keyword

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords.** These words are predefined and always written in lower case or small letter. These keywords cann't be used as a variable name as it assigned with fixed meaning. Some examples are **int, short, signed, unsigned, default, volatile, float, long, double, break, continue, typedef, static, do, for, union, return, while, do, extern, register, enum, case, goto, struct, char, auto, const etc.**

data types

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

C has the following 4 types of data types

basic built-in data types: int, float, double, char

Enumeration data type: enum

Derived data type: pointer, array, structure, union

Void data type: void

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly A variable declared char can only store character type value.

There are two types of type qualifier in c

Size qualifier: short, long

Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char	1	-128 to 127
	Unsigned char	1	0 to 255
int	int or signed int	2	-32768 to 32767
	unsigned int	2	0 to 65535
	short int or signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double	8	1.7E-308 to 1.7E+308
	Long double	10	3.4E-4932 to 1.1E+4932

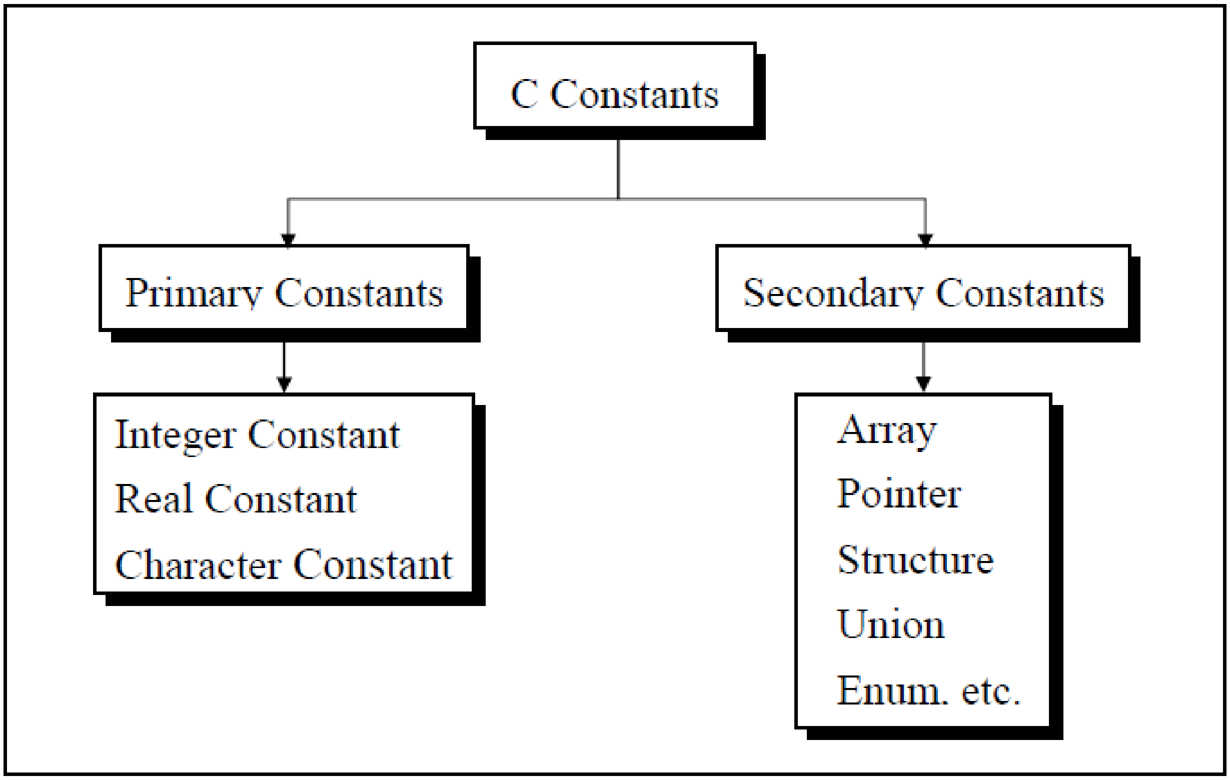
Lecture Note: 4

Constants

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a *constant*. A constant is an entity that doesn’t change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. C constants can be divided into two major categories:

Primary Constants
Secondary Constants

These constants are further categorized as



Numeric constant
Character constant
String constant

Numeric constant: Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant.** An integer constants are whole number which have no decimal point. Types of integer constants are:

Decimal constant: 0-------9(base 10)
Octal constant: 0-------7(base 8)
Hexa decimal constant: 0----9, A------F(base 16)

In decimal constant first digit should not be zero unlike octal constant first digit must be zero(as 076, 0127) and in hexadecimal constant first two digit should be 0x/ 0X (such as 0x24, 0x87A). By default type of integer constant is integer but if the value of integer constant is exceeds range then value represented by integer type is taken to be unsigned integer or long integer. It can also be explicitly mention integer and unsigned integer type by suffix l/L and u/U.

Real constant is also called floating point constant. To construct real constant we must follow the rule of ,

-real constant must have at least one digit.
-It must have a decimal point.
-It could be either positive or negative.
-Default sign is positive.
-No commas or blanks are allowed within a real constant. Ex.: +325.34
426.0
-32.76

To express small/large real constant exponent(scientific) form is used where number is written in mantissa and exponent form separated by e/E. Exponent can be positive or negative integer but mantissa can be real/integer type, for example $3.6*10^5$=3.6e+5. By default type of floating point constant is double, it can also be explicitly defined it by suffix of f/F.

Character constant

Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as ‘9’,’c’,’$’, ‘ ’ etc. Every character constant has a unique integer like value in machine’s character code as if machine using ASCII (American standard code for information interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:

A------------ Z ASCII value (65-90)

a-------------z ASCII value (97-122)

0-------------9 ASCII value (48-59)

; ASCII value (59)

String constant

Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String constant has zero, one or more than one character and at the end of the string null character(\0) is automatically placed by compiler. Some examples are “,sarathina” , “908”, “3”,” ”, “A” etc. In C although same characters are enclosed within single and double quotes it represents different meaning such as “A” and ‘A’ are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

Symbolic constant

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constant are generally defined at the beginning of the program as

#define name value , here name generally written in upper case for example

```
#define MAX 10

#define CH ‘b’

  #define NAME “sony”
```

Variables

Variable is a data name which is used to store some data value or symbolic names for storing program
computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax:

```
  int a;

 char c;

 float f;
```

Variable initialization

When we assign any initial value to variable during the declaration, is called initialization of variables. When variable is declared but contain undefined value then it is called garbage value. The variable is initialized with the assignment operator such as

Data type variable name=constant;

```
Example: int a=20;

         Or int a;

              a=20;
```

statements

Lecture Note: 5

Expressions

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational for example:-

```
int z=  x+y    // arithmatic expression

 a>b      //relational

a==b         // logical
func(a, b)   // function call
```

Expressions consisting entirely of constant values are called *constant expressions*. So, the expression
121 + 17 - 110
is a constant expression because each of the terms of the expression is a constant value. But if i were declared to be an integer variable, the expression
180 + 2 – j
would not represent a constant expression.

Operator

This is a symbol use to perform some operation on variables, operands or with the constant. Some operator required 2 operand to perform operation or Some required single operation.

Several operators are there those are, arithmetic operator, assignment, increment , decrement, logical, conditional, comma, size of , bitwise and others.

1. Arithmatic Operator

This operator used for numeric calculation. These are of either Unary arithmetic operator, Binary arithmetic operator. Where Unary arithmetic operator required only one operand such as +,-, ++, --,!, tiled. And these operators are addition, subtraction, multiplication, division. Binary arithmetic operator on other hand required two operand and its operators are +(addition), -(subtraction), *(multiplication), /(division), %(modulus). But modulus cannot applied with floating point operand as well as there are no exponent operator in c.

Unary (+) and Unary (-) is different from addition and subtraction.

When both the operand are integer then it is called integer arithmetic and the result is always integer. When both the operand are floating point then it is called floating arithmetic and when operand is of integer and floating point then it is called mix type or mixed mode arithmetic . And the result is in float type.

2. Assignment Operator

A value can be stored in a variable with the use of assignment operator. The assignment operator(=) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side should be variable or constant or any expression. When variable on the left hand side is occur on the right hand side then we can avoid by writing the compound statement. For example,

```
  int x= y;

int  Sum=x+y+z;
```

3. Increment and Decrement

The Unary operator ++, --, is used as increment and decrement which acts upon single operand. Increment operator increases the value of variable by one .Similarly decrement operator decrease the value of the variable by one. And these operator can only used with the variable, but cann't   use with expression and constant as ++6 or ++(x+y+z).

It again categories into prefix post fix . In the prefix the value of the variable is incremented 1st, then the new value is used, where as in postfix the operator is written after the operand(such as m++,m--).

EXAMPLE

let y=12;

z= ++y;

y= y+1;

z= y;

Similarly in the postfix increment and decrement operator is used in the operation . And then increment and decrement is perform.

EXAMPLE

let x= 5;

y= x++;

y=x;

x= x+1;

4.Relational Operator

It is use to compared value of two expressions depending on their relation. Expression that contain relational operator is called relational expression.

Here the value is assign according to true or false value.

a.(a>=b) || (b>20)

b.(b>a) && (e>b)

c. 0(b!=7)

5. Conditional Operator

It sometimes called as ternary operator. Since it required three expressions as operand and it is represented as (? , :).

SYNTAX

```
exp1 ? exp2 :exp3
```

Here exp1 is first evaluated. It is true then value return will be exp2 . If false then exp3.

EXAMPLE

```
void main()
{
 int a=10, b=2
  int s= (a>b) ? a:b;
  printf(“value is:%d”);
 }
```

Output:

```
    Value is:10
```

6. Comma Operator

Comma operator is use to permit different expression to be appear in a situation where only one expression would be used. All the expression are separator by comma and are evaluated from left to right.

EXAMPLE

```
int i, j, k, l;
for(i=1,j=2;i<=5;j<=10;i++;j++)
```

7. Sizeof Operator

Size of operator is a Unary operator, which gives size of operand in terms of byte that occupied in the memory. An operand may be variable, constant or data type qualifier.

Generally it is used make portable program(program that can be run on different machine) . It determines the length of entities, arrays and structures when their size are not known to the programmer. It is also use to allocate size of memory dynamically during execution of the program.

EXAMPLE

```
main( )

{

int sum;

float f;

printf( "%d%d" ,size of(f), size of (sum) );

printf("%d%d", size of(235 L), size of(A));

}
```

Lecture Note: 6

8. Bitwise Operator

Bitwise operator permit programmer to access and manipulate of data at bit level. Various bitwise operator enlisted are

one's complement	(~)
bitwise AND	(&)
bitwise OR	(|)
bitwise XOR	(^)
left shift	(<<)
right shift	(>>)

These operator can operate on integer and character value but not on float and double. In bitwise operator the function showbits( ) function is used to display the binary representation of any integer or character value.

In one's complement all 0 changes to 1 and all 1 changes to 0. In the bitwise OR its value would obtaining by 0 to 2 bits.

As the bitwise OR operator is used to set on a particular bit in a number. Bitwise AND the logical AND.

It operate on 2operands and operands are compared on bit by bit basic. And hence both the operands are of same type.

Logical or Boolean Operator

Operator used with one or more operand and return either value zero (for false) or one (for true). The operand may be constant, variables or expressions. And the expression that combines two or more expressions is termed as logical expression. C has three logical operators :

Operator	**Meaning**
&&	AND
||	OR
!	NOT

Where logical NOT is a unary operator and other two are binary operator. Logical AND gives result true if both the conditions are true, otherwise result is false. And logial OR gives result false if both the condition false, otherwise result is true.

Precedence and associativity of operators

Operators	**Description**	**Precedence level**	**Associativity**
()	function call	**1**	left to right
[]	array subscript		
→	arrow operator		
.	dot operator		
+	unary plus	2	right to left
-	unary minus		
++	increment		
- -	decrement		
!	logical not		
~	1’s complement		
*****	indirection		
&	address		
(data type)	type cast		
sizeof	size in byte		
*****	multiplication	3	left to right
/	division		
%	modulus		
+	addition	4	left to right

-	subtraction		
<<	left shift	**5**	left to right
>>	right shift		
<=	less than equal to	6	left to right
>=	greater than equal to		
<	less than		
>	greater than		
==	equal to	7	left to right
!=	not equal to		
&	bitwise AND	8	left to right
^	bitwise XOR	9	left to right
|	bitwise OR	10	left to right
&&	logical AND	11	
||	logical OR	12	
?:	conditional operator	13	
=, *=, /=, %= **&=, ^=, <<=** **>>=**	assignment operator	14	right to left
,	comma operator	15	

Lecture Note: 7

Control Statement

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program, that is called control statement. Control statement defined how the control is transferred from one part to the other part of the program. There are several control statement like if...else, switch, while, do....while, for loop, break, continue, goto etc.

Loops in C

Loop:-it is a block of statement that performs set of instructions. In loops

Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

There are three types of loops in c

1.While loop

2.do while loop

3.for loop

While loop

Syntax:-

```
while(condition)

{

Statement 1;

Statement 2;

}
Or      while(test condition)

        Statement;
```

The test condition may be any expression .when we want to do something a fixed no of times but not known about the number of iteration, in a program then while loop is used.

Here first condition is checked if,   it is true body of the loop is executed else, If condition is false control will be come out of loop.

Example:-

```
/* wap to print 5 times welcome to C” */
#include<stdio.h>
void main()
{
int p=1;
While(p<=5)
{
printf(“Welcome to C\n”);
P=p+1;
}
}
```

```
Output: Welcome to C
                 Welcome to C
                Welcome to C
                Welcome to C
                Welcome to C
```

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

do while loop

This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Syntax:-

```
Do

{

Statement;

}

while(condition);
```

Example:-

```
#include<stdio.h>

void main()

{

int X=4;

 do

{

 Printf("%d",X);

X=X+1;
```

```
  }whie(X<=10);
   Printf(“ ”);
}
```

Output: 4 5 6 7 8 9 10

Here firstly statement inside body is executed then condition is checked. If the condition is true again body of loop is executed and this process continue until the condition becomes false. Unlike while loop semicolon is placed at the end of while.

There is minor difference between while and do while loop, while loop test the condition before executing any of the statement of loop. Whereas do while loop test condition after having executed the statement at least one within the loop.

If initial condition is false while loop would not executed it’s statement on other hand do while loop executed it’s statement at least once even If condition fails for first time. It means do while loop always executes at least once. **Notes:**

Do while loop used rarely when we want to execute a loop at least once.

<u>Lecture Note: 8</u>

for loop

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)

{

Statement;

}
```

Or

```
for(initialized counter; test counter; update counter)

{

Statement;

}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Example:-

```
void main()

{
int i;

for(i=1;i<10;i++)

{
 Printf(“ %d ”, i);

  }

}
```

Output:-1 2 3   4 5 6 7 8 9

Nesting of loop

When a loop written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

```
void main()

{

int i,j;

for(i=0;i<2;i++)

for(j=0;j<5; j++)

printf(“%d %d”, i, j);

 }
```

```
Output: i=0

     j=0 1 2 3 4

      i=1

     j=0 1 2 3 4
```

Break statement(break)

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in which it appears and it is generally written with condition. It is written with the keyword as **break.** When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to the first statement after the loop. This break statement is usually associated with **if** statement.

Example :

```
void main()

{

int j=0;

for(;j<6;j++)

if(j==4)

break;

}
```

Output:

```
0 1 2 3
```

Continue statement (key word continue)

Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through the beginning of the loop. It is usually associated with the if statement. It is useful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

Example:-

```
void main()

{

int n;

for(n=2; n<=9; n++)

{

if(n==4)

continue;

printf(“%d”, n);

  }

}

Printf(“out of loop”);

}
```

Output: 2 3 5 6 7 8 9 out of loop

if statement

Statement execute set of command like when condition is true and its syntax is

```
        If (condition)

    Statement;
```

The statement is executed only when condition is true. If the if statement body is consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

```
void main()

{

    int n;
     printf (“ enter a number:”);

      scanf(“%d”,&n);

         If (n>10)

        Printf(“ number is grater”);

     }
```

Output:

```
          Enter a number:12

          Number is greater
```

if…..else ... Statement

it is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded as true & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

Its syntax is:-

```
if (condition)
 {
 Statement1;
  Statement2;
 }
   else
     {
      Statement1;
      Statement2;
       }
```

Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be a if statement with in an else statement.

Example:-

/* To check a number is eve or odd */

```
 void main()
{
  int n;
      printf (“enter a number:”);
     sacnf (“%d”, &n);
      If (n%2==0)
         printf (“even number”);
else
         printf(“odd number”);
}
```

```
Output: enter a number:121
odd number
```

Lecture Note: 10

Nesting of if …else

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```
                 if (condition)
                  {
```

```
        If (condition)
         Statement1;
    else
           statement2;
      }
            Statement3;
```

If….else LADDER

In this type of nesting there is an if else statement in every else part except the last part. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is :-

```
      if (condition)
       Statement1;
   else if (condition)
         statement2;
   else if (condition)
        statement3;
    else
           statement4;
```

This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested "else if" would not executed.

But it has disadvantage over if else statement that, in if else statement whenever the condition is true, other condition are not checked. While in this case, all condition are checked.

Lecture Note: 11

ARRAY

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
int  arr[100];
```

int mark[100];

DECLARATION OF AN ARRAY :

Its syntax is :

Data type array name [size];

```
int arr[100];

int mark[100];

    int a[5]={10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

```
int ar[5];

 ar[0], ar[1], ar[2], ar[3], ar[4];
```

Symbolic constant can also be used to specify the size of the array as:

```
#define SIZE 10;
```

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3…}

Example:

```
in ar[5]={20,60,90, 100,120}
```

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

So if i & j are not variable then the valid subscript are
ar [i*7],ar[i*i],ar[i++],ar[3];

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

```
/*Write a program to input values into an array and display them*/

#include<stdio.h>

int main()

{

int arr[5],i;

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

{

printf("enter a value for arr[%d] \n",i);

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

}
printf("the array elements are: \n");

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

{

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

}

return 0;
}
```

OUTPUT:

```
Enter a value for arr[0] = 12

Enter a value for arr[1] =45

Enter a value for arr[2] =59

Enter a value for arr[3] =98

Enter a value for arr[4]  =21

The array elements are 12 45 59 98  21
```

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21

ar[0]	ar[1]	ar[2]	ar[3]	ar[4]
12	45	59	98	21
2000	2002	2004	2006	2008

Example 2:

```
    /* Write a program to add 10 array elements */
     #include<stdio.h>
      void main()
{
 int i ;
int arr [10];
int sum=o;
for (i=0; i<=9; i++)
{
printf (“enter the %d element \n”, i+1);
scanf (“%d”, &arr[i]);
}
for (i=0; i<=9; i++)
{
sum = sum + a[i];
}
printf (“the sum of 10 array elements is %d”, sum);
}
```

OUTPUT:

```
Enter a value for arr[0] =5
Enter a value for arr[1] =10
Enter a value for arr[2] =15
Enter a value for arr[3]  =20
Enter a value for arr[4] =25
Enter a value for arr[5] =30
Enter a value for arr[6] =35
Enter a value for arr[7] =40
Enter a value for arr[8] =45
Enter a value for arr[9] =50
Sum = 275
```

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example:-

```
int  marks[]={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero .

For example:-

```
int marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0

Again if we initialize an array like

```
int array[100]={0};
```

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-

```
 int arr[5]={1,2,3,4,5,6,7,8};//error
```

we cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as

```
     int a[5] ={1,2,3,4,5};
   int b[5];
   b=a;//not valid
```

(**note**:-here we will have to copy all the elements of array one by one, using for loop.)

Single dimensional arrays and functions

```
/*program to pass array elements to a function*/
#include<stdio.h>
void main()
{
int arr[10],i;
printf(“enter the array elements\n”);
for(i=0;i<10;i++)
{
scanf(“%d”,&arr[i]);
check(arr[i]);
}
}
```

```
void check(int num)

{

if(num%2=0)

{

printf(”%d is even \n”,num);

}

else

{

printf(”%d is odd \n”,num);

}

}
```

Lecture Note: 12

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example:-

int a[2][3];

Total no of elements=row*column is 2*3 =6

It means the matrix consist of 2 rows and 3 columns

For example:-

20	2	7
8	3	15

Positions of 2-D array elements in an array are as below

00	01	02
10	11	12

a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]
20	2	7	8	3	15
2000	2002	2004	2006	2008	

Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

int a[4][5];

for reading value:-

```
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
```

For displaying value:-

```
    for(i=0;i<4;i++)
    {
    for(j=0;j<5;j++)
        {
            printf("%d",a[i][j]);
        }
    }
```

Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array. for example:-

int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

```
    Mat[0][0]=11,     Mat[1][0]=14,     Mat[2][0]=17      Mat[3][0]=20
    Mat[0][1]=12,   Mat[1][1]=15,   Mat[2][1]=18      Mat[3][1]=21
    Mat[0][2]=13,     Mat[1][2]=16,     Mat[2][2]=19     Mat[3][2]=22
```

While initializing we can group the elements row wise using inner braces.

for example:-

int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};

And while initializing , it is necessary to mention the 2nd dimension where 1st dimension is optional.

int mat[][3];

int mat[2][3];

int mat[][];
int mat[2][]; } invalid

If we **initialize an array** as

int mat[4][3]={{11},{12,13},{14,15,16},{17}};

Then the compiler will assume its all rest value as 0,which are not defined.

Mat[0][0]=11, Mat[1][0]=12, Mat[2][0]=14, Mat[3][0]=17

Mat[0][1]=0, Mat[1][1]=13, Mat[2][1]=15 Mat[3][1]=0

Mat[0][2]=0, Mat[1][2]=0, Mat[2][2]=16, Mat[3][2]=0

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic constant

Such as

#define ROW 2;

```
#define COLUMN 3;

int mat[ROW][COLUMN];
```

String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

```
char name[]={‘j’,’o’,’h’,’n’,’\o’};
```

Here each character occupies 1 byte of memory and last character is always NULL character. Where ’\o’ and 0 (zero) are not same, where **ASCII** value of ‘\o’ is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

J	o	h	N	‘\o’

The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as;

```
char name[]=”John”;
```

Here the NULL character is not necessary and the compiler will assume it automatically.

String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character (‘\o’).

Example – “m”

“Tajmahal”

“My age is %d and height is %f\n”

The string constant itself becomes a pointer to the first character in array.

Example-char crr[20]=”Taj mahal”;

1000	1001	1002	1003	1004	1005	1006	1007	100	1009
T	a	j		M	A	H	a	l	\o

It is called base address.

Lecture Note: 13

String library function

There are several string library functions used to manipulate string and the prototypes for these functions are in header file “string.h”. Several string functions are

strlen()

This function return the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string.

For example-

```
    strlen(“suresh”);
```

It return the value 6.

In array version to calculate legnth:-

```
    int str(char str[])
{
    int i=0;
    while(str[i]!=’\o’)
    {
        i++;
    }
    return i;
}
```

Example:-

```
    #include<stdio.h>
    #include<string.h>
    void main()
{
    char str[50];
    print(”Enter a string:”);
    gets(str);

    printf(“Length of the string is %d\n”,strlen(str));

}
```

Output:

```
Enter a string: C in Depth

Length of the string is 8
```

strcmp()

This function is used to compare two strings. If the two string match, strcmp() return a value 0 otherwise it return a non-zero value. It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

```
    strcmp(s1,s2)

    return a value:

        <0 when s1<s2

        =0 when s1=s2

        >0 when s1>s2
```

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

For example:

```
/*String comparison……………………….*/
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[10],str2[10];
        printf(“Enter two strings:”);
        gets(str1);
        gets(str2);
        if(strcmp(str1,str2)==0)
{
        printf(“String are same\n”);
}
else
{
        printf(“String are not same\n”);
}
}
```

strcpy()

This function is used to copying one string to another string. The function strcpy(str1,str2) copies str2 to str1 including the NULL character. Here str2 is the source string and str1 is the destination string.

The old content of the destination string str1 are lost. The function returns a pointer to destination string str1.

Example:-

```
        #include<stdio.h>
        #include<string.h>
        void main()
{
        char str1[10],str2[10];
        printf("Enter a string:");
        scanf("%s",str2);
        strcpy(str1,str2);
        printf("First string:%s\t\tSecond string:%s\n",str1,str2);
        strcpy(str,"Delhi");
        strcpy(str2,"Bangalore");
printf("First string :%s\t\tSecond string:%s",str1,str2);
```

strcat()

This function is used to append a copy of a string at the end of the other string. If the first string is “”Purva” and second string is “Belmont” then after using this function the string becomes “PusvaBelmont”. The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected. A pointer to the first string str1 is returned by the function.

Example:-

```
        #include<stdio.h>

        #include<string.h>

        void main()

{

        char str1[20],str[20];

        printf(“Enter two strings:”);

        gets(str1);

        gets(str2);

        strcat(str1,str2);

        printf(“First string:%s\t second string:%s\n”,str1,str2);

        strcat(str1,”-one”);

        printf(“Now first string is %s\n”,str1);

}
```

Output

```
        Enter two strings: data

        Base
        First string: database second string: database
`       Now first string is: database-one
```

Lecture Note: 14

FUNCTION

A function is a self contained block of codes or sub programs with a set of statements that perform some specific task or coherent task when it is called.

It is something like to hiring a person to do some specific task like, every six months servicing a bike and hand over to it.

Any ‘C’ program contain at least one function i.e main().

There are basically two types of function those are

1. **Library function**
2. **User defined function**

The user defined functions defined by the user according to its requirement

System defined function can’t be modified, it can only read and can be used. These function are supplied with every C compiler

Source of these library function are pre complied and only object code get used by the user by linking to the code by linker

Here in system defined function description:

Function definition : predefined, precompiled, stored in the library

Function declaration : In header file with or function prototype.

Function call : By the programmer

User defined function

Syntax:-

```
Return type     name of function (type 1 arg 1, type2 arg2, type3 arg3)
Return type        function name       argument list of the above syntax
```

So when user gets his own function three thing he has to know, these are.

Function declaration

Function definition

Function call

These three things are represented like

```
     int function(int, int, int);     /*function declaration*/
     main()     /* calling function*/
      {
     function(arg1,arg2,arg3);
      }
  int function(type 1 arg 1,type2 arg2,type3, arg3)   /*function definition/*
  {
  Local variable declaration;
 Statement;
 Return value;
  }
```

Function declaration:-

Function declaration is also known as function prototype. It inform the compiler about three thing, those are name of the function, number and type of argument received by the function and the type of value returned by the function.

While declaring the name of the argument is optional and the function prototype always terminated by the semicolon.

Function definition:-

Function definition consists of the whole description and code of the function.

It tells about what function is doing what are its inputs and what are its out put

It consists of two parts function header and function body

Syntax:-

```
   return type function(type 1 arg1, type2 arg2, type3 arg3) /*function header*/
    {
     Local variable declaration;
     Statement 1;
     Statement 2;
     Return value
    }
```

The return type denotes the type of the value that function will return and it is optional and if it is omitted, it is assumed to be int by default. The body of the function is the compound statements or block which consists of local variable declaration statement and optional return statement.

The local variable declared inside a function is local to that function only. It can’t be used anywhere in the program and its existence is only within this function.

The arguments of the function **definition** are known as **formal arguments**.

Function Call

When the function get called by the calling function then that is called, function call. The compiler execute these functions when the semicolon is followed by the function name.

Example:-

```
 function(arg1,arg2,arg3);
```

The argument that are used inside the function call are called **actual argument**

Ex:-

```
  int S=sum(a, b);                    //actual arguments
```

Actual argument

The arguments which are mentioned or used inside the function call is knows as actual argument and these are the original values and copy of these are actually sent to the called function

It can be written as constant, expression or any function call like

```
Function (x);
 Function (20, 30);
Function (a*b, c*d);
Function(2,3,sum(a, b));
```

Formal Arguments

The arguments which are mentioned in function definition are called formal arguments or dummy arguments.

These arguments are used to just hold the copied of the values that are sent by the calling function through the function call.

These arguments are like other local variables which are created when the function call starts and destroyed when the function ends.

The basic difference between the formal argument and the actual argument are

1) The formal argument are declared inside the parenthesis where as the local variable declared at the beginning of the function block.

2). The **formal argument** are automatically initialized when the copy of actual arguments are passed while other local variable are assigned values through the statements.

Order number and type of actual arguments in the function call should be match with the order number and type of the formal arguments.

Return type

It is used to return value to the calling function. It can be used in two way as

```
                return

      Or       return(expression);

       Ex:-     return (a);

                return (a*b);

               return (a*b+c);
```

Here the 1st return statement used to terminate the function without returning any value

```
Ex:-   /*summation of two values*/

        int sum (int a1, int a2);

       main()
```

```
{

   int a,b;
    printf(“enter two no”);

    scanf(“%d%d”,&a,&b);

    int S=sum(a,b);

     printf(“summation is = %d”,s);

      }

     int sum(intx1,int y1)

      {
    int z=x1+y1;

     Return z;

     }
```

Advantage of function

By using function large and difficult program can be divided in to sub programs and solved. When we want to perform some task repeatedly or some code is to be used more than once at different place in the program, then function avoids this repeatition or rewritten over and over.

Due to reducing size, modular function it is easy to modify and test

Notes:-

C program is a collection of one or more function.

A function is get called when function is followed by the semicolon.

A function is defined when a function name followed by a pair of curly braces

Any function can be called by another function even main() can be called by other function.

```
main()
{

function1()

}

function1()

{

Statement;

function2;

}

function 2()

{


}
```

So every function in a program must be called directly or indirectly by the main() function. A function can be called any number of times.

A function can call itself again and again and this process is called **recursion**.

A function can be called from other function **but** a function can’t be defined in another function

Lecture Note: 15

Category of Function based on argument and return type

i) Function with no argument & no return value

Function that have no argument and no return value is written as:-

```
   void function(void);
  main()
   {
  void  function()
   {
 Statement;
   }
```

Example:-

```
void  me();
    main()
   {
     me();
    printf(“in main”);
   }
   void   me()
  {
     printf(“come on”);
  }
```

```
Output: come on
            inn main
```

ii) Function with no argument but return value

Syntax:-

```
 int   fun(void);
 main()
{
    int r;
    r=fun();
  }
    int   fun()
   {
   reurn(exp);
   }
```

Example:-

```
 int   sum();
 main()
 {
 int   b=sum();
 printf(“entered   %d\n, b”);
  }
int   sum()
  {
 int  a,b,s;
```

```
     s=a+b;

    return s;

       }
```

Here called function is independent and are initialized. The values aren’t passed by the calling function .Here the calling function and called function are communicated partly with each other.

Lecture Note: 16

iii) function with argument but no return value

Here the function have argument so the calling function send data to the called function but called function dose n’t return value.

```
         Syntax:-
                      void   fun (int,int);

                      main()

                     {

                    int (a,b);

                   }

           void   fun(int x, int y);

            {

           Statement;

            }
```

Here the result obtained by the called function.

iv) function with argument and return value

Here the calling function has the argument to pass to the called function and the called function returned value to the calling function.

Syntax:-

```
                 fun(int,int);
                  main()
                {
                    int r=fun(a,b);
                 }
                     int  fun(intx,inty)
                 {
                            return(exp);
                 }
```

Example:

```
      main()
     {
      int fun(int);
      int  a,num;
      printf(“enter value:\n”);
      scanf(“%d”,&a)
```

```
int num=fun(a);
  }
 int fun(int x)
 {
   ++x;
          return x;
   }
```

Call by value and call by reference

There are two way through which we can pass the arguments to the function such as **call by** value and **call by reference**.

1. Call by value

In the call by value copy of the actual argument is passed to the formal argument and the operation is done on formal argument.

When the function is called by ‘call by value’ method, it doesn’t affect content of the actual argument.

Changes made to formal argument are local to block of called function so when the control back to calling function the changes made is vanish.

Example:-

```
  main()
 {
 int x,y;
change(int,int);
```

```
            printf(“enter two values:\n”);
             scanf(“%d%d”,&x,&y);
              change(x ,y);
          printf(“value of x=%d and y=%d\n”,x ,y);
            }
          change(int a,int b);
         {
          int k;
          k=a;
          a=b;
          b=k;
         }
```

```
Output: enter two values: 12
 23
Value of x=12 and y=23
```

2. Call by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value.

Here formal argument is alter to the actual argument, it means formal arguments calls the actual arguments.

Example:-

```
        void main()
```

```
     {
          int a,b;
          change(int *,int*);
          printf(“enter two values:\n”);
          scanf(“%d%d”,&a,&b);
          change(&a,&b);
          printf(“after changing two value of a=%d and b=%d\n:”a,b);
     }
          change(int *a, int *b)
          {
          int k;
           k=*a;
          *a=*b;
        *b= k;
printf(“value in this function a=%d and b=%d\n”,*a,*b);
}
```

```
Output: enter two values: 12
32
Value in this function a=32 and b=12
After changing two value of a=32 and b=12
```

So here instead of passing value of the variable, directly passing address of the variables. Formal argument directly access the value and swapping is possible even after calling a function.

Lecture Note: 17

Local, Global and Static variable

Local variable:-

variables that are defined with in a body of function or block. The local variables can be used only in that function or block in which they are declared. Same variables may be used in different functions such as

```
function()

{

        int a,b;

        function 1();

}

function2 ()

{

        int a=0;

        b=20;

}
```

Global variable:-

the variables that are defined outside of the function is called global variable. All functions in the program can access and modify global variables. Global variables are automatically initialized at the time of initialization.

Example:

```
        #include<stdio.h>

        void function(void);

        void function1(void);

        void function2(void);

        int a, b=20;

void main()

{

        printf("inside main a=%d,b=%d \n",a,b);

function();

function1();

function2();

        }

         function()

        {

                Prinf("inside function a=%d,b=%d\n",a,b);

        }

        function 1()

        {
```

```
        prinf(“inside function a=%d,b=%d\n”,a,b);
    }
    function 2()
    {
        prinf(“inside function a=%d,b=%d\n”,a,);
    }
```

Static variables: static variables are declared by writing the key word static.

-syntax:-

```
    static data type variable name;
    static int a;
```

-the static variables initialized only once and it retain between the function call. If its variable is not initialized, then it is automatically initialized to zero.

Example:

```
    void fun1(void);
    void fun2(void);
    void main()
    {
        fun1();
        fun2();
    }
    void fun1()
    {
            int a=10, static int b=2;
            printf(“a=%d, b=%d”,a,b);
            a++;
            b++;
    }
Output:a= 10 b=  2
            a=10   b= 3
```

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. It is powerful technique of writing complicated algorithm in easiest way. According to recursion problem is defined in term of itself. Here statement with in body of the function calls the same function and same times it is called as circular definition. In other words recursion is the process of defining something in form of itself.

Syntax:

```
    main ()
    {
            rec(); /*function call*/
            rec();
            rec();
```

Ex:- /*calculate factorial of a no.using recursion*/

```
    int fact(int);
    void main()
    {
        int num;
        printf(“enter a number”);
        scanf(“%d”,&num);
        f=fact(num);
        printf(“factorial is =%d\n”f);
    }
    fact (int num)
    {
        If  (num==0||num==1)
return 1;
else
return(num*fact(num-1));
    }
```

Lecture Note: 18

Monolithic Programming

The program which contains a single function for the large program is called monolithic program. In monolithic program not divided the program, it is huge long pieces of code that jump back and forth doing all the tasks like single thread of execution, the program requires. Problem arise in monolithic program is that, when the program **size** increases it leads inconvenience and difficult to maintain such as testing, debugging etc. Many disadvantages of monolithic programming are:

1. Difficult to check error on large programs size.
2. Difficult to maintain because of huge size.
3. Code can be specific to a particular problem. i.e. it cannot be reused.

Many early languages (FORTRAN, COBOL, BASIC, C) required one huge workspace with labelled areas that may does specific tasks but are not isolated.

Modular Programming

The process of subdividing a computer program into separate sub-programs such as functions and subroutines is called Modular programming. **Modular programming sometimes also called as structured programming.** It enables multiple programmers to divide up the large program and debug pieces of program independently and tested.

. Then the linker will link all these modules to form the complete program. This principle dividing software up into parts, or modules, where a module can be changed, replaced, or removed, with minimal effect on the other software it works with. Segmenting the program into modules clearly defined functions, it can determine the source of program errors more easily. Breaking down program functions into modules, where each of which accomplishes one function and contains all the source code and variables needed to accomplish that function. Modular program is the solution to the problem of very large program that are difficult to debug, test and maintain. A program module may be rewritten while its inputs and outputs remain the same. The person making a change may only understand a small portion of the original program.

Object-oriented programming (OOP) is compatible with the modular programming concept to a large extent.

. , Less code has to be written that makes shorter.

- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Programs can be designed more easily because a small team deals with only a small part of the entire code.
- Modular programming allows many programmers to collaborate on the same application.
- The code is stored across multiple files.
- Code is short, simple and easy to understand and modify, make simple to figure out how the program is operate and reduce likely hood of bugs.
- Errors can easily be identified, as they are localized to a subroutine or function or isolated to specific module.
- The same code can be reused in many applications.
- The scoping of variables and functions can easily be controlled.

Disadvantages
However it may takes longer to develop the program using this technique.

Storage Classes

Storage class in c language is a specifier which tells the compiler where and how to store variables, its initial value and scope of the variables in a program. Or attributes of variable is known as storage class or in compiler point of view a variable identify some physical location within a computer where its string of bits value can be stored is known as storage class.

The kind of location in the computer, where value can be stored is either in the memory or in the register. There are various storage class which determined, in which of the two location value would be stored.

Syntax of declaring storage classes is:-

storageclass datatype variable name;

There are four types of storage classes and all are keywords:-

1) Automatic (auto)

2) Register (register)

3) Static (static)

4) External (extern)

Examples:-

```
auto float x; or float x;
     extern int x;
     register char c;
     static int y;
```

Compiler assume different storage class based on:-

1) Storage class:- tells us about storage place(where variable would be stored).

2) Intial value :-what would be the initial value of the variable.

If initial value not assigned, then what value taken by uninitialized variable.

3) Scope of the variable:-what would be the value of the variable of the program.

4)Life time :- It is the time between the creation and distribution of a variable or how long would variable exists.

1. Automatic storage class

The keyword used to declare automatic storage class is auto.

Its features:-

Storage-memory location

Default initial value:-unpredictable value or garbage value.

Scope:-local to the block or function in which variable is defined.

Life time:-Till the control remains within function or block in which it is defined. It terminates when function is released.

The variable without any storage class specifier is called automatic variable.

Example:-

```
main( )

{

auto int i;

printf("i=",i);

}
```

Lecture Note: 19

2. Register storage class

The keyword used to declare this storage class is register.

The features are:-

Storage:-CPU register.

Default initial value :-garbage value

Scope :-local to the function or block in which it is defined.

Life time :-till controls remains within function or blocks in which it is defined.

Register variable don't have memory address so we can't apply address operator on it. CPU register generally of 16 bits or 2 bytes. So we can apply storage classes only for integers, characters, pointer type.

Variable stored in register storage class always access faster than,which is always stored in the memory. But to store all variable in the CPU register is not possible because of limitation of the register pair.

And when variable is used at many places like loop counter, then it is better to declare it as register class.

Example:-

```
main( )

{
register int i;

for(i=1;i<=12;i++)

printf("%d",i);

}
```

3 Static storage class

The keyword used to declare static storage class is static.

Its feature are:-

Storage:-memory location

Default initial value:- zero

Scope :- local to the block or function in which it is defined.

Life time:- value of the variable persist or remain between different function call.

Example:-

```
main( )
```

```
{
reduce( );
reduce( );
reduce ( );
}
reduce( )
{
static int x=10;
printf("%d",x);
x++;
}
```

Output:-10,11,12

External storage classes

The keyword used for this class is extern.

Features are:-

Storage:- memory area

Default initial value:-zero

Scope :- global

Life time:-as long as program execution remains it retains.

Declaration does not create variables, only it refer that already been created at somewhere else. So, memory is not allocated at a time of declaration and the external variables are declared at outside of all the function.

Example:-

```
     int i,j;
     void main( )
   {
    printf( “i=%d”,i );
     receive( );
     receive ( );
     reduce( );
     reduce( );
}
     receive( )
     {
    i=i+2;
  printf(“on increase i=%d”,i);
    }
  reduce( )
   {
   i=i-1;
  printf(“on reduce i=%d”,i);
         }
```

Output:-i=0,2,4,3,2.

When there is large program i.e divided into several files, then external variable should be preferred. External variable extend the scope of variable.

Lecture Note: 20

POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

Data type *pointer name;

Here * before pointer indicate the compiler that variable declared as a pointer.

e.g.

```
int *p1; //pointer to integer type

float *p2; //pointer to float type

char *p3; //pointer to character type
```

When pointer declared, it contains garbage value i.e. it may point any value in the memory.

Two operators are used in the pointer i.e. **address operator(&)** and **indirection operator or dereference operator (*).**

Indirection operator gives the values stored at a particular address.

Address operator cannot be used in any constant or any expression.

Example:

```
    void main()

    {

      int i=105;

      int *p;

      p=&i;

t
printf(“value of i=%d”,*p);

printf(“value of i=%d”,*/&i);

printf(“address of i=%d”,&i);

printf(“address of i=%d”,p);

printf(“address of p=%u”,&p);

}
```

Pointer Expression

Pointer assignment

```
int i=10;

int *p=&i;//value assigning to the pointer
```

Here declaration tells the compiler that P will be used to store the address of integer value or in other word P is a pointer to an integer and *p reads the **value at the address contain in p.**

P++;

printf(“value of p=%d”);

We can assign value of 1 pointer variable to other when their base type and data type is same or both the pointer points to the same variable as in the array.

Int *p1,*p2;

P1=&a[1];

P2=&a[3];

We can assign constant 0 to a pointer of any type for that symbolic constant ‘**NULL**’ is used such as

*p=NULL;

It means pointer doesn’t point to any valid memory location.

Pointer Arithmetic

Pointer arithmetic is different from ordinary arithmetic and it is perform relative to the data type(base type of a pointer).

Example:-

If integer pointer contain address of 2000 on incrementing we get address of 2002 instead of 2001, because, size of the integer is of 2 bytes.

Note:-

When we move a pointer, somewhere else in memory by incrementing or decrement or adding or subtracting integer, it is not necessary that, pointer still pointer to a variable of same data, because, memory allocation to the variable are done by the compiler.

But in case of array it is possible, since there data are stored in a consecutive manner.

Ex:-

```
void main( )

{

static int a[ ]={20,30,105,82,97,72,66,102};

int *p,*p1;

P=&a[1];

P1=&a[6];

printf(“%d”,*p1-*p);

printf(“%d”,p1-p);

}
```

Arithmetic operation never perform on pointer are:

addition, multiplication and division of two pointer.

multiplication between the pointer by any number.

division of pointer by any number

-add of float or double value to the pointer.

Operation performed in pointer are:-

/* Addition of a number through pointer */

Example

```
int i=100;

int *p;
```

```
p=&i;
p=p+2;
p=p+3;
p=p+9;
```

ii /* Subtraction of a number from a pointer'*/

Ex:-

```
int i=22;
*p1=&a;
p1=p1-10;
p1=p1-2;
```

iii- Subtraction of one pointer to another is possible when pointer variable point to an element of same type such as an array.

Ex:-

```
in tar[ ]={2,3,4,5,6,7};
int *ptr1,*ptr1;
ptr1=&a[3]; //2000+4
ptr2=&a[6];  //2000+6
```

Precedence of dereference (*) Operator and increment operator and decrement operator

The precedence level of difference operator increment or decrement operator is same and their associatively from right to left.

Example :-

int x=25;

int *p=&x;

Let us calculate int y=*p++;

Equivalent to *(p++)

Since the operator associate from right to left, increment operator will applied to the pointer p.

i) int y=*p++; equivalent to *(p++)

p =p++ or p=p+1

ii) *++p;→*(++p)→p=p+1

y=*p

iii)int y=++*p

equivalent to ++(*p)

p=p+1 then *p

iv) y=(*p)++→equivalent to *p++

y=*p then

P=p+1 ;

Since it is postfix increment the value of p.

Pointer Comparison

Pointer variable can be compared when both variable, object of same data type and it is useful when both pointers variable points to element of same array.

Moreover pointer variable are compared with zero which is usually expressed as null, so several operators are used for comparison like the relational operator.

==,!=,<=,<,>,>=, can be used with pointer. Equal and not equal operators used to compare two pointer should finding whether they contain same address or not and they will equal only if are null or contains address of same variable.

Ex:-

```
  void main()
{
static int arr[]={20,25,15,27,105,96}
int *x,*y;
x=&a[5];
y=&(a+5);
if(x==y)
printf(“same”);
else
printf(“not”);



}
```

Lecture Note: 22

Pointer to pointer

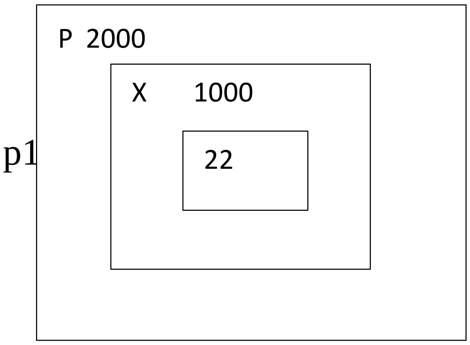
Addition of pointer variable stored in some other variable is called pointer to pointer variable.

Or

Pointer within another pointer is called pointer to pointer.

Syntax:-

```
        Data type **p;

        int x=22;

        int *p=&x;

        int **p1=&p;

printf("value of x=%d",x);

printf("value of x=%d",*p);

printf("value of x=%d",*&x);

printf("value of x=%d",**p1);

printf("value of p=%u",&p);

printf("address of p=%u",p1);

printf("address of x=%u",p);

printf("address of p1=%u",&p1);

printf("value of p=%u",p);

printf("value of p=%u",&x);
```



*Under revision

3000

Pointer vs array

Example :-

```
    void main()

{

static char arr[]=”Rama”;

char*p=”Rama”;

printf(“%s%s”, arr, p);
```

In the above example, at the first time printf( ), print the same value array and pointer.

Here array arr, as **pointer to character** and **p act as a pointer to array of character .** When we are trying to increase the value of arr it would give the error because its known to compiler about an array and its base address which is always printed to base address is known as constant pointer and the base address of array which is not allowed by the compiler.

```
printf(“size of (p)”,size of (ar));

size of (p)             2/4 bytes

size of(ar)               5 byes
```

Sructure

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure declaration-

```
struct tagname

{

Data type member1;

Data type member2;

Data type member3;

………

………

Data type member n;

};
```

OR

```
struct

{

Data type member1;

Data type member2;
Data type member3;
………
………
Data type member n;
};
```

OR

```
struct tagname
{
struct element 1;
struct element 2;
struct element 3;
………
………
struct element n;
};
```

Structure variable declaration;

```
struct student
{
        int age;
char name[20];
char branch[20];
}; struct student s;
```

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student
{
        int age=20;
char name[20]=”sona”;
}s1;
```

The above is **invalid.**

A structure can be initialized as

```
struct student
{
        int age,roll;
char name[20];
} struct student s1={16,101,”sona”};
  struct student s2={17,102,”rupa”};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativety is from left to right.

```
structure variable ;
s1.name[];
s1.roll;
s1.age;
```

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int roll, age;
char branch;
} s1,s2;
printf(“\n enter roll, age, branch=”);
scanf(“%d %d %c”, &s1.roll, &s1.age, &s1.branch);
s2.roll=s1.roll;
printf(“ students details=\n”);
printf(“%d %d %c”, s1.roll, s1.age, s1.branch);
printf(“%d”, s2.roll);
```

```
}
```

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Lecture Note:24

Size of structure-

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

```
sizeof(struct student); or

sizeof(s1);

sizeof(s2);
```

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>

#include<string.h>

struct student

{
```

```
char name[30];
char branch[25];
int roll;
};
void main()
{
struct student s[200];
int i;
s[i].roll=i+1;
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:");
scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch);
printf("\n");
}
printf("\nDisplaying information of students:\n\n");
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in above example.

Array within structures

```
struct student
{
char name[30];
int roll,age,marks[5];
}; struct student s[200];
```

We can also initialize using same syntax as in array.

Nested structure

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
```

```
{
element 1;
element 2;
………
………
struct student1
{
member 1;
member 2;
}variable 1;
……….
……….
element n;
}variable 2;
```

It is possible to define structure outside & declare its variable inside other structure.

```
struct date
{
int date,month;
};
struct student
{
 char nm[20];
int roll;
struct date d;
}; struct student s1;
   struct student s2,s3;
```

Nested structure may also be initialized at the time of declaration like in above example.

```
struct student s={“name”,200, {date, month}};
                      {“ram”,201, {12,11}};
```

Nesting of structure within itself is not valid. Nesting of structure can be extended to any level.

```
struct time
{
int hr,min;
};
struct day
{
int date,month;
struct time t1;
};
struct student
```

```
{
char nm[20];
struct day d;
}stud1, stud2, stud3;
```

Lecture Note: 25

Passing structure elements to function

We can pass each element of the structure through function but passing individual element is difficult when number of structure element increases. To overcome this, we use to pass the whole structure through function instead of passing individual element.

```
#include<stdio.h>
#include<string.h>
void main()
{
struct student
{
char name[30];
char branch[25];
int roll;
}struct student s;
printf("\n enter name=");
gets(s.name);
printf("\nEnter roll:");
scanf("%d",&s.roll);
printf("\nEnter branch:");
gets(s.branch);
display(name,roll,branch);
}
display(char name, int roll, char branch)
{
printf(“\n name=%s,\n roll=%d, \n branch=%s”, s.name, s.roll. s.branch);
}
```

Passing entire structure to function

```
#include<stdio.h>
#include<string.h>
struct student
{
char name[30];
int age,roll;
};
display(struct student);                        //passing entire structure
void main()
```

```
{
   struct student s1={”sona”,16,101 };
   struct student s2={”rupa”,17,102 };
display(s1);
display(s2);
}
display(struct student s)
{
printf(“\n name=%s, \n age=%d ,\n roll=%d”, s.name, s.age, s.roll);
}
```

```
Output: name=sona
            roll=16
```

Lecture Note: 26

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union:

```
union student
{
datatype member1;
datatype member2;
};
```

Like structure variable, union variable can be declared with definition or separately such as

```
union union name
{
Datatype member1;
}var1;
```

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

```
struct student

{

int i;

char ch[10];

};struct student s;
```

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Lecture Note:27

Nested of Union

When one union is inside the another union it is called nested of union.

Example:-

```
union a

{
int i;

int age;

};

union b
```

```
{
char name[10];
union a aa;
}; union b bb;
```

There can also be union inside structure or structure in union.

Example:-

```
  void  main()
    {
   struct a
  {
int i;
char ch[20];
};
struct b
{
int i;
char d[10];
};
union z
{
struct a a1;
struct b b1;
}; union z z1;
z1.b1.j=20;
z1.a1.i=10;
z1.a1.ch[10]= “ i“;
z1.b1.d[0]=”j “;
printf(“ “);
```

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.

Two types of problem may occur in static memory allocation.

If number of values to be stored is less than the size of memory, there would be wastage of memory.

If we would want to store more values by increase in size during the execution on assigned size then it fails.

Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function.** These library function prototype are found in the header file, “alloc.h” where it has defined.

Function take memory from memory area is called heap and release when not required.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is void*malloc(size);

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

int *p=(datatype*)malloc(size)

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type**(datatype)** and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example int*p=(int*)malloc(10);

So, from the above pointer p, allocated IO contigious memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as *p=(int*)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf(“not sufficient memory”);

Example:

/*calculate the average of mark*/

void main()

{

int n , avg,i,*p,sum=0;

```
printf("enter the no. of marks ”);

scanf(“%d”,&n);
p=(int *)malloc(n*size(int));

if(p==null)

printf(“not sufficient”);

exit();

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

scanf(“%d”,(p+i));

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

Printf(“%d”,*(p+i));

sum=sum+*p;

avg=sum/n;

printf(“avg=%d”,avg);
```

Lecture Note: 28

calloc()

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .

two arguments are there

1st argument specify number of blocks

2nd argument specify size of each block.

Example:-

```
        int *p= (int*) calloc(5, 2);
```

```
int*p=(int *)calloc(5, size of (int));
```

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialised by zero(but this initialisation) is not reliable.

realloc()

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation of memory.

It takes two argument such as;

```
int *ptr=(int *)malloc(size);
```

```
int*p=(int *)realloc(ptr, new size);
```

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.

Example:

```
#include<stdio.h>

#include<alloc.h>

void main()

int i,*p;
```

```
p=(int*)malloc(5*size of (int));
if(p==null)
{
printf(“space not available”);
exit();
printf(“enter 5 integer”);
for(i=0;i<5;i++)
{
scanf(“%d”,(p+i));
int*ptr=(int*)realloc(9*size of (int) );
if(ptr==null)
{
printf(“not available”);
exit();
}
printf(“enter 4 more integer”);
for(i=5;i<9;i++)
scanf(“%d”,(p+i));
for(i=0;i<9;i++)
                    printf(“%d”,*(p+i));
}
```

free()

Function free() is used to release space allocated dynamically, the memory released by free() is made available to heap again. It can be used for further purpose.

Syntax for free declaration .

void(*ptr)

Or

free(p)

When program is terminated, memory released automatically by the operating system. Even we don’t free the memory, it doesn’t give error, thus lead to memory leak.

We can’t free the memory, those didn’t allocated.

Lecture Note: 29

Dynamic array

Array is the example where memory is organized in contiguous way, in the dynamic memory allocation function used such as malloc(), calloc(), realloc() always made up of contiguous way and as usual we can access the element in two ways as:

Subscript notation

Pointer notation

Example:

```
#include<stdio.h>
#include<alloc.h>
void main()
{
printf(“enter the no.of values”);
scanf(“%d”,&n);
p=(int*)malloc(n*size of int);
If(p==null)
printf(“not available memory”);
exit();
}
for(i=0;i<n;i++)
{
printf(“enter an integer”);
scanf(“%d”,&p[i]);
for(i=0;i<n;i++)
{
printf(“%d”,p[i]);
}
}
```

File handling

File: the file is a permanent storage medium in which we can store the data permanently.

Types of file can be handled

we can handle three type of file as

(1) sequential file

(2) random access file

(3) binary file

File Operation

opening a file:

Before performing any type of operation, a file must be opened and for this fopen() function is used.

syntax:

file-pointer=fopen("FILE NAME ","Mode of open");

example:

    FILE *fp=fopen("ar.c","r");

If fopen() unable to open a file than it will return NULL to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the address of a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

FILE* var;

Modes of open

The file can be open in three different ways as

Read mode ' r'/rt

Write mode 'w'/wt

Appened Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file

Syntax:

```
character_variable=getc(file_ptr);
```

Writing acharacter into a file

putc() is used to write a character into a file

puts(character-var,file-ptr);

ClOSING A FILE

fclose() function close a file.

fclose(file-ptr);

fcloseall () is used to close all the opened file at a time

File Operation

The following file operation carried out the file

(1)creation of a new file

(3)writing a file

(4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using A new type called **file pointer**.

Operation with fopen()

File pointer=fopen(“FILE NAME”,”mode of open”);

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Lecture Note: 30

Reading and writing a characters from/to a file

fgetc() is used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

```
/*Program to copy a file to another*/
#include<stdio.h>
void main()
{
FILE *fs,*fd;
char ch;
If(fs=fopen(“scr.txt”,”r”)==0)
{
printf(“sorry….The source file cannot be opened”);
return;
}
If(fd=fopen(“dest.txt”,”w”)==0)
{
printf(“Sorry…..The destination file cannot be opened”);
fclose(fs);
return;
}
while(ch=fgets(fs)!=EOF)
fputc(ch,fd);
fcloseall();
}
```

Reading and writing a string from/to a file

getw() is used for reading a string from the file

Syntax:

```
gets(file pointer);
```

putw() is used to writing a character to a file

Syntax:

```
fputs(integer,file_pointer);
```

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
FILE *fp;
int word;
/*place the word in a file*/
fp=fopen(“dgt.txt”,”wb”);
If(fp==NULL)
{
printf(“Error opening file”);
exit(1);
}
word=94;
putw(word,fp);
If(ferror(fp))
printf(“Error writing to file\n”);
else
printf(“Successful write\n”);
fclose(fp);
/*reopen the file*/
fp=fopen(“dgt.txt”,”rb”);
If(fp==NULL)
{
printf(“Error opening file”);
exit(1);
}

/*extract the word*/
word=getw(fp);
If(ferror(fp))
printf(“Error reading file\n”);
else
printf(“Successful read:word=%d\n”,word);
/*clean up*/
fclose(fp);
}
```

Lecture Note: 31

Reading and writing a string from/to a file

fgets() is used for reading a string from the file

Syntax:

fgets(string, length, file pointer);

fputs() is used to writing a character to a file

Syntax:

fputs(string,file_pointer);

```
#include<string.h>
#include<stdio.h>
void main(void)
{
FILE*stream;
char string[]=”This is a test”;
char msg[20];
/*open a file for update*/
stream=fopen(“DUMMY.FIL”,”w+”);

/*write a string into the file*/
fwrite(string,strlen(string),1,stream);
/*seek to the start of the file*/
fseek(stream,0,SEEK_SET);
/*read a string from the file*/

fgets(msg,strlen(string)+1,stream);
/*display the string*/

printf("%s",msg);

fclose(stream);

}
```

BOOKS:

1 E.Balagurusamy “Programming in C”. Tata McGraw Hill

2 Y. Kanetkar “Let Us C”. BPB publication

3Ashok N. Kamthane “Programming with ANSI and TURBO C”. Pearson Education

4Programming in C, a complete introduction to the programming language, Stephan G. Kocham, third edition

5 C in Depth, S.K Srivastava and Deepali Srivastava