COMPUTER SYSTEMS

A Computer is an electronic device that stores, manipulates and retrieves the data. We can also refer computer computes the information supplied to it and generates data. A System is a group of several objects with a process. For Example: Educational System involves teacher, students (objects). Teacher teaches subject to students i.e., teaching (process). Similarly a computer system can have objects and process.

The following are the objects of computer System

a) User (A person who uses the computer)

b) Hardware

c) Software

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

The hardware of a computer system can be classified as

Input Devices (I/P)

Processing Devices (CPU)

Output Devices (O/P)

ALU: It performs the Arithmetic and Logical Operations such as

 +,-,*,/   (Arithmetic Operators)

&&, || (Logical Operators)

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

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

The Memory unit is classified into two types.

They are   1) Primary Memory

2) Secondary Memory

Primary memory: The following are the types of memories 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 cannot 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.

5

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.

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, Mercian 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

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 computers which ask for processing are only dumb terminals.

Client/Server Environment

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

Distributed Computing

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

Example: Ebay on Internet

COMPUTER LANGUAGES

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

1940,,s	--	Machine Languages
1950,,s	--	Symbolic Languages
1960,,s	--	High Level Languages

Machine Language

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

Advantages:

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

Disadvantages:

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

Example Addition of two numbers

```
        2          □ 0 0 1 0
   +    3          □ 0 0 1 1
       ---         ---------------
        5     □      0 1 0 1
       ---         ---------------
```

Symbolic Languages (or) Assembly Language

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

Advantages:

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

hardware Disadvantages:

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

Example:

2	PUSH2,A
3	PUSH3,B
+	ADDA,B
5	PRINTC

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

9

5) Better readability

6) Low Development cost

7) Easier to document

8) Portable

Disadvantages:

1) Needs translator

2) Requires high execution time

3) Poor control on hardware

4) Less efficient

Example: C language

#include<stdio.h>

void main()
{
        int a, b, c;
        scanf("%d%d%",&a,&b);

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

Difference between Machine, Assembly, High Level Languages

Feature	Machine	Assembly	High Level
Form	0,,s and 1,,s	Mnemonic codes	Normal English
Machine Dependent	Dependent	Dependent	Independent
Translator	Not Needed	Needed(Assembler)	Needed(Compiler)
Execution Time	Less	Less	High
Languages	Only one	Different Manufacturers	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 executed 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 soon.

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 thecode.

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

11

4)   Executing the program

Sl. No.	Phase	Name of Code	Tools	File Extension
1	Text Editor	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, Get.,

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.

12

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 programs functions and system 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

P. Write a algorithm 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=0then
      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 arithmetic operations and data-manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flow line
	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.

15
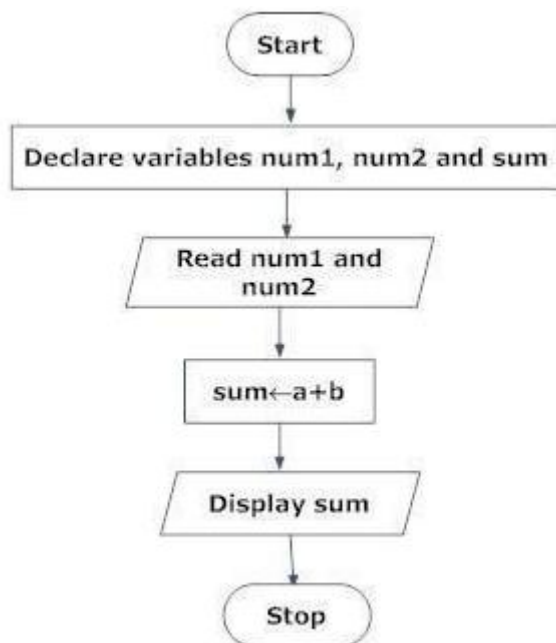
Draw flowchart to find the largest among three different numbers entered by user.
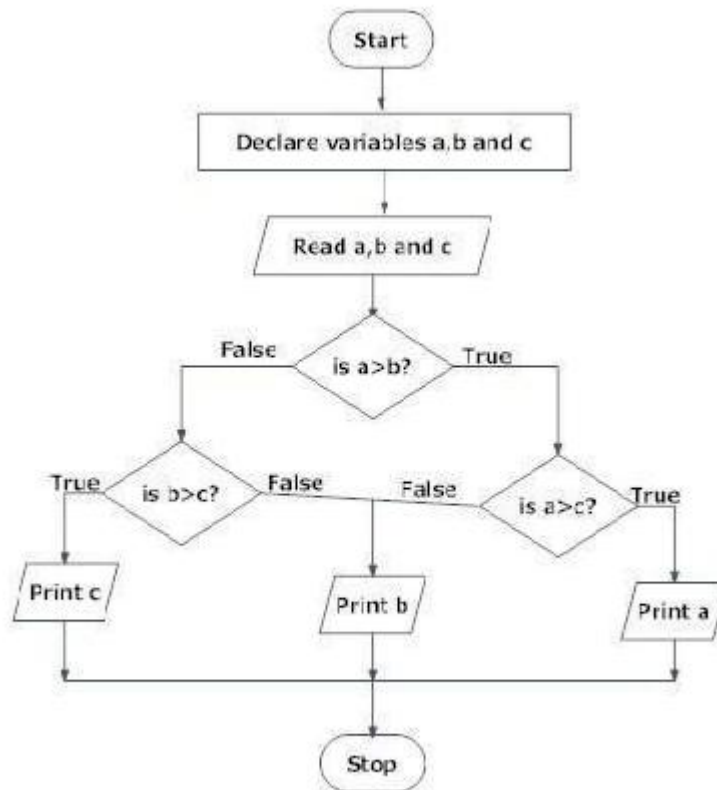
INTRODUCTION TO C LANGUAGE

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 Equipment 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 around1970
- The language was formalized in 1988 by the American National Standard Institute (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 adopted 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
- Prints poolers
- Network Drivers
- Modern Programs
- Databases
- Language Interpreters
- Utilities

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.
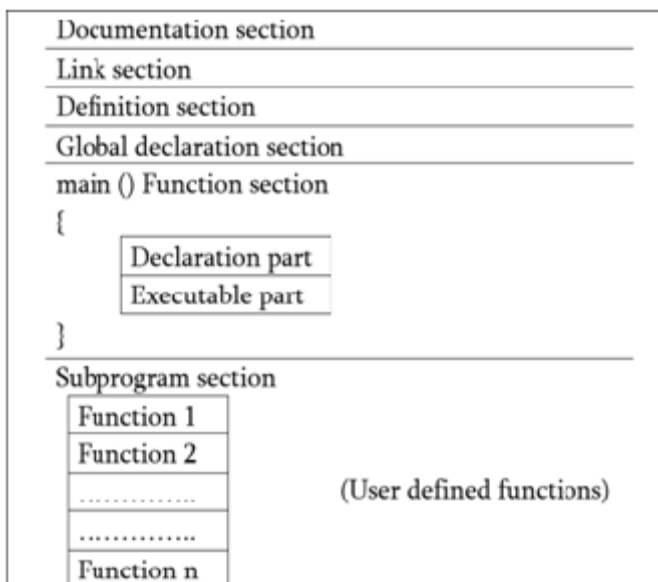
HISTORY TO C LANGUAGE

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 DECPDP-

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   (User defined functions) |
| .... |
| .... |
| Function n |

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 semicolon.

6. **Subprogram section:** If the program is a program then the subprogram section contains all the 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.
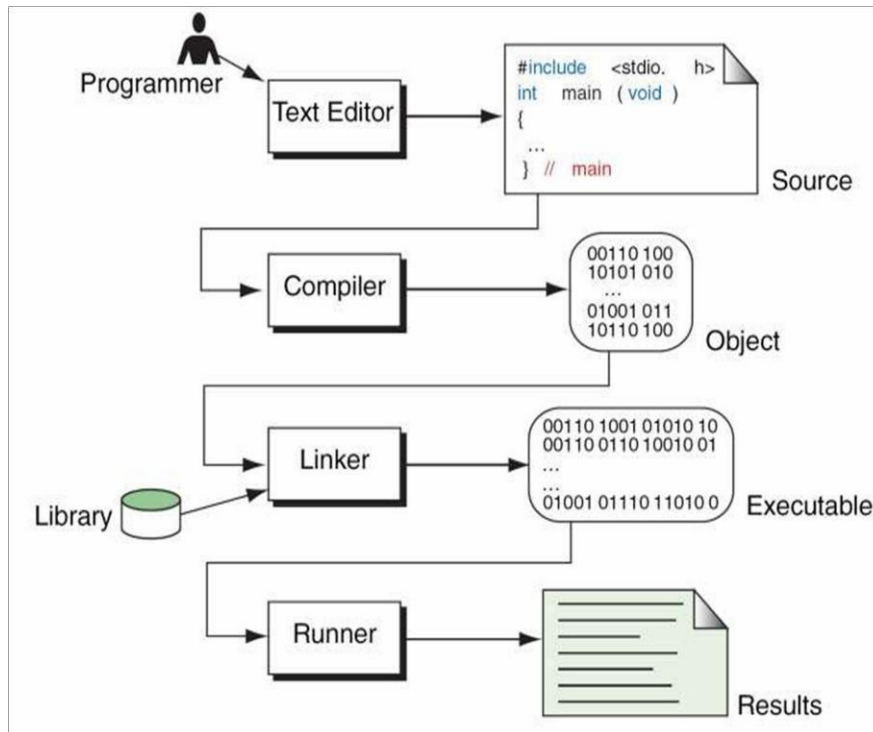
The steps involved in Creating and Running Programs are:

- Writing and Editing Programs
- Compiling Programs
- Linking Programs
- Executing Programs

Writing and Editing Programs

- To solve a particular problem a Program has to be created as a file using text editor /word processor. This is called source file.
- The program has to be written as per the structure and rules defined by the high-level language that is used for writing the program (C, JAVAetc).

Compiling Programs

- The compiler corresponding to the high-level language will scan the source file, checks the program for the correct grammar (syntax) rules of the language.
- If the program is syntactically correct, the compiler generates an output file called _Object File,, which will be in a binary format and consists of machine language instructions corresponding to the computer on which the program gets executed.

Linking Programs:

- Linker program combines the Object File with the required library functions to produce another file called ― executable file‖. Object file will be the input to the linker program.
- The executable file is created on disk. This file has to be put into (loaded) the memory.

Executing Programs:

- Loader program loads the executable file from disk into the memory and directs the CPU to start execution.
- The CPU will start execution of the program that is loaded into the memory.

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	size of	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 whitespace.

Itshouldbeupto31characterslongasonlyfirst31charactersaresignificant. Some

examples of 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.

$$[] \, () \, \{\} \, , ; : * \ldots = \#$$

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 initialize 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 andf.
int d = 3, f=5;             // definition and initializing d and f.
byte z=22;                  // definition and initializesz.
char x= 'x';                // the variable x has the value'x'.
```

For definition without an initialize: 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.

Data types in C Language

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These data types have different storage capacities.

C language supports 2 different type of data types:

1. **Primary data types**:

   These are fundamental data types in C namely integer (int), floating point (float),

   character(char) and void.

2. **Derived data types**:

   Derived data types are nothing but primary data types but a little twisted or grouped together

   like **array**, **structure**, **union** and **pointer**. These are discussed in details later.

   Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x

   can hold only integer values. Every variable which is used in the program must be declared as what

   data-type it is.

27

Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 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	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

`void` type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this data type as we start learning more advanced topics in C language, like functions, pointers etc.

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

Arithmetic operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20, then −

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Example

Try the following example to understand all the arithmetic operators available in C −

```
#include <stdio.h>
main() {
int a = 21;
   int b = 10;
   int c ;
c = a + b;
   printf("Line 1 - Value of c is %d\n", c );
c = a - b;
   printf("Line 2 - Value of c is %d\n", c );
```

```
c = a * b;

   printf("Line 3 - Value of c is %d\n", c );

c = a / b;

   printf("Line 4 - Value of c is %d\n", c );

c = a % b;

   printf("Line 5 - Value of c is %d\n", c );

c = a++;

   printf("Line 6 - Value of c is %d\n", c );

c = a--;

   printf("Line 7 - Value of c is %d\n", c );

}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

Assignment operators

The following table lists the assignment operators supported by the C language −

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand	C += A is equivalent

	and assign the result to the left operand.	to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2

^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Relational operators

The following table shows all the relational operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Example

Try the following example to understand all the relational operators available in C −

```c
#include <stdio.h>

main() {

int a = 21;

int b = 10;

int c ;

if( a == b ) {

    printf("Line 1 - a is equal to b\n" );

  } else {

    printf("Line 1 - a is not equal to b\n" );

  }

if ( a < b ) {

    printf("Line 2 - a is less than b\n" );

  } else {

    printf("Line 2 - a is not less than b\n" );

  }


  if ( a > b ) {

    printf("Line 3 - a is greater than b\n" );

  } else {

    printf("Line 3 - a is not greater than b\n" );

  }

/* Lets change value of a and b */

  a = 5;

  b = 20;

if ( a <= b ) {
```

```
   printf("Line 4 - a is either less than or equal to b\n" );

 }

if ( b >= a ) {

   printf("Line 5 - b is either greater than or equal to b\n" );

 }

}
```

When you compile and execute the above program, it produces the following result –

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

Logical operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Example

Try the following example to understand all the logical operators available in C −

35

```c
#include <stdio.h>

main() {

int a = 5;

  int b = 20;

  int c ;

if ( a && b ) {

    printf("Line 1 - Condition is true\n" );

  }

if ( a || b ) {

    printf("Line 2 - Condition is true\n" );

  }

/* lets change the value of a and b */

  a = 0;

  b = 10;

if ( a && b ) {

    printf("Line 3 - Condition is true\n" );

  } else {

    printf("Line 3 - Condition is not true\n" );

  }

if ( !(a && b) ) {

    printf("Line 4 - Condition is true\n" );

  }

}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bit wise operators

The following table lists the Bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -60, i.e,. 1100 0100 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Try the following example to understand all the bitwise operators available in C −

```c
#include <stdio.h>

main() {

unsigned int a = 60; /* 60 = 0011 1100 */

  unsigned int b = 13;           /* 13 = 0000 1101 */

  int c = 0;

c = a&b;        /* 12 = 0000 1100 */

printf("Line 1 - Value of c is %d\n", c );

c = a|b;        /* 61 = 0011 1101*/

  printf("Line 2 - Value of c is %d\n", c);

c = a ^b;       /* 49 = 0011 0001 */

  printf("Line 3 - Value of c is %d\n", c );

c = ~a;         /*-61 = 1100 0011 */

printf("Line 4 - Value of c is %d\n", c );

c = a<<2;       /* 240 = 1111 0000*/

  printf("Line 5 - Value of c is %d\n", c);

c = a>>2;       /* 15 = 0000 1111*/

  printf("Line 6 - Value of c is %d\n", c );

}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Conditional operators (ternary operators)

- Conditionaloperatorsreturnonevalueifconditionistrueandreturnsanother valueisconditionisfalse.
- This operator is also called as ternaryoperator **(Condition? true_value: false_value);**
              (A > 100 ? 0 : 1);
      Syntax   :

38

> **Example :**

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

Example

```
#include<stdio.h>
int main()
{
int num;
printf("Enter the Number : ");
scanf("%d",&num);
(num%2==0)?printf("Even"):printf("Odd");
}
```

Increment/decrement operators

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; (or) i++;
decrement operator: --i; (or) 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‖.

```
#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‖.

```c
#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

Special operators

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.
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 isusedtogetthe value of the variable that the pointer is pointing to. Please refer **C – pointer** topic to know more about pointers.

```c
#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

```c
#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) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*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
Note1: **Note2:**	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. 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 expressions

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

44

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:

45

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 thearithmetic

  expression.

## CONTROL STRUCTURES

Decision statements- if and Switch

if Statement

Syntax of if statement:
The statements inside the body of ‒if‖ only execute if the given condition returns true. If the condition returns false then the statements inside ‒if‖ are skipped.

```
if (condition)
{
    //Block of C statements here
    //These statements will only execute if the condition is true
}
```

Flow Diagram of if statement
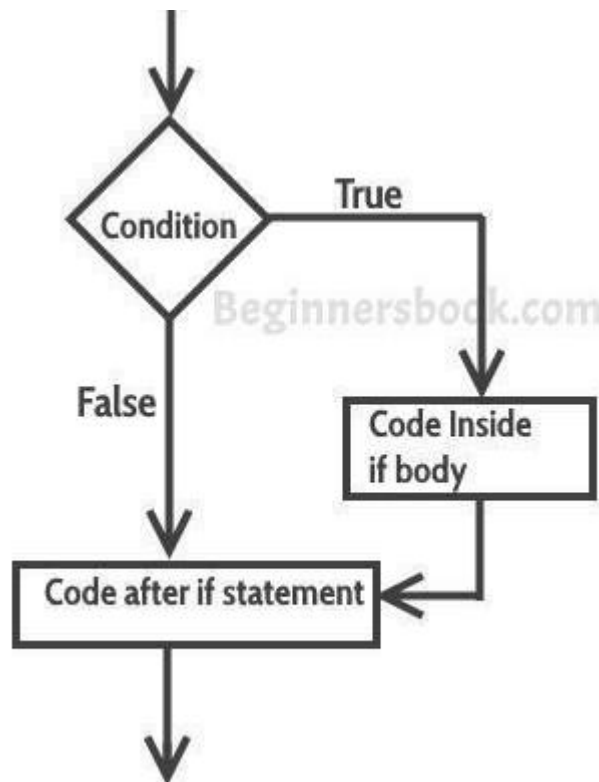
Example of if statement

```
#include
<stdio.h>intmain()
{
   int x = 20;
   int y = 22;
   if(x<y)
   {
      printf("Variable x is less than y");
   }
   return 0;
}
```

Output:
Variable x is less than y

Explanation: The condition (x<y) specified in the ―if‖ returns true for the value of x and y, so the statement inside

the body of „if‟ is executed.

If else statement

Syntax of if else statement:

If condition returns true then the statements inside the body of ―if‖ are executed and the statements inside body of
―else‖ are skipped.
If condition returns false then the statements inside the body of ―if‖ are skipped and the statements in ―else‖ are
executed.

```
if(condition) {
  // Statements inside body of if
}
else {
  //Statements inside body of else
}
```


Flow diagram of if else statement
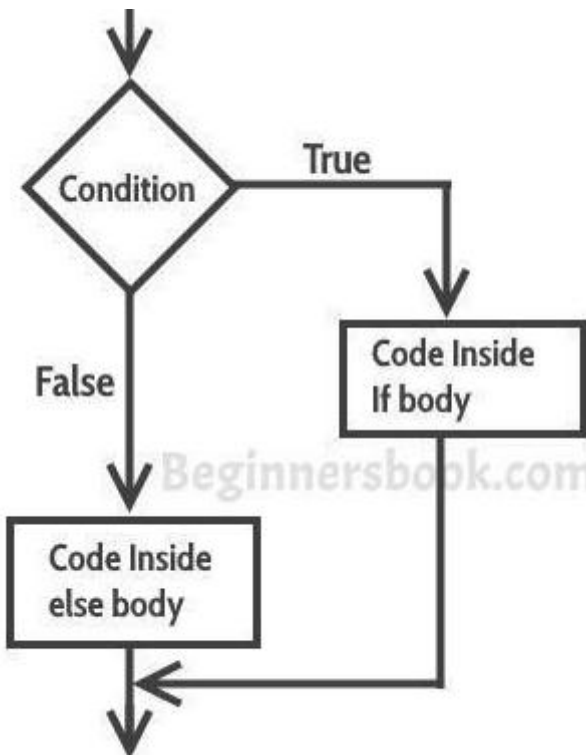
Example of if else statement

In this program user is asked to enter the age and based on the input, the if..else statement checks whether the entered age is greater than or equal to 18. If this condition meet then display message ―You are eligible for voting‖, however if the condition doesn't meet then display a different message ―You are not eligible for voting‖.

```c
#include <stdio.h> int
main()
{
    int age;
    printf("Enter your age:");
    scanf("%d",&age);
    if(age >=18)
            printf("You are eligible for voting");
    else
            printf("You are not eligible for voting"); return 0;}
```

Nested If..else statement

When an if else statement is present inside the body of another ―if‖ or ―else‖ then this is called nested if else.

Syntax of Nested if else statement:

```
if(condition) {
   //Nested if else inside the body of "if"
   if(condition2) {
     //Statements inside the body of nested "if"
   }
   else {
     //Statements inside the body of nested "else"
   }
}
else {
   //Statements inside the body of "else"
}
```

Example of nested if..else

```c
#include <stdio.h>
int main()
{
  int var1, var2;
  printf("Input the value ofvar1:");
  scanf("%d", &var1);
  printf("Input the value ofvar2:");
  scanf("%d",&var2);
  if (var1 != var2)
  {
          printf("var1 is not equal to var2\n");
          //Nested if else
          if (var1 > var2)
          {
          printf("var1 is greater than var2\n");
          }
          else
          {
                  printf("var2 is greater than var1\n");
          }
  }
  else
  {
          printf("var1 is equal to var2\n");
  }
  return 0;}
```

Output:

```
Input the value of var1:12
Input the value of var2:21
var1 is not equal to var2
var2 is greater than var1
```

else..if statement

The else..if statement is useful when you need to check multiple conditions within the program, nesting of if-else blocks can be avoided using else..if statement.

Syntax of else..if statement:

```
if (condition1)
{
  //These statements would execute if the condition1 is true
}
else if(condition2)
{
  //These statements would execute if the condition2 is true
}
else if (condition3)
{
  //These statements would execute if the condition3 is true
}
.
.
else
{
  //These statements would execute if all the conditions return false.
}
```

Example of else..if statement

Lets take the same example that we have seen above while discussing nested if..else. We will rewrite the same program using else..if statements.

```
#include <stdio.h>
int main()
{
  int var1, var2;
  printf("Input the value of var1:");
  scanf("%d", &var1);
```

```
    printf("Input the value of var2:");
    scanf("%d",&var2);
    if (var1 !=var2)
    {
            printf("var1 is not equal to var2\n");
    }
    else if (var1 > var2)
    {
            printf("var1 is greater than var2\n");
    }
    else if (var2 > var1)
    {
            printf("var2 is greater than var1\n");
    }
    else
    {
            printf("var1 is equal to var2\n");
    }
    return 0;
}
```
Output:

```
Input the value of var1:12
Input the value of var2:21
var1 is not equal to var2
```

Switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows −

```
switch(expression) {

case constant-expression :

    statement(s);

    break; /* optional */

case constant-expression :

    statement(s);

    break; /* optional */

  /* you can have any number of case statements */
```

```
  default : /* Optional */

  statement(s);

}
```

The following rules apply to a **switch** statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and colon.

- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or literal.

- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.

- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow   *Diagram*

52
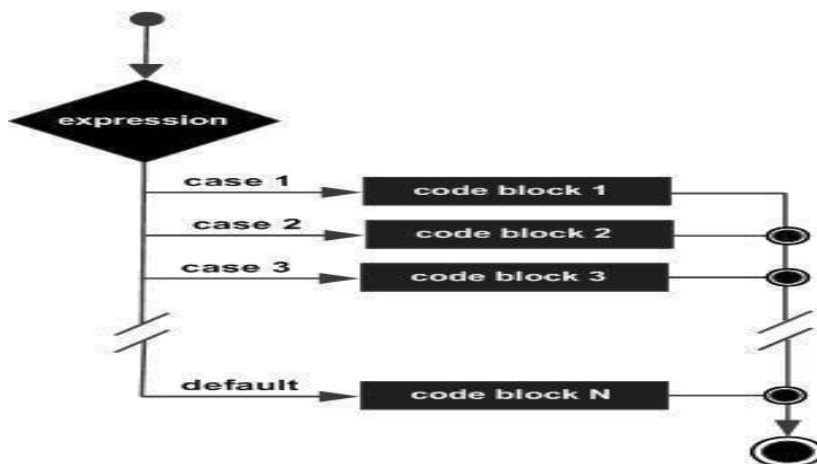
Example:

```c
#include <stdio.h>

int main () {
/* local variable definition */
  char grade = 'B';

  switch(grade) {
    case 'A' :
      printf("Excellent!\n" );
      break;
    case 'B':
    case 'C':
      printf("Well done\n" );
      break;
    case 'D' :
      printf("You passed\n" );
      break;
    case 'F':
      printf("Better try again\n");
      break;
    default:
      printf("Invalid grade\n" );
  }

  printf("Your grade is %c\n", grade );

  return 0;
}
```

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

```
Well done
```

Your grade is B

Loop Control Statements: While, do-while, for

While loop

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.
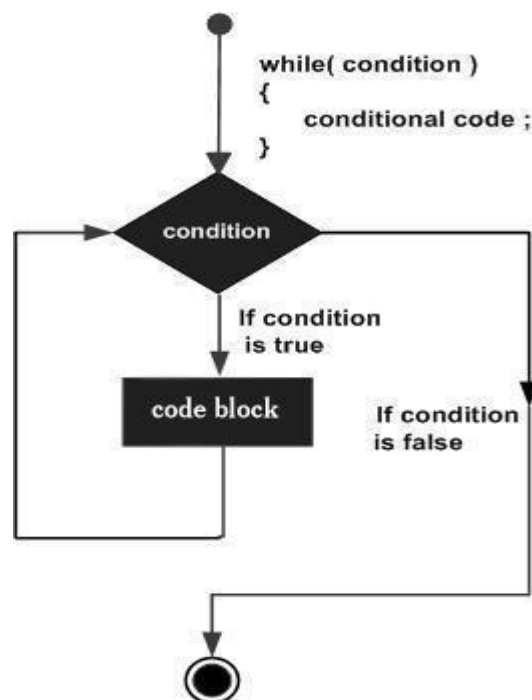
Syntax

The syntax of a **while** loop in C programming language is –

```
while(condition) {
  statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control
passes to the line immediately following the loop.

Flow Diagram

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```c
#include <stdio.h>

 int main () {
/* local variable definition */

  int a = 10;
 /* while loop execution */

  while( a < 20 ) {

    printf("value of a: %d\n", a);

    a++;

  }

 return 0;}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Do-while loop

A do while loop is similar to while loop with one exception that it executes the statements inside the body of do-while before checking the condition. On the other hand in the while loop, first the condition is checked and then the statements in while loop are executed. So you can say that if a condition is false at the first place then the do while would run once, however the while loop would not run at all.
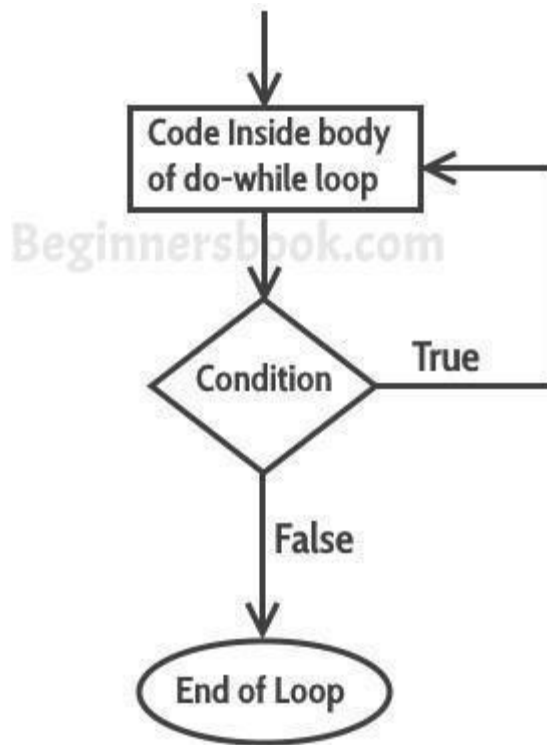
C – do..while loop

Syntax of do-while loop

```c
do
{
   //Statements
```

}while(condition test);

Flow diagram of do while loop

Example of do while loop

```c
#include <stdio.h>
int main()
{
        int j=0;
        do
        {
                printf("Value of variable j is: %d\n", j);
                j++;
        }while (j<=3);
        return 0;
}
```
Output:

Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3

Program to print table for the given number using do while loop

```c
1.  #include<stdio.h>
2.  int main(){
3.  int i=1,number=0;
4.  printf("Enter a number:");
5.  scanf("%d", &number);
6.  do{
7.  printf("%d\n",(number*i));
8.  i++;
9.  }while(i<=10);
10. return 0;
11. }
```

Output

```
Enter a number: 5
5
10
15
20
25
30
35
40
45
50

Enter a number: 10
10
20
30
40
50
60
70
80
90
100
```

For loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

57

Syntax

The syntax of a **for** loop in C programming language is –

```
for ( init; condition; increment )
{
   statement(s);
}
```

Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for _loop.

- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Flow Diagram

```
for( init; condition; increment )
{
    conditional code ;
}
```
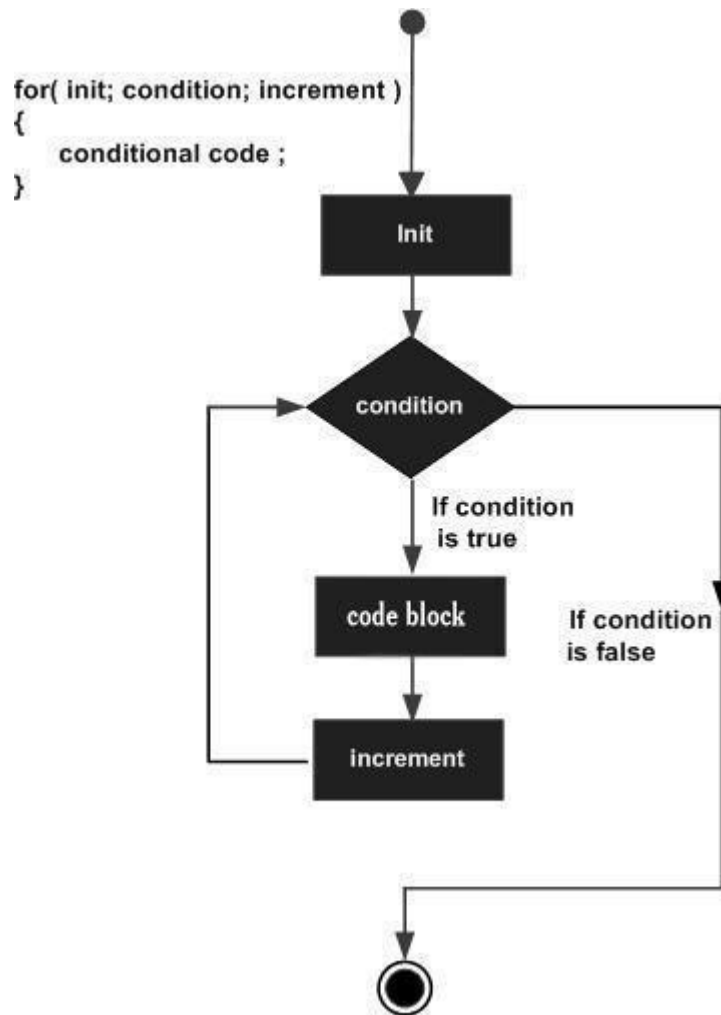

Example

```c
#include <stdio.h>

 int main () {

int a;

       /* for loop execution */

   for( a = 10; a < 20; a = a + 1 ){

 printf("value of a: %d\n", a);

   }

 return 0;

}
```

59

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Jump statements-break , continue and goto

Break statement

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

Break in For Loop :

for(initialization ; condition ; incrimination)

{

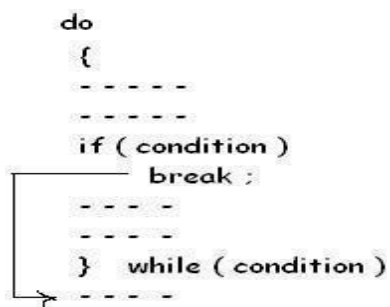statement1;

Statement2;

Break;

}

Break in While Loop :

initialization:
while(condition)
{
statement1;
statement2;
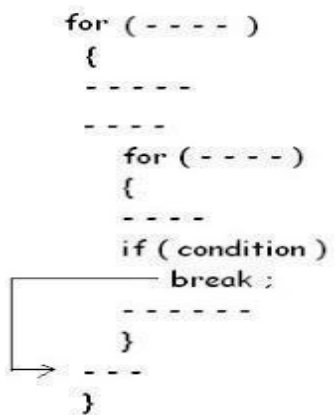incrimination
break;
}

Break Statement in Do-While :

Initialization:
do
{
statement1;
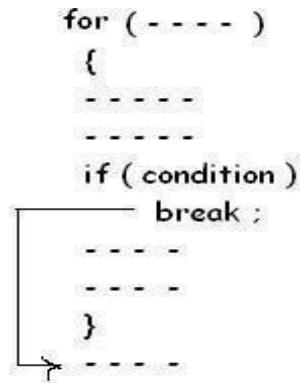statement2;
incrimination
break;
}while(condition);

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 ;
  - - - -
  - - - -
  }
  - - - -
```
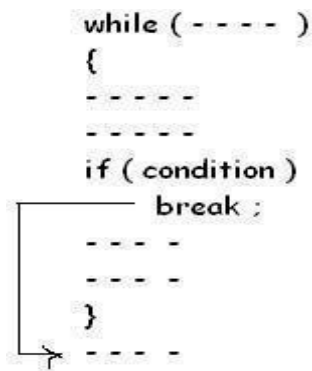
<u>Continue statement</u>:

```
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.

Types of

Go to Forward

Backward

```
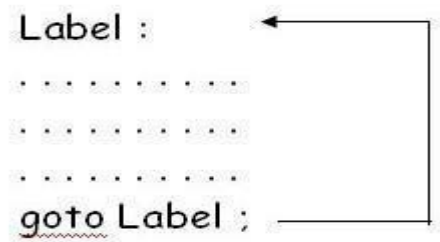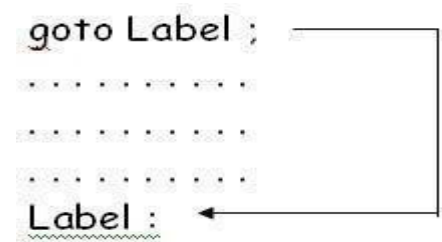goto Label ;  ─────────┐
. . . . . . . . . . .   │
. . . . . . . . . . .   │
. . . . . . . . . . .   │
Label :  ◄──────────────┘
```

```
Label :  ◄──────────┐
. . . . . . . . . .   │
. . . . . . . . . .   │
. . . . . . . . . .   │
goto Label ;  ────────┘
```

MODULE -III

ARRAYS AND FUNCTIONS

ARRAY

C Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.

- Array might be belonging to any of the datatypes
- Array size must be a constantvalue.
- Always, Contiguous (adjacent) memory locations are used to store array elements inmemory.
- It is a best practice to initialize an array to zero or null while declaring, if we don"t assign any values toarray.

Example for C Arrays:

int a[10];   // integer array
char b[10];   // character array i.e. string

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 C ARRAYS:

1. Static Array
2. Dynamic Array

Types of Static Array

1. One dimensional array
2. Multi dimensional array
    - o   Two dimensional array
    - o   Three dimensional array
    - o   four dimensional arrayed…

One Dimensional Array :

Syntax : data-type arr_name[array size];

Array declaration, initializationandaccessing	Example
Arraydeclarationsyntax: data_type arr_name [arr_size];**Array initialization syntax:** data_type arr_name [arr_size]=(value1, value2, value3,….);**Array accessing syntax:** arr_name[index];	**Integer arrayexample:** int age [5]; int age[5]={0, 1, 2, 3, 4}; age[0]; /*0 is accessed*/ age[1]; /*1 is accessed*/ age[2]; /*2 isaccessed*/ **Character array example:** char str[10]; char str[10]={,,H",,,a",,,i"}; (or) char str[0] = ,,H"; char str[1] = ,,a"; char str[2] = ,,i; str[0]; /*H is accessed*/ str[1]; /*a is accessed*/ str[2]; /*i is accessed*/

EXAMPLE PROGRAM FOR ONE DIMENSIONAL ARRAY IN C:

```c
#include<stdio.h>

int main()
{
    int i;
    int arr[5] = {10,20,30,40,50};

    // declaring and Initializing array in C
    //To initialize all array elements to 0, use int arr[5]={0};
    /* Above array can be initialized as below
    also arr[0] =10;
    arr[1] =20;
    arr[2] =30;
    arr[3] = 40;
    arr[4] = 50;*/

    for(i=0;i<5;i++)
    {
        // Accessing each variable
        printf("value of arr[%d] is %d \n", i, arr[i]);
    }
```

}

value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is30
value of arr[3] is 40
value of arr[4] is50

Two dimensional array

The two dimensional array in C language is represented in the form of rows and columns, also known as matrix. It is also known as *array of arrays* or *list of arrays*.

The two dimensional, three dimensional or other dimensional arrays are also known as *multidimensional* arrays.

Declaration of two dimensional Array in C

We can declare an array in the c language in the following way.

data_type array_name[size1][size2];

A simple example to declare two dimensional array is given below.

int two dimen[4][3];

Here, 4 is the *row* number and 3 is the *column* number.

Initialization of 2D Array in C

A way to initialize the two dimensional array at the time of declaration is given below.

int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

Two dimensional array example in C

```
#include<stdio.h>
int main(){
inti=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

68

```
//traversing 2D array
for(i=0;i<4;i++){
for(j=0;j<3;j++){
 printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
}//end ofj
}//end ofi
return 0;
}
```

Output

```
arr[0][0] =1
arr[0][1] =2
arr[0][2] =3
arr[1][0] =2
arr[1][1] =3
arr[1][2] =4
arr[2][0] =3
arr[2][1] =4
arr[2][2] =5
arr[3][0] =4
arr[3][1] =5
arr[3][2] =6
```

Strings

 In C programming, array of characters or collection of characters is called a string. A string always recognized in double quotes. A string is terminated by a null character /0. For example:

 .

―String‖

Here, ―String‖ is a string. When, compiler encounters strings, it appends a null character /0 at the end of string.

S	T	R	I	N	G	\0

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 second name:‖);
scanf(-%s‖,&str2) ;
puts(str1);
puts(str2);
}
```

String Handling 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.   strcpy().

7.   strrev().

1. strlen()

```
size_t strlen(const char *str);
```

The function takes a single argument, i.e, the string variable whose length is to be found, and returns the length of the string passed.

The strlen() function is defined in <string.h>header file.

Example

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

```c
   char a[20]=‖Program‖
   char b[20]={„P‟,‟r‟,‟o‟,‟g‟,‟r‟,‟a‟,‟m‟,‟\0‟};
   char c[20];
   printf(―Enter string: ―);
   gets(c);
   printf(―Length of string a = %d \n‖, strlen(a));
   //calculates the length of string before null charcter.
   printf(―Length of string b = %d \n‖, strlen(b));
   printf(―Length of string c = %d \n‖, strlen(c));
   return 0;
}
```

Output

Enter string: String

Length of string a = 7
Length of string b = 7
Length of string c = 6

2. strupr()

strupr( ) function converts a given string into uppercase. Syntax for strupr( ) function is given below.

char *strupr(char *string);

Example

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

int main()
{
   char str[ ] = "Modify This String To Upper";
   printf("%s\n",strupr(str));
   return 0;
}
```

OUTPUT:
MODIFY THIS STRING TO UPPER

3. strlwr()

strlwr( ) function converts a given string into lowercase. Syntax for strlwr( ) function is given below.

char *strlwr(char *string);

Example

```
#include<stdio.h>
#include<string.h>
int main ()
{
   char str[ ] = "MODIFY This String To Lower";
   printf("%s\n",strlwr (str));
   return 0;
}
```

OUTPUT:
modify this string to lower

4. strcmp()

- strcmp( ) function in C compares two given strings and returns zero if they are same.

- If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns >0 value. Syntax for strcmp( ) function is given below.

int strcmp (const char * str1, const char * str2);

- strcmp( ) function is case sensitive. i.e, ―A‖ and ―a‖ are treated as different characters.

Example

```
#include <stdio.h>
#include <string.h>
int main( )
{
  char str1[ ] = "fresh" ;
  char str2[ ] = "refresh" ;
  int i, j, k ;
  i = strcmp ( str1, "fresh" ) ;
  j = strcmp ( str1, str2 );
  k = strcmp ( str1, "f" );
  printf ( "\n%d %d %d", i, j, k ) ;
  return 0;
}
```

0 -1 1

5. strcat()

- strcat( ) function in C language concatenates two given strings. It concatenates source string at the end of destination string. Syntax for strcat( ) function is given below

.
char * strcat (char * destination, const char * source);

- Example:
  strcat (str2, str1); – str1 is concatenated at the end of str2.
  strcat (str1, str2); – str2 is concatenated at the end of str1.

- As you know, each string in C is ended up with null character(„\0").

- In strcat( ) operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strcat( )operation.

Example

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

int main( )
{
  char source[ ] = " fresh2refresh" ;
  char target[ ]= " C tutorial" ;

  printf ( "\nSource string = %s", source ) ;
  printf ( "\nTarget string = %s", target ) ;

  strcat ( target, source ) ;

  printf ( "\nTarget string after strcat( ) = %s", target ) ;
}
```
OUTPUT:
Sourcestring   =fresh2refresh
Targetstring   = Ctutorial
Target string after strcat( ) = C tutorial fresh2refresh

6. strcpy().

- strcpy( ) function copies contents of one string into another string. Syntax for strcpy function id given below.
  char * strcpy (char * destination, const char * source);

- Example:
  strcpy (str1, str2) – It copies contents of str2 into str1.
  strcpy (str2, str1) – It copies contents of str1 into str2.

- If destination string length is less than source string, entire source string value won't be copied into destination string.

Example

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

int main( )
{
  char source[ ] = "fresh2refresh" ;
  char target[20]= "" ;
  printf ( "\nsource string = %s", source ) ;
  printf ( "\ntarget string = %s", target ) ;
  strcpy ( target, source ) ;
  printf ( "\ntarget string after strcpy( ) = %s", target ) ;
  return 0;
}
```

OUTPUT:

source string = fresh2refresh
target string =
target string after strcpy( ) = fresh2refresh

7. strrev()

strrev( ) function reverses a given string in C language. Syntax for strrev( ) function is given below.

char *strrev(char *string);

Example

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

int main()
{
  char name[30] = "Hello";

  printf("String before strrev( ) : %s\n",name);

  printf("String after strrev( ) : %s",strrev(name));

  return 0;
}
```

String before strrev( ) : Hello
String after strrev()   :olleH

Arrays of strings

A string is a 1-D array of characters, so an array of strings is a 2-D array of characters.

Just like we can create a 2-D array of int, float etc; we can also create a 2-D array of character or array of strings. Here is how we can declare a 2-D array of characters.

char ch_arr[3][10]={
                       {„s",„p",„i",„k",„e",„\0"},
                       { „t",„o",„m",„\0"},
                       {„j",„e",„r",„r",„y",„\0"}};

It is important to end each 1-D array by the null character otherwise, it's just an array of characters. We can't use them as strings.

Declaring an array of string this way is a tedious and error-prone process that's why C provides a more compact way to it. This above initialization is equivalent to:

char ch_arr[3][10]={−spike‖,‖tom‖,‖jerry‖};

The following program demonstrates how to print an array of strings.

```c
#include<stdio.h>
int main()
{
   int i;
  char ch_arr[3][10] = {
                   "spike",
                   "tom",
                   "jerry"
                };

   printf("1st way \n\n");

   for(i = 0; i < 3; i++)
   {
     printf("string = %s \t address = %u\n", ch_arr + i, ch_arr + i);
   }

   // signal to operating system program ran fine
   return 0;
}
```

Expected output

string = spike address = 2686736
string = tom address = 2686746
string = jerry address = 2686756"

We already know that the name of an array is a constant pointer so the following operations are invalid.

ch_arr[0] = "tyke"; // invalid
ch_arr[1] = "dragon"; // invalid

Here we are trying to assign a string literal (a pointer) to a constant pointer which is obviously not possible.
To assign a new string to ch_arr use the following methods.

strcpy(ch_arr[0], "type"); // valid
scanf(ch_arr[0], "type"); // valid

Introduction to functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows −

```
    return_type function_name( parameter list ) {

  body of the function

}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */

int max(int num1, int num2) {

/* local variable declaration */

  int result;

if (num1 > num2)

   result = num1;

 else

   result = num2;

return result;

}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), the function declaration is as follows −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

returnType functionName(type1 argument1, type2 argument2,...);

In the above example, `int add Numbers(int a, int b);` is the function prototype which provides following information to the compiler:

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the `main ()` function.

Category of functions:

A function depending an whether the arguments are present or not and whether a value is returned or not, may belong to one of following categories

1. Function with no return values, no arguments
2. Functions with arguments, no return values

3. Functions with arguments and return values
4. Functions with no arguments and return values.

Function with no return values, no arguments

In this category, the function has no arguments. It does not receive any data from the calling function. Similarly, it doesn't return any value. The calling function doesn't receive any data from the called function. So, there is no communication between calling and called functions.

Functions with arguments, no return values

In this category, function has some arguments. It receives data from the calling function, but it doesn't return a value to the calling function. The calling function doesn't receive any data from the called function. So, it is one way data communication between called and calling functions.

Eg: Printing n Natural numbers

```
#include<stdio.h>
#include<conio.h>
void nat( int);
voidmain()
{
int n;
printf("\n Enter n value:");
scanf("%d",&n);
nat(n);
}
void nat(int n)
{
int i;
for(i=1;i<=n;i++)
printf("%d\t",i);
}
```

Note:
In the main() function, n value is passed to the nat() function. The n value is now stored in the formal argument n, declared in the function definition and subsequently, the natural numbers up to n are obtained.

Functions with arguments and return values

In this category, functions have some arguments and it receives data from the calling function. Similarly, it returns a value to the calling function. The calling function receives data from the

called function. So, it is two-way data communication between calling and called functions.

Eg. Factorial of a Number

```
#include<stdio.h>
#include<conio.h>
intfact(int);
voidmain()
{
int n;
printf("\n Enter n:");
scanf("%d",&n);
printf("\n Factorial of the number : %d", fact(n));

}
int fact(int n)
{
int i,f;
for(i=1,f=1;i<=n;i++)
f=f*i;
return(f);
}
```

Functions with no arguments and return values.

In this category, the functions have no arguments and it doesn't receive any data from the calling function, but it returns a value to the calling function. The calling function receives data from the called function. So, it is one way data communication between calling and called functions.

Eg. Sum of Numbers

```
#include<stdio.h>
#include<conio.h>
int sum();
void main()
{
int s;
printf("\n Enter number of elements to be added :");
s=sum();
printf("\n Sum of the elements :%d",s);

}
```

```
int sum()

{
int a[20], i, s=0,n;
scanf("%d",&n);
printf("\n Enter theelements:");
for(i=0;i< n; i++)
scanf("%d",&a[i]);
for(i=0;i< n; i++)
s=s+a[i];
return s;
}
```

Inter Function communication

When a function gets executed in the program, the execution control is transferred from calling function to called function and executes function definition, and finally comes back to the calling function. In this process, both calling and called functions have to communicate each other to exchange information. The process of exchanging information between calling and called functions is called as inter function communication.

In C, the inter function communication is classified as follows...

- Downward Communication
- Upward Communication
- Bi-directional Communication

Downward Communication

In this type of communication, the data is transferred from calling function to called function but not

from called function to calling function. The function with parameters and without return value is

considered under Downward communication.

Example

```
#include <stdio.h>
#include<conio.h>
void main(){
  int num1, num2 ;
  void addition(int, int) ; // function declaration
  clrscr() ;
  num1 = 10 ;
  num2 = 20 ;

  printf("\nBefore swap: num1 = %d, num2 = %d", num1, num2) ;
  addition(num1, num2) ; // calling function

  getch() ;
}
```

81

```c
void addition(int a, int b) // called function
{

    printf("SUM = %d", a+b) ;

}
```

Output

SUM=30

Upward Communication

In this type of communication, the data is transferred from called function to calling function but not

from calling function to called function. The function without parameters and with return value is

considered under upward communication.

Example
```c
#include <stdio.h>
#include<conio.h>
void main(){
    int result ;
    int addition() ; // function declaration
    clrscr() ;

    result = addition() ; // calling function

    printf("SUM = %d", result) ;
    getch() ;
}

int addition() // called function
{
    int num1, num2 ;
    num1 =10;
    num2 =20;
    return (num1+num2) ;
}
```

Output

SUM=30

Bi-Directional Communication

In this type of communication, the data is transferred from called function to calling function and also

from calling function to called function. The function with parameters and with return value is considered

under Bi-Directional communication.

82

Example

```
#include <stdio.h>
#include<conio.h>
void main(){
  int num1, num2, result ;
  int addition(int, int) ; // function declaration
  clrscr() ;

  num1 = 10 ;
  num2 = 20 ;

  result = addition(num1, num2) ; // calling function

  printf("SUM = %d", result) ;
  getch() ;
}
int addition(int a, int b) // called function
{
  return (a+b) ;
}
```

Output

SUM=30

Function Calls

This calls the actual function

Syntax:

function_name (arguments list);

There are two ways that a C function can be called from a program. They are,

1. Call byvalue
2. Call byreference

Call by Value

- In call by value method, the value of the variable is passed to the function as parameter.

- The value of the actual parameter cannot be modified by formal parameter.

- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

83

Note:

- Actual parameter – This is the argument which is used in function call.

- Formal parameter – This is the argument which is used in function definition

Call by Reference:

- In call by reference method, the address of the variable is passed to the function as parameter.

- The value of the actual parameter can be modified by formal parameter.

- Same memory is used for both actual and formal parameters since only address is used by both parameters.

Parameter Passing Mechanisms:

In C Programming we have different ways of parameter passing schemes such as Call by Value and Call by Reference.

Function is good programming style in which we can write reusable code that can be called whenever require.

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called argument.

Two Ways of Passing Argument to Function in C Language :

A. Call by Reference

B. Call by Value

Let us discuss different ways one by one –

 A. Call by Value:

```
#include<stdio.h>
void interchange(int number1,int number2)
{
int temp;
temp = number1;
number1 = number2;
number2 = temp;
}
int main() {

int num1=50,num2=70;
interchange(num1,num2);
printf("\nNumber 1 : %d",num1);
printf("\nNumber 2 : %d",num2);
```

return(0);
}

Output:
Number 1 :50
Number 2 :70

Explanation: Call by Value

While Passing Parameters using call by value , Xerox copy of original parameter is created and passed to the called function.
Any update made inside method will not affect the original value of variable in calling function.

In the above example num1 and num2 are the original values and Xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively. As their scope is limited to only function so they cannot alter the values inside main function.
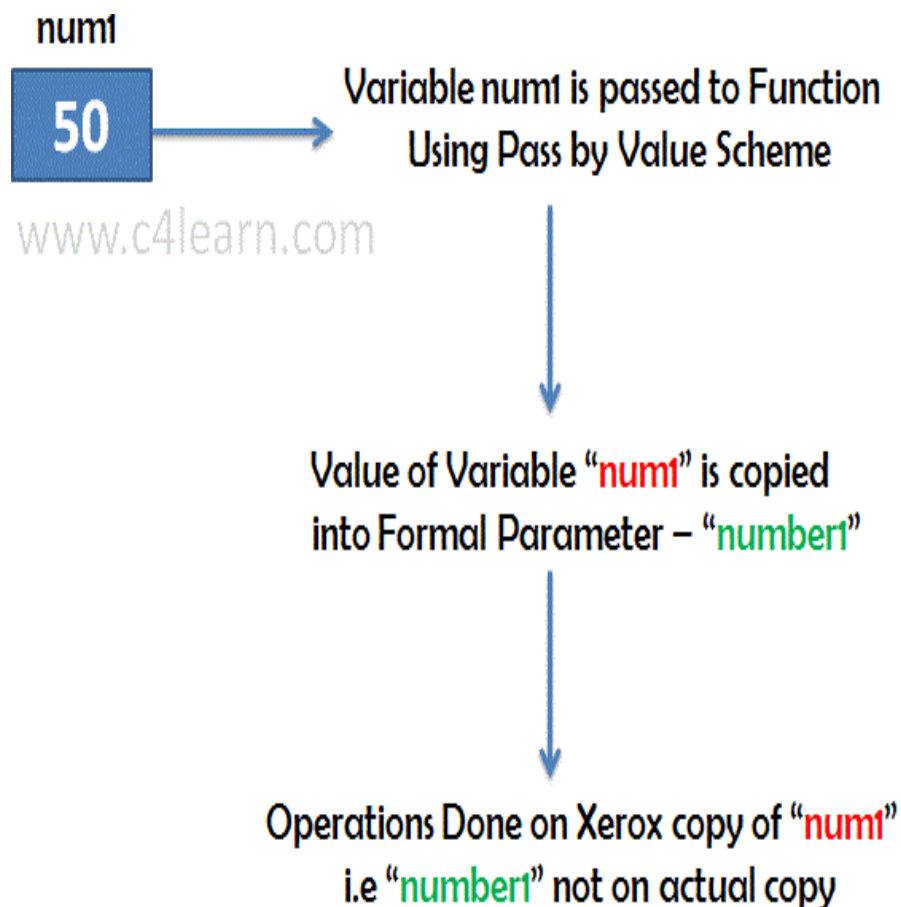
**B. Call by Reference/Pointer/Address:**

```c
#include<stdio.h>
void interchange(int *num1,int *num2)
{
int temp;
temp = *num1;
*num1 = *num2;
*num2 = temp;
}
int main() {
int num1=50,num2=70;
interchange(&num1,&num2);
printf("\nNumber 1 : %d",num1);
printf("\nNumber 2 : %d",num2);
return(0);
}
```

Output :
Number 1 :70
Number 2 :50

Call by Address

While passing parameter using call by address scheme, we are passing the actual address of the variable to the called function.

Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.

Summary of Call By Value and Call By Reference :

Point	Call by Value	Call by Reference
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

Recursion

The process of calling a function by itself is called recursion and the function which calls itself is called recursive function. Recursion is used to solve various mathematical problems by dividing it into smaller problems.
Syntax of Recursive Function

```
return_type recursive_func ([argument list])
{
 statements;
    ... ... ...
    recursive_func ([actual argument]);
    ... ... ...
}
```

Flowchart of Recursion

Fig: Flowchart showing recursion

Note: In order to prevent infinite recursive call, we need to define proper exit condition in a recursive function.

For example, consider the program below:

Example #1: C Program to show infinite recursive function

```c
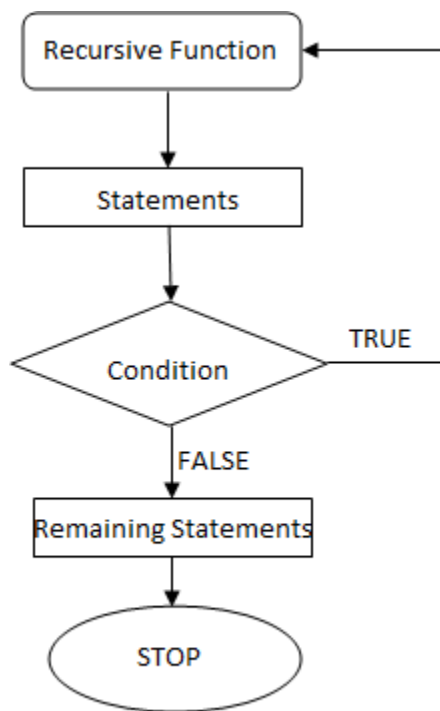#include<stdio.h>
intmain()
{
   printf("Hello world");
   main();
   return 0;
}
```

In this program, we are calling *main()* from *main()* which is recursion. But we haven't defined any

condition for the program to exit. Hence this code will print "**Hello world**" infinitely in the output screen.

Types of recursion
- DirectRecursion
- IndirectRecursion

Direct Recursion

A function is said to be direct recursive if it calls itself directly.

Example #2: C Program Function to show direct recursion

```c
int fibo (int n)
{
   if (n==1 || n==2)
      return 1;
   else
      return (fibo(n-1)+fibo(n-2));
}
```

In this program, *fibo()* is a direct recursive function. This is because, inside *fibo()* function, there
is a statement which calls *fibo()* function again directly.

Indirect Recursion

A function is said to be indirect recursive if it calls another function and this new function calls
the first calling function again.

Example #3: C Program Function to show indirect recursion

```
int func1(int n)
{
    if (n<=1)
        return 1;
    else
        return func2(n);
}

int func2(int n)
{
    return func1(n);
}
```

Passing Array to a Function:

Whenever we need to pass a list of elements as argument to any function in C language, it is preferred to do so using an array.

Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

We can either have an array as a Parameter

int sum (int arr[]);

Or, we can have a pointer in the parameter list, to hold the base address of our

array. int sum (int* ptr);

Returning an Array from a function

We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned.

```
int* sum (intx[])
{
// statements return x ;
}
```

Passing a single array element to a function(Call by value)

In this type of function call, the actual parameter is copied to the formal parameters.

Example 1:

```
#include<stdio.h>
void giveMeArray(int a);
int main()
{
int myArray[] = { 2, 3, 4 };
giveMeArray(myArray[2]);
return 0;
}
void giveMeArray(int a)
{
printf("%d", a);
}
```

Output: 4

Example 2:

```
#include <stdio.h> void disp( char ch)
{
printf("%c ", ch);
}
int main()
{
char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
for (int x=0; x<10; x++)
{
disp (arr[x]);
}
return 0;
}
OUTPUT: a b c d e f g h i j
```

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```c
#include <stdio.h>
void disp( int *num)
{
printf("%d ", *num);
}

int main()
{
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8,9, 0};
for (int i=0; i<10;i++)
{
/* Passing addresses of array elements*/
disp (&arr[i]);
}
return 0;
}
```

OUTPUT:

1 2 3 4 5 6 7 8 9 0

Passing a complete One-dimensional array to a function

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```c
#include<stdio.h>
float findAverage(int marks[]);
int main()
{
float avg;
int marks[] = {99, 90, 96, 93, 95};
avg = findAverage(marks);
printf("Average marks = %.1f", avg);
return 0;
}
float findAverage(int marks[])
{
int i, sum = 0;
float avg;
for (i = 0; i <= 4; i++)
{
sum += age[i];
}
avg = (sum / 5);
return avg;
}
```
Output: 94.6
assing a Multi-dimensional array to a function

```c
#include<stdio.h>
void displayArray(int arr[3][3]); int main()
{
int arr[3][3], i, j;
printf("Please enter 9 numbers for the array: \n");
for (i = 0; i < 3;++i)
{
for (j = 0; j < 3;++j)
{
scanf("%d", &arr[i][j]);
}
}
// passing the array as argument displayArray(arr);
return 0;
}
void displayArray(int arr[3][3])
{
int i, j;
printf("The complete array is:\n");
for (i = 0; i < 3;++i)
{
// getting cursor to new lineprintf("\n");
for (j = 0; j < 3;++j)
{
// \t is used to provide tab space printf("%4d", arr[i][j]);
}
}
}
```

Output:
Please enter 9 numbers for the array: 1 2 3 4 5 6 7 8 9
The complete array is: 1 2 3 4 5 6 7 8 9

Passing string to function

- 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 andc++.

- 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 character.

- **Example: " GauravArora"**

- The above string contains 12characters.

- **Example: Charar[20]**

- The above example will store 19 character with I null character.

Strings are just char arrays. So, they can be passed to a function in a similar manner as arrays.

```
#include <stdio.h>
void displayString(char str[]);
int main()
{
char str[50]; printf("Enter string: ");
gets(str);
displayString(str);
// Passing string c to function. return 0;
}
void displayString(char str[])
{
printf("String Output: ");
 puts(str);
}
```

Here, string c is passed from main() function to user-defined function

displayString(). In function declaration, str[] is the formal argument

```
Example-2 :
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void strreverse(char *string)
{
int i, len;
char c;
len=strlen(string);
charstring2[len+1];

for(i=0; i<len;i++)
{
c=string[i];
```

```
string2[len-i]=c;

}
string2[len+1]='\0';
string=string2;
//printf("%s\n", string);
}


int main(int argc, char *argv[])
{
char str[256];
printf("Type a String to reverse it.[max. 255 chars]\n");
fgets(str, 255, stdin);
strreverse(&str[0]);
printf("%s", str);
return 0;
}
```

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

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.

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.

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.

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);
}
```

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);
}
```

Static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life- time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

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

```
#include <stdio.h>

void func(void);
static int count = 5; /* global variable
*/ main() {
while(count--)
{
func();
}
return 0;
}
  void func( void )
{
static int i = 5;
 /* local static variable*/
i++;
printf("i is %d and count is %d\n", i, count);
}
```

Register Storage Class

register keyword is used to define local variable.

Local variable are stored in register instead of

RAM.

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

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

RAM instead it is stored in Register.

This is generally used for

faster access.

 Communisis-**Counter**-

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=num—num2;
printf("\nSum of Numbers :%d",sum);

return(0);
}
```

Preprocessor directives

The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation (Preprocessor directives are executed before compilation.). It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, #define is the directive that defines a macro. Whitespace is also allowed before and after the #.

The # and the directive name cannot come from a macro expansion. For example, if foo is defined as a macro expanding to define, that does not make #foo a valid preprocessing directive.

All preprocessor directives starts with hash # symbol.

 List of preprocessor directives :

1. #include
2. #define
3. #undef
4. #ifdef
5. #ifndef
6. #if
7. #else
8. #elif
9. #endif
10. #error
11. #pragma

1. #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system- defined and user-defined header files. If included file is not found, compiler renders error. It has three variants:

#include <file>

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories.

#include "file"

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file.

#include anything else

This variant is called a computed #include. Any #include directive whose argument does not fit the above two forms is a computed include.

2. Macro's(#define)

Let's start with macro, as we discuss, a macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Syntax

#define token value

There are two types of macros:

1. Object-likeMacros
2. Function-likeMacros

1. Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

#define PI 3.1415

Here, PI is the macro name which will be replaced by the value 3.14. Let's see an example of Object-like Macros
:

```
#include
<stdio.h>
#define PI
3.1415 main()
{
```

Output:

3.14000

2. Function-like Macros

The function-like macro looks like function call. For example:

#define MIN(a,b) ((a)<(b)?(a):(b)

Here MIN is the macro name. Let's see an example of Function-like Macros :

```c
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:
Minimum between 10 and 20 is: 10

Preprocessor Formatting

A preprocessing directive cannot be more than one line in normal circumstances. It may be
split cosmetically with Backslash-Newline. Comments containing Newlines can alsodivide
the directive into multiple lines.
for example, you can split a line cosmetically with Backslash-Newline anywhere:

is equivalent into #define FOO 1020.

3. #undef

To undefined a macro means to cancel its definition. This is done with the

#undef directive. Syntax:

#undef token

define and undefine example

```c
#include <stdio.h>
#define PI 3.1415
#undef PI
main()
{ printf("%f",PI);
}
```

Compile Time Error: 'PI' undeclared

4. #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it

executes the code.

Syntax:

```
#ifdef MACRO
//code
#endif
```

5. #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it

Syntax:

```
#ifndef MACRO
//code
#endif
```

6. #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it

executes the code.

Syntax:

```
#if expression
//code
#endif
```

7. #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression
//if code
#else
//else code
#endif
```

Syntax with #elif

```
#if expression
//if code
#elif expression
//elif code
#else
//else code
#endif
```

8. #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

```
#include<stdio.h>
#ifndef MATH_H
#error First include then compile #else
void main(){ float a; a=sqrt(7);
printf("%f",a);
}
#endif
```

9. #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature. Different compilers can provide different usage of #pragma directive.

Syntax:
#pragma token

STRUCTURES, UNIONS AND POINTERS

Need of structures

Structure is a collection of variables of different types under a single name.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2,salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

Structure Definition in C

Keyword structis used for creating a structure.

Syntax of structure

struct structure_name

{

data_type member1;

data_type member2;

.

.

data_type memeber;};

Note: Don't forget the semicolon }; in the ending line.

We can create the structure for a person as mentioned above as:

struct person

{

char name[50];
int citNo;
float salary;

};

This declaration above creates the derived data type struct person.

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated. For the above structure of a person, variable can be declared as:

struct person

{

char name[50];
int citNo;
float salary;

};
int main()

{

struct person person1, person2, person3[20];
return 0;
}

Another way of creating a structure variable is

struct person

{

char name[50];
int citNo;
float salary;

} person1, person2, person3[20];

In both cases, two variables person1, person2 and an array person3 having 20 elements of type **struct person** are created.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

     1. Member operator(.)

      2. Structure pointer operator(->) (is discussed in structure and pointers tutorial)

      3. Any member of a structure can be accessed as:

structure_variable_name.member_name

Suppose, we want to access salary for variable person2. Then, it can be accessed as:

    person2.salary

Example of structure

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)

```c
#include <stdio.h> struct Distance
{
int feet; floatinch;
} dist1, dist2,sum;


int main()
{
printf("1st distance\n");
// Input of feet for structure variable dist1 printf("Enter feet:");
scanf("%d",&dist1.feet);


// Input of inch for structure variable dist1 printf("Enter inch: ");
scanf("%f",&dist1.inch);
```

106

```
printf("2nd distance\n");


// Input of feet for structure variable dist2 printf("Enter feet:");
scanf("%d",&dist2.feet);


// Input of feet for structure variable dist2 printf("Enter inch: ");
scanf("%f",&dist2.inch);
sum.feet = dist1.feet + dist2.feet; sum.inch = dist1.inch + dist2.inch;

if (sum.inch > 12)
{
//If inch is greater than 12, changing it to feet.
++sum.feet;
sum.inch = sum.inch - 12;
}
// printing sum of distance dist1 and dist2
printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch); return 0;
}
```

Output

1st distance Enter feet: 12
Enter inch: 7.9 2nd distance Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"

Structure Initialization

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

2. Let us discuss very familiar example of structure student , we can initialize structure

   variable in different ways–

Way1: Declare and Initialize

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

107

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

Way2: 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.

Way3: 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 membersare 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

Way4: Initializing inside main

```
    struct student
    {
    int mark1;
```

108

```
    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

Accessing StructureMembers

1. Array elements are accessed using the Subscript variable , Similarly Structuremembers

   are accessed using dot [.]operator.

2. (.) is called as –Structure member Operator‖.

3. Use this Operator in between **"Structure name"** &**"membername".**

```
#include<stdio.h>
struct stud
{

char name[20];
char fname[10];
};

struct stud s;
```

109

```
main()
{

scanf("%s%s",&s.name,&s.fname);
printf("%s%s",s.name,s.fname);
}
```

Output:

Vedha

srinivas

Vedhasrinivas

```
struct employee
    {
    char name[100];
    int age;
    float salary;
    char department[50];

    } employee_one = {"Jack", 30, 1234.5, "Sales"};

    int age = employee_one.age;

    float salary= employee_one.salary;

    char department= employee_one.department;
```

ACCESSING ARRAY OF STRUCTURE ELEMTS

```
STRUCT STUD
{
Datatype member1;
Datatype member2;
.
```

.

    } struct stud s[50];

Members of structures are accessed through dot operator. Eg:

stud[i].memeber1 Stud[i].member2

```c
#include<stdio.h>
struct stud
{
char name[20];

};

struct stud s[10];
main()
{

int i,n;
for(i=0;i<2;i++)
{

scanf("%s",&s[i].name);

printf("%s",s[i].name);

}

}
```

Output:

Swapna

Divya

Swapna

divya

Nested Structures

A structure can be nested inside another structure. In other words, the members of a

structure can be of any other type including structure.

Here is the syntax to create nested structures.

```
structure tagname_1
{
member1; member2; member3;...
Member n; structure tagname_2

{
member_1; member_2; member_3;...
member_n;
},var1
}var2;
```

To access the members of the inner structure, we write a variable name of the outer structure, followed

by a dot(.) operator, followed by the variable of the inner structure, followed by a dot(.) operator, which

is then followed by the name of the member we want to access.

var2.var1.member_1 - refers to the member_1 of structure tagname_2

var2.var1.member_2 - refers to the member_2 of structure tagname_2

Let's take an example:

```
struct student
{
struct person

{
char name[20];
int age;

char dob[10];

} p ;
int rollno;

float marks;

} stu;
```

Here we have defined structure person as a member of structure student. Here is how we can access the members of person structure.

stu.p.name - refers to the name of the person

stu.p.age - refers to the age of the person stu.p.dob -

refers to the date of birth of the person

It is important to note that structure person doesn't exist on its own. We can't declare structure variable of type struct person anywhere else in the program.

Instead of defining the structure inside another structure. We could have defined it outside and then declare it's variable inside the structure where we want to use it. For example:

```
struct person
{
char name[20];
int age;
char dob[10];
};
```

We can use this structure as a part of a bigger structure

```
struct student

{

struct person info;

 int rollno;

float marks;

}
```

Here the first member is of type struct person. If we use this method of creating nested structures then you must first define the structures before creating variables of its types. So, it's mandatory for you to first define person structure before using it's variable as a member of the structure student.

The advantage of using this method is that now we can declare a variable of type struct person in anywhere else in the program.

Initializing nested Structures

Nested structures can be initialized at the time of declaration. For example:

```
struct person
{
char name[20]; int age; char dob[10];
};
struct student
{
struct person info; int rollno; float
marks[10];
}
struct student student_1 = {{"Adam", 25,1990},101,90};
```

The following program demonstrates how we can use nested structures.

```
#include<stdio.h>
struct person
{
char name[20];
 int age;
char dob[10];
};

struct student
{
struct person info; int roll_no; float marks;
};

int main()
{
struct student s1;
printf("Details of student: \n\n");
printf("Enter name: "); scanf("%s", s1.info.name);
printf("Enter age: "); scanf("%d", &s1.info.age);
printf("Enter dob: "); scanf("%s", s1.info.dob);
printf("Enter roll no: "); scanf("%d", &s1.roll_no);
printf("Enter marks: "); scanf("%f", &s1.marks);
printf("\n*******************************\n\n");
printf("Name: %s\n", s1.info.name);
printf("Age: %d\n", s1.info.age);
printf("DOB: %s\n", s1.info.dob); printf("Roll no:%d\n", s1.roll_no); printf("Marks: %.2f\n",
s1.marks);
signal to operating system program ran fine return 0;
}
```

Need of array of structures:

Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type, therefore an array can be of structure type.

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.

Syntax

Struct struct-name
{
datatype var1;
datatype var2;

----
datatype varN
};
Struct struct-name obj[size]

Initializing Array of Structure:

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 AbdulKalam",350.00},

{"Complete C",1200,"HerbtSchildt",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,"AbdulKalam",350.00},

{"Complete C",1200,"HerbtSchildt",450.00}

};

}
```

C Program on book details using array of structures:

```c
#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---------BookDetails------------------- ");
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;
}
```

Some Observations and Important Points:

Tip #1 : All Structure Members need not be initialized

```c
#include<stdio.h>

struct Book

{

char bname[20];

int pages;

char author[20];

floatprice;

}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);

}
```

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];

floatprice;

}b1[3] ={

{},

{"Book2",500,"AAK",350.00},

{"Book3",120,"HST",450.00}

};

Output :

BookName   :

Book Pages : 0 Book

Author :

Book Price : 0.000000

Structures and functions

In C, structure can be passed to functions by two methods:

1.Passing by value (passing actual value as argument)

2.Passing by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

C program to create a structure student, containing name and roll and display the information.

```c
#include <stdio.h>
struct student
{
char name[50];
int roll;
};
void display(struct student stu);
int main()
{
struct student stud;
printf("Enter student's name: ");
scanf("%s", &stud.name);
printf("Enter roll number:");
scanf("%d",&stud.roll);
display(stud); // passing structure variable stud as argument return 0;
}
void display(struct student stu){
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);

}
```

Output :
Enter student's name: Raju
Enter roll number: 48
Name: Raju
Roll : 48

Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference. If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

C program to add two distances and display the result without the return statement.

```c
#include <stdio.h>
struct distance
{
int feet;
float inch;
};
void add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d", &dist1.feet);
printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d", &dist2.feet);
printf("Enter inch: ");
scanf("%f", &dist2.inch);
add(dist1, dist2, &dist3);
printf("\nSum of distances = %d\'-%.1f\"", dist3.feet, dist3.inch);
return 0;
}
void add(struct distance d1,struct distance d2, struct distance *d3)
{
d3-> feet = d1.feet + d2.feet;
d3-> inch = d1.inch + d2.inch;
if (d3->inch >= 12)
{
d3->inch -= 12;
++d3->feet;
}
}
```

Output:

First distance
Enter feet:12
Enter inch: 6.8
Second distance
Enter feet:5
Enter inch: 7.5
Sum of distances = 18'-2.3"

Structure and Pointer

Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name
{
member1;
member2;
.
.
.
};

int main()
{
struct name *ptr;
}
```

Here, the pointer variable of type **struct name**is created.

Accessing structure's member through pointer

A structure's member can be accesssed through pointer in two ways:

Referencing pointer to another address to access memory

Using dynamic memory allocation

1.**Referencing pointer to another address to access the memory**

Consider an example to access structure's member through pointer.

```
#include<stdio.h>

typedef struct person
{
int age;
float weight;
};

int main()
{
struct person *personPtr, person1;
personPtr=&person1;          // Referencing pointer to memory address ofperson1
printf("Enterinteger:");
scanf("%d",&(*personPtr).age);
```

```
printf("Enter number: ");
scanf("%f",&(*personPtr).weight);
printf("Displaying: ");
printf("%d%f",(*personPtr).age,(*personPtr).weight);
return 0;
}
```

In this example, the pointer variable of type struct personis referenced to the address of person1. Then, only the structure member through pointer can can accessed.

Using -> operator to access structure pointer member

Structure pointer member can also be accessed using -> operator.

(*personPtr).age is same as personPtr->age

(*personPtr).weight is same as personPtr->weight

2. Accessing structure member through pointer using dynamic memoryallocation

To access structure member using pointers, memory can be allocated dynamically using malloc()functiondefined under "stdlib.h" header file.

Syntax to use malloc()

ptr = (cast-type*) malloc(byte-size)

Example to use structure's member through pointer using malloc() function.

```
#include<stdio.h>
#include<stdlib.h>
struct person {
int age;
float weight;
char name[30];
};
int main()
{
struct person *ptr;
int i, num;
printf("Enter number of persons: ");
scanf("%d", &num);
ptr = (struct person*) malloc(num * sizeof(struct person));

// Above statement allocates the memory for n structures with pointer personPtr pointing to base
address */ for(i = 0; i < num; ++i)
```

```c
{
printf("Enter name, age and weight of the person respectively:\n"); scanf("%s%d%f", &(ptr+i)-
>name, &(ptr+i)->age, &(ptr+i)->weight);
 }
printf("Displaying Infromation:\n");
for(i = 0; i < num; ++i)
printf("%s\t%d\t%.2f\n", (ptr+i)->name, (ptr+i)->age, (ptr+i)->weight);
return 0;
}
```

Output

Enter number of persons: 2
Enter name, age and weight of the person respectively:
Adam

2
3.2

Enter name, age and weight of the person respectively: Eve6

2.3

Displaying Information:

Adam 2   3.20
Eve   6   2.30

In C, structure can be passed to functions by two methods:

- Passing by value (passing actual value asargument)

- Passing by reference (passing address of anargument)

Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

C program to create a structure student, containing name and roll and display the information.

```c
#include <stdio.h>

struct student
{
char name[50];
int roll;
};

void display(struct student stu);

// function prototype should be below to the structure declaration otherwise compiler shows error
int main()
{
struct student stud;
printf("Enterstudent's name: ");
scanf("%s",&stud.name);
printf("Enter rollnumber:");
scanf("%d",&stud.roll);
display(stud); // passing structure variable stud as argument return 0;
 }
void display(struct student stu){
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);
}
```

Output

Enter student's name: Kevin Amla
Enter roll number: 149

Output

Name: Kevin Amla
Roll: 149

Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

C program to add two distances (feet-inch system) and display the result without the return statement.

125

```c
#include <stdio.h>
struct distance
{
int feet;
float inch;
};

void add(struct distance d1,struct distance d2, struct distance *d3);
int main()
 {
struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d", &dist1.feet);
printf("Enter inch: ");
scanf("%f", &dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d", &dist2.feet);
printf("Enter inch: ");
scanf("%f", &dist2.inch);
add(dist1, dist2, &dist3);

//passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by
reference

printf("\nSum of distances = %d\'-%.1f\"", dist3.feet, dist3.inch);

return 0;
}
void add(struct distance d1,struct distance d2, struct distance *d3)
{

//Adding distances d1 and d2 and storing it in d3 d3-

>feet = d1.feet + d2.feet;

d3->inch = d1.inch + d2.inch;

if (d3->inch >= 12) { /* if inch is greater or equal to 12, converting it to feet. */ d3-
>inch -=12;
++d3->feet;
}
}
```

Output

First distance
Enter feet: 12
Enter inch: 6.8

Second distance Enter
feet: 5
Enter inch: 7.5

Sum of distances = 18'-2.3"

In this program, structure variables dist1 and dist2 are passed by value to the addfunction
(because value of dist1 and dist2 does not need to be displayed in main function).

But, dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument.

Due to this, the structure pointer variable d3 inside the add function points to the address ofdist3
from the calling main function. So, any change made to the d3 variable is seen in dist3 variable
in main function.

As a result, the correct sum is displayed in the output.

Self Referential Structures:

Self referential structures contain a pointer member that points to a structure of the same
structure type.

In other words, a self-referential C structure is the one which includes a pointer to an instance
of itself.

Syntax of Self-Referential Structure in C Programming

```
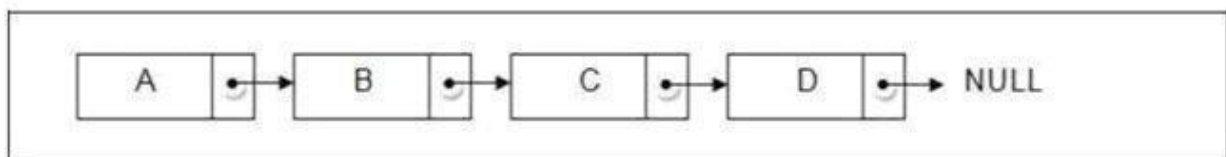struct demo
{
Data_type member1, member2;
struct demo *ptr1, *ptr2;
}
```

As you can see in the syntax, ptr1 and ptr2 are structure pointers that are pointing to the structure demo, so structure demo is a self referential structure. These types of data structures are helpful in implementing data structures like linked lists and trees.

It is an error to use a structure variable as a member of its own struct type structure or union type union, respectively.

Self Referential Structure Example

```
struct node
{
int data;
struct node *nextPtr;
}
```

nextPtr

- is a pointer member that points to a structure of the same type as the one beingdeclared.

- is referred to as a link. Links can tie one node to anothernode.

The concept of **linked lists**, **stacks**, **queues**, trees and many others works on the principle of self-referential structures.

One important point worth noting is that you cannot reference the typedef that you create within the structure itself in C programming.

An example of Self-Referential Structure in C

```
#include<stdio.h>
#include<stdlib.h>-
struct node //structure of the node in the list
{
int info;
struct node * link;
};
int main()
{
int choice;
typedef struct node NODE;
NODE *PTR, *START;
START = NULL;//Initialising START to NULL
while(1)
{
printf("\n1.Enter the new node at the start\n");
printf("2.Display the elements of the list\n");
```

```c
printf("3.Exit\n");
printf("Enter Choice\n");
scanf("%d",&choice);
switch(choice)
{
case 1:PTR = (NODE*)malloc(sizeof(NODE)); //Allocating Memory to new node
printf("Enter the number you want to enter at the start\n");
scanf("%d",&PTR->info);
if(START == NULL)
{
START = PTR;
PTR->link = NULL;
}
else
{
PTR->link = START;
START = PTR;
}
break;
case 2:PTR = START;
printf("The elements in the list are::\n");
while(PTR->link != NULL)
{
printf("%d\t",PTR->info);
PTR = PTR->link;
}
printf("%d",PTR->info);
break;
case 3:exit(1);
break;
default: printf("\nEnter Valid Choice");
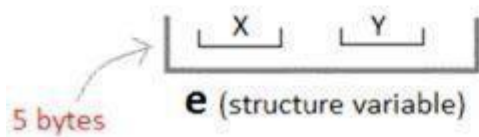}
}
return 0;
}
```

Unions in CLanguage

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest datamember.

Structure

```
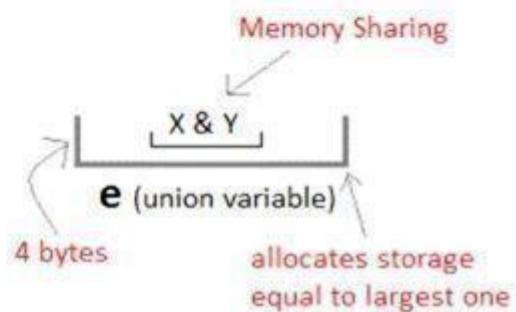struct Emp
{
 char X ;    // size 1 byte
 float Y ;   // size 4 byte
}e ;
```


5 bytes   e (structure variable)

Unions

```
union Emp
{
 char X ;
 float Y ;
}e ;
```

Memory Sharing

e (union variable)

4 bytes   allocates storage
           equal to largest one

```
#include<stdio.h>
struct student
{
char sname[20];
char fname[50];
int marks;
}s;

main()
{
printf("size of union=%d",sizeof(union student));
}
```

Output:size of union =50
Maximum size of the variable is the size of union so its size is 50.

This implies that although a union may contain many members of different types, it cannot handle all the members at the same time. A union is declared using the union keyword.

```
union item
{
int m; float
x; char c;
}It1;
```

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size. Thecompiler allocates the storage that is large enough to hold the largest variable type in theunion.

In the union declared above the member x requires 4 bytes which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

Accessing a Union Member

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
int a;
float b;
char c;
}t;

t.a;       //to access members of union t
t.b;
t.c;
```

Time for an Example

```
#include <stdio.h>

union item
{
int a;
float b;
char ch;
};


int main( )
{

union item it; it.a=12;

it.b = 20.2;
it.ch = 'z';

printf("%d\n", it.a);
```

```
printf("%f\n", it.b);
printf("%c\n", it.ch);

return 0;
}
```

output
-26426
20.1999

As you can see here, the values of a and b get corrupted and only variable c prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have thememory.

In the above example, value of the variable c was stored at last, hence the value of other variables is lost.

Bit fields

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure calle 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 situation. 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, above structure can be re-written as follows:

```
struct
{
unsigned int
widthValidated : 1;
unsigned int
heightValidated :
1;
} status
```

Now, the above structure will require 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 status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate 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 sim ple 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 m ain( )
{
printf( "Mem ory size occupied by status1 : %d\n", sizeof(status1));
printf( "Mem ory size occupied by status2 : %d\n", sizeof(status2));
return 0;
}
```

When the above code is compiled and executed, it produces the aboveresult:

Memory size occupied by status1 :8
Memory size occupied by status2 :4

Bit Field Declaration

The declaration of a bit-field has the form inside a structure:

```
struct
{
type [member_name] : width ;
};
```

Below the description of variable elements of a bit field:

Elements	Description
type	An integer type that determines how the bit-field's value is interpreted. The type may be int, signed int, unsigned int.
member_name	The name of thebit-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 only 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 C compiler that age variable is going to use only 3 bits to store the value, if you will 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 m ain( )
{
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 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.

Syntax:

typedef data_type new_name;

typedef: It is a keyword.
data_type: It is the name of any existing type or user defined type created using structure/union.
new_name: alias or new name you want to give to any existing type or user defined type.

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 −

```c
#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;
}
```

OUTPUT:
Book title : C Programming Book
author : Nuha Ali
Book subject : C Programming Tutorial Book
book_id : 6495407

Structures with typedef:

Consider the following student structure

struct student

{

    int mark [2];

    char name [10];

    float average;

}

Variable for the above structure can be declared in two ways.

1 st way:

struct student record; /* for normal variable */ struct

student*record;   /* for pointer variable*/

136

typedef struct student status;

When we use ―typedef‖ keyword before struct <tag_name> like above, after that we can simply use type definition ―status‖ in the C program to declare structure variable. Now, structure variable declaration will be, ―status record‖. This is equal to ―struct student record‖. Type definition for ―struct student‖ is status.
        status = ―struct student‖.

An Alternative Way for Structure Declaration Using Typedef in C:

```
typedef struct student
{
int mark [2];
char name [10];
float average;
} status;
```

To declare structure variable, we can use the belowstatements.

status record1; /* record 1 is structure variable */ status

record2;   /* record 2 is structure variable*/

// Structure using typedef:

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
int id;
char name[20];
float percentage;
} status;
int main()
{
status record;
record.id=1;
strcpy(record.name, "Raju");
record.percentage = 86.5;
printf(" Id is: %d \n", record.id);
printf(" Name is: %s \n", record.name);
printf(" Percentage is: %f \n",
record.percentage);
return 0;
}
```

OUTPUT:

Id is: 1

Name is: Raju

Percentage is: 86.500000

Another Example

```
#include <stdio.h>
#include <limits.h>
int main()
{
typedef long long int LLI;
printf("Storage size for long long
int data " \"type : %ld \n", sizeof(LLI));
return 0;
}
```

OUTPUT: Storage size for long long int data type : 8

Enumeration data type:

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword enum is used.

Syntax:

enum flag {const1, const2,……constN};

Here, name of the enumeration is flag. Constants likeconst1,const2,................, constN arevalues of type flag.

By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).

// Changing the default value of enum elements enum

suit{

```
club=0;
diamonds=10;
hearts=20;
spades=3;
```

};

138

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

Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Like any variable or constant, you must declare a pointer before using it to store any variable address.

The general form of a pointer variable declaration is

Datatype *variable_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 follow the rules of identifier. Examples for different pointer declarations:

int *ip; /* pointer to an integer */

double *dp; /* pointer to a double */

float *fp; /* pointer to a float */

char *ch; /* pointer to a character*/

Diff Between pointer and normalvariable:

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

Overview: int i; int *j; j=&i;

Variable Name →	I	j
Value of Variable →	3	65524
Address of Location →	65524	65522

Explanation:

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

Theaddressofthevariable„**i**"isstoredinanotherintegervariablewhosenameis„**j**"andwhichis having corresponding address65522

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

Pointer Basic Example:

```
#include
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;
}
```

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

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.

Pointer information:

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.

Simple Pointer Example #1:

```
#include<stdio.h>intmain()
{
int a = 3; int *ptr; ptr = &a;
printf(―the value of is %d‖, *ptr);
printf(―the value of a is %d‖,a);
return(0);
}
```

Explanation of Example:

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

Reference operator (&) and Dereference operator (*)

The operator, & is called reference operator. It gives you the address of a variable.

One more complement operator is dereference operator (*), that gets the value from the address
Program on Reference and Dereference operator

```
#include <stdio.h>
int main()
{
int* pc; int c; c=22;
printf("Address of c:%u\n",&c);
printf("Value of c:%d\n\n",c); pc=&c;
printf("Address of pointer pc:%u\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
c=11;
```

142

```
printf("Address of pointer pc:%u\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
printf("Address of c:%u\n",&c);
printf("Value of c:%d\n\n",c);
return0;
}
```

OUTPUT:

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2D

Pointer Arithmetic

Pointer is a variable that points to a memory location. Memory addresses are numeric value that ranges from zero to maximum memory size in bytes. These addresses can be manipulated like simple variables. You can increment, decrement, calculate or compare these addresses manually.

C language provides a set of operators to perform arithmetic and comparison of memory addresses. Pointer arithmetic and comparison in C is supported by following operators -

- Increment and decrement ++ and--
- Addition and Subtraction + and-
- Comparison <, >, <=, >=, ==,!=

Pointer increment and decrement

Increment operator when used with a pointer variable returns next address pointed by the pointer. The next address returned is the sum of current pointed address and size of pointer data type.

Or in simple terms, incrementing a pointer will cause the pointer to point to a memory location skipping Nbytes from current pointed memory location. Where N is size of pointer data type.

Similarly, decrement operator returns the previous address pointed by the pointer. The returned address is the difference of current pointed address and size of pointer data type.

For example, consider the below statements.

143

```
int num = 5; // Suppose address of num = 0x1230
int*ptr;       // Pointervariable

ptr=&num;      // ptr points to 0x1230 or ptr points tonum
ptr++;         // ptr now points to 0x1234, since integer size is 4 bytes
ptr--;         // ptr now points to0x1230
```

Example program to perform pointer increment and decrement

Array in memory are stored sequentially, hence is the best example to demonstrate pointer increment, decrement operations.

```c
#include <stdio.h>
#define SIZE 5

int main()
{
    int arr[SIZE] = {10, 20, 30, 40, 50};
    int *ptr;
    int count;

    ptr = &arr[0]; // ptr points to arr[0]

    count = 0;

    printf("Accessing array elements using pointer \n");
    while(count < SIZE)
    {
        printf("arr[%d] = %d \n", count, *ptr);

        // Move pointer to next array element
        ptr++;

        count++;
    }

    return0;

}
```

Output-

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

144

Pointer addition and subtraction

Pointer increment operation increments pointer by one. Causing it to point to a memory location skipping N bytes (where N is size of pointer data type).

We know that increment operation is equivalent to addition by one. Suppose an integer pointer int * ptr. Now, ptr++ is equivalent to ptr = ptr + 1. Similarly, you can add or subtract any integer value to apointer.

Adding K to a pointer causes it to point to a memory location skipping K * N bytes. Where K is a constant integer and N is size of pointer data type.

Let us revise the above program to print array using pointer.

```c
#include <stdio.h>
#define SIZE 5

int main()
{
    int arr[SIZE] = {10, 20, 30, 40, 50};
    int *ptr;
    int count;

    ptr = &arr[0]; // ptr points to arr[0]

    count = 0;

    printf("Accessing array elements using pointer \n");
    while(count < SIZE)
    {
        printf("arr[%d] = %d \n", count, *(ptr + count));

        count++;
    }

    return 0;

}
```

- When count = 0, (ptr + count) is equivalent to (ptr + 0) which points to arr[0] and hence prints 10.
- When count = 1, (ptr + count) is equivalent to (ptr + 1) which points to arr[1] and hence prints 20.
- Similarly when count = 4, (ptr + count) is equivalent to (ptr + 4) which points to arr[4]and hence prints50.

Output of above program is same as first program.

Pointer comparison

In C, you can compare two pointers using <u>relational operator</u>. You can perform six different type of pointer comparison <, >, <=, >=, == and !=.

Note: Pointer comparison compares two pointer addresses to which they point to, instead of comparing their values.

Pointer comparisons are less used when compared to pointer arithmetic. However, I frequently use pointer comparison when dealing with arrays.

Pointer comparisons are useful,

- If you want to check if two pointer points to same location. Forexample,

```c
int main()
{
   int num = 10;
   int *ptr1=&num;     // ptr1 points tonum
   int *ptr2=&num;     // ptr2 also points to nm

   if(ptr1 == ptr2)
   {
     // Both pointers points to same memory location
     // Do some task
   }

   return 0;
}
```

- If you want to check if a pointer points within an array range. Forexample,

```c
intmain()
{
   int arr[5] = {10, 20, 30, 40, 50};
   int *ptr = &arr[0]; // ptr points to arr[0]

   while(ptr <= &arr[4])
   {
     // ptr will always point within the array
     // Do some task
```

146

```
    // Move ptr to next array element
    ptr++;
}

return0;

}
```

Pointer to pointer

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **DoublePointer**.
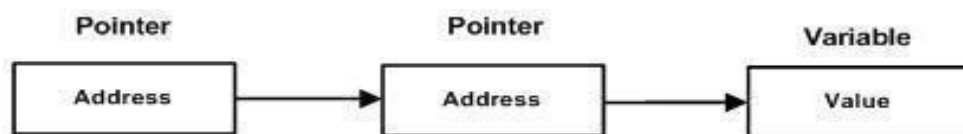
Syntax:
```
int **p1;
```

Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:
```
int ***p2;
```

Simple program to represent Pointer to a Pointer

```
#include
<stdio.h>int main()
{
int a =10;
   int *p1;       //this can store the address of variablea
   int**p2;
```

147

```
    /*
      this can store the address of pointer variable p1 only.
      It cannot store the address of variable 'a'
    */
  p1 = &a;
  p2 = &p1;
   printf("Address of a = %u\n", &a);
  printf("Address of p1 = %u\n", &p1);
  printf("Address of p2 = %u\n\n", &p2);


  // below print statement will give the address of 'a'
  printf("Value at the address stored by p2 = %u\n", *p2);
  printf("Value at the address stored by p1 = %d\n\n", *p1);
  printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
   /*
      This is not allowed, it will give a compile time error-
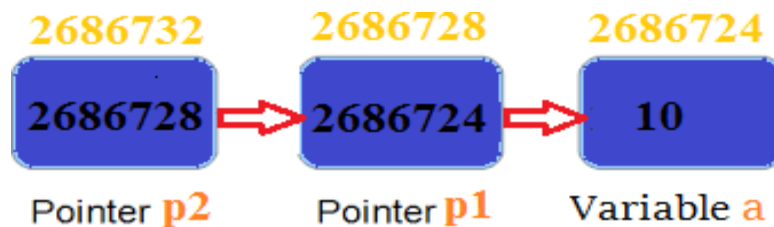      p2 = &a;
      printf("%u", p2);
   */
   return0;
}
```

Output

Address of a =
2686724Address of p1
=2686728
Address of p2 =2686732
Value at the address stored by p2 =
2686724Value at the address stored by p1 =
10Value of **p2 =10

Explanation of the above program

- p1pointer variable can only hold the address of the variable a(i.e Number of indirection operator(*)-1 variable). Similarly, p2variable can only hold the address of variable p1. It cannot hold the address of variable a.

- *p2gives us the value at an address stored by the p2pointer. p2stores the address of p1pointer and value at the address of p1is the address of variable a. Thus, *p2prints address ofa.

- **p2can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement,youknow*p2meanstheaddressofvariablea.Hence,thevalueattheaddress*p2is 10. Thus, **p2prints 10.

Generic pointers

A Generic pointer is a special pointer that can point to object of any type. A Generic pointer is typeless pointer also known as **void pointer**. void pointer is an approach towards generic functions and generic programming in C.

Note: Writing programs without being constrained by data type is known as generic programming.A generic function is a special function that focuses on logic without confining to data type. For example, logic to insert values in array is common for all types and hence can be transformed t generic function.

Syntax to declare void or Generic pointer

  void * pointer-name;

Example to declare void or Generic pointer

void * vPtr;

How to dereference a void or Generic pointer

Dereferencing is the process of retrieving data from memory location pointed by a pointer. It converts block of raw memory bytes to a meaningful data (data is meaningful if **type** is associated).

While dereferencing a void or Generic pointer, the C compiler does not have any clue about type of value pointed by the void pointer. Hence, dereferencing a void pointer is illegal in C. But, a pointer will become useless if you cannot dereference it back.

To dereference a void pointer you must <u>typecast</u>it to a valid pointer type.

Example to dereference a void or Generic pointer

```
int num = 10;
void * vPtr = &num; // void pointer pointing at num int
value = *((int *) vPtr); // Dereferencing void pointer
```

void or Generic pointer arithmetic

 void or **Generic** pointer arithmetic is illegal in C programming, due to the absence

of type. However, some compiler supports void pointer arithmetic by assuming it as

a char pointer. To perform pointer arithmetic on void pointer you must first typecast

to other type.

Example of void or Generic pointer arithmetic

```
int arr[] = {10, 20, 30, 40, 50};
void * vPtr = &arr; // void pointer pointing at arr
vPtr = ((int *) vPtr + 1); // add 1 to void pointer
```

Example program to use void pointer

Write a C function to accept an array and print its elements. The function must accept array of different types

```
/**C program to demonstrate void pointer */
#include <stdio.h>
#define SIZE 10
/* Function declaration */
```

```c
void printArray(void * vPtr, int size, int type);
int main()
{
int num[SIZE] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
float fractional[SIZE] = {1.1f, 1.2f, 1.3f, 1.4f, 1.5f, 1.6f, 1.7f, 1.8f, 1.9f, 2.0f}; char
characters[SIZE] = {'C', 'o', 'd', 'e', 'f', 'o', 'r', 'w', 'i', 'n'}; printf("\nElements of integer array: ");
printArray(&num, SIZE, 1);
printf("\nElements of float array: ");
printArray(&fractional, SIZE, 2);
printf("\nElements of character array: ");
printArray(&characters, SIZE, 3);
return 0;
}
/**
*Function to print array of different types.
*@vPtr Pointer to an array
*@size Size of the array
*@type Integer value specifying type of array. 1 - Integer,
*2      - Float,
*3      - Character
*/
void printArray(void * vPtr, int size, int type)
{
int i;
for(i=0; i<size; i++)
{
//Print array elements based on their type switch(type)
{
case 1:
/* Typecast void pointer to integer then print
*/ printf("%d, ", *((int *)vPtr + i)); break;
case 2:
/* Typecast void pointer to float then print */ printf("%f, ", *((float *)vPtr + i));
break; case 3:
/* Typecast void pointer to char then print */ printf("%c, ", *((char *)vPtr + i));
break;
}
}
}

Output:
Elements of integer array: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
Elements of float array: 1.100000, 1.200000, 1.300000, 1.400000, 1.500000, 1.600000,1.700000,
1.800000, 1.900000, 2.000000,
Elements of character array: C, o, d, e, f, o, r, w, i, n
```

151

while not end of file and array not exhausted, read a string store it in an array of strings and assign the string to an element of a pointer array access the array of strings and print them out access the array of pointers .

Arrays and Pointers

Arrays and pointers are closely related in C. In fact an array declared as int A[10];

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a const int*. Since A is a constant pointer, A = NULL would be an illegal statement. Arrays and pointers are synonymous in terms of how they use to access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

Consider the following array:

int age[5];
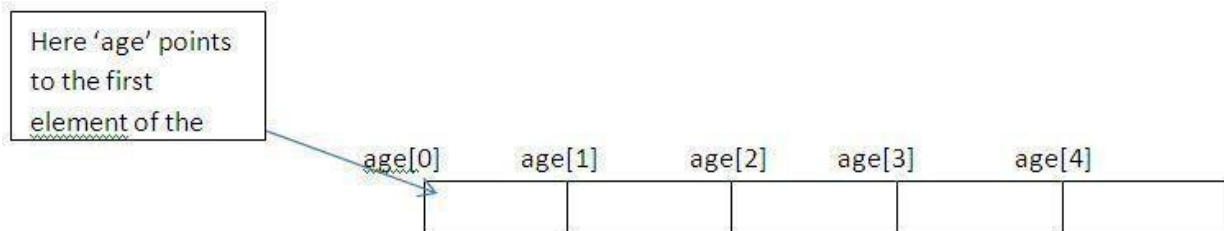
In C , name of the array always points to the first element of an array. Here, address of first element of an array is &age[0]. Also, age represents the address of the pointer where it is pointing. Hence, &age[0] is equivalent to age. Note, value inside the address &age[0] and address age are equal. Value in address &age[0] is age[0] and value in address age is *age. Hence, age[0] is equivalent to*age.

C arrays can be of any type. We define array of ints, chars, doubles etc. We can also define an array of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

char* A[n];

each cell in the array A[i] is a char* and so it can point to a character. Now if you would like to assign a string to each A[i] you can do something like this.

A[i] = malloc(length_of_string + 1);

Again this only allocates memory for a string and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words) you need to allocate memory for n char*'s and then allocate just the right amount of memory for each string.

In C, you can declare an array and can use pointer to alter the data of an array. This program declares the array of six element and the elements of that array are accessed using pointer, and returns the sum.

Program to find the sum of six numbers with arrays and pointers.

```
#include <stdio.h>
int main()
{
int i,class[6],sum=0;
printf("Enter 2 numbers:\n");
for(i=0;i<6;++i)
{

scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]
sum+= *(class+i); // *(class+i) is equivalent to class[i]
}

printf("Sum=%d",sum);
return 0;
}
Output
Enter 2 numbers:
2
3
```

Pointer	Array
A pointer is a place in memory that keeps address of another place inside	An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
Allows us to indirectly access variables. In other words, we can talk about its address rather than its value	Expression a[4] refers to the 5th element of the array a.
Pointer can't be initialized at definition	Array can be initialized at definition. Example int num[] = { 2, 4, 5}

Pointer is dynamic in nature. The memory allocation can be resized or freed later.	They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically

Pointer Definition:

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Function basics:

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

Functions Parameter passing methods:

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Syntax: datatype function_name(datatype variable_name);

Call by reference:

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect theargument.

Syntax: datatype function_name(datatype *variable_name);

Function example: Call by value

```c
#include <stdio.h>
void swap(int i, int j)
{
int t;
t=i;
i=j;
j=t;
}
void main()
{
int a,b;
a=5;
b=10;
printf("%d %d\n", a, b);
swap(a,b);
printf("%d %d\n", a, b);
}
```

Analysis: When you execute this program, you will find that no swapping takes place. The values of a and b are passed to swap, and the swap function does swap them, but when the function returns nothing happens.

Function example: Call by reference:

```c
#include <stdio.h>
void swap(int *i, int*j)
{
int t;
t =*i;
*i = *j;
*j = t;
}

void main()
{
int a,b;
a=5;
b=10;
printf("%d %d\n",a,b);
swap(&a,&b);
printf("%d %d\n",a,b);
}
```

Analysis: The above code uses ***i** and ***j**, it means **a** and **b**. When the function completes, a andb have beenswapped.

Passing Pointer to a Function:

When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passedvariable.

Example:1

```c
#include <stdio.h>
void salaryhike(int *var, int b)
{
*var = *var+b;
}
int main()
{
int salary=0,bonus=0;
printf("Enter the employee currentsalary:");
scanf("%d",&salary);
printf("Enter bonus:");
scanf("%d", &bonus);
salaryhike(&salary, bonus);
printf("Final salary: %d", salary);
return 0;
}
```
OUTPUT:
```
 Enter theemployee
 current salary:10000
 Enter bonus:2000
 Final salary:12000
```

Example: 2

```c
 #include <stdio.h>
void swapnum(int *num1, int *num2)
{
int tempnum;

tempnum = *num1;
*num1 = *num2;
*num2 = tempnum;
}
int main( )
{
int v1 = 11, v2 = 77 ;
printf("Before swapping:");
printf("\nValue of v1 is: %d", v1);
```

156

```c
printf("\nValue of v2 is: %d", v2);

 /*calling swap function*/
swapnum( &v1, &v2 );

printf("\nAfter swapping:");
printf("\nValue of v1 is: %d", v1);
printf("\nValue of v2 is: %d", v2);
}
```
Output:
Before swapping:
Value of v1 is: 11
Value of v2 is: 77
After swapping:
Value of v1 is:77
Value of v2 is: 11

Functions of returning pointers

It is also possible for functions to return a function pointer as a value. This ability increases the flexibility of programs. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

Example of function pointers as returned values

```c
#include <stdio.h>
int* larger(int*, int*);
void main()
{
int a = 15;
int b = 92;
int *p;
p = larger(&a, &b);
printf("%d islarger",*p);
}

int* larger(int *x, int*y)
{
if(*x > *y)
return x;
else
return y;
}
```

Note:
1. Either use argument with functions. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function aswell.
2. Or, use static local variables inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available throughouttheprogram.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function.
A pointer to a function is declared as follows,

        type (*pointer-name)(parameter);

A function pointer can point to a specific function when it is assigned the name of that function.

```
 int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>
int sum(int x, int
y)
{
return x+y;
}
int main( )
{
int (*fp)(int, int);
fp = sum;
int s = fp(10, 15);
printf("Sum is %d", s);
return 0;
}
```

Out put: 25

Example function returning pointer

```
#include <stdio.h>
int* findLarger(int*, int*);
```

```c
void main()
{
int numa=0;
int numb=0;
int *result;
printf("\n\n Pointer : Show a function returning pointer:\n");
printf("      \n");
printf(" Input the first number : ");
scanf("%d", &numa);
printf(" Input the second number : ");
scanf("%d", &numb);
result=findLarger(&numa, &numb);
printf(" The number %d is larger. \n\n",*result);
}
int* findLarger(int *n1, int *n2)
{
if(*n1 > *n2) return n1;
else
return n2;
}
```

Out put:
Pointer : Show a function returning pointer :

Input the first number : 5
Input the second number : 6
The number 6 is larger.

Dynamic memory allocation

The exact size of array is unknown until the compile time, i.e., time when a compiler 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.

Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

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

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.

```c
#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;
}
```

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can"t be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

Difference Between malloc() and calloc() Functions :

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn"t initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

161

FILE HANDLING AND BASICALGORITHMS

File Operations

In C, you can perform four major operations on the file, either text or binary:
1. Creating a newfile
2. Opening an existingfile
3. Closing afile
4. Reading from and writing information to afile

Working With Files
When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fp;

Opening a file - for creation and edit

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

fp = fopen("fileopen","mode")

For Example:
fopen("E:\\cprogram\\newprogram.txt","w");
fopen("E:\\cprogram\\oldprogram.bin","rb");

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'. The writing mode allows you to create and edit (overwrite) the contents of thefile.
- Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode'rb'. The reading mode only allows you to read the file, you cannot write into thefile.

Sr.No.	Mode & Description
1	**"r"** Opens a file for reading. The file must exist.
2	**"w"**

		Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.
3	**"a"**	Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.
4	**"r+"**	Opens a file to update both reading and writing. The filemust exist.
5	**"w+"**	Creates an empty file for both reading and writing.
6	**"a+"**	Opens a file for reading and appending.

Closing a file:

The file (both text and binary) should be closed after reading/writing. Closing a file is performed using library function fclose().

fclose(fp); //fp is the file pointer associated with file to be closed

Reading And Writing A Text File

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

Writing to a text file

Example 1: Write to a text file using fprintf()

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

```c
int main()
{
int num;
FILE *fp;
fptr = fopen("C:\\program.txt","w");
if(fp == NULL)
{
printf("Error!");
exit(1);
}
printf("Enter num: ");
scanf("%d",&num);

fprintf(fp,"%d",num);
fclose(fp);

return 0;
}
```

This program takes a number from user and stores in the file program.txt.

File Types

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

In C language, we use a structure **pointer of file type** to declare a file.

FILE *fp;

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	description
fopen()	create a new file or open a existing

fclose()	file closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from afile
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

Opening a File or Creating a File

The fopen()function is used to create a new file or to open an existing file.

General Syntax :

*fp = FILE *fopen(const char *filename, const char *mode);

Here **filename** is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

*fp is the FILE pointer (FILE *fp), which will hold the reference to the opened (or created) file.

Closing a File

The fclose()function is used to close an already opened file.

General Syntax :

int fclose(FILE *fp);

Here fclose() function closes the file and returns **zero** on success, or **EOF** if there is an error
in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and
writing on file. getc()and putc()are simplest functions used to read and write individual
characters to afile.

```
#include<stdio.h>
#include<conio.h>
main()
{
FILE *fp;
char ch;
fp = fopen("one.txt", "w");
printf("Enter data");
while( (ch = getchar()) != EOF)
{
putc(ch,fp);
}
fclose(fp);
fp = fopen("one.txt", "r");
while( (ch = getc()) != EOF)
printf("%c",ch); fclose(fp);
}
```

Reading and Writing from File using fprintf() and fscanf()

```
#include<stdio.h>
#include<conio.h>
struct emp
{
char name[10];
int age;
};

void main()
{
struct emp e;
FILE *p,*q;
```

166

```
p = fopen("one.txt", "a");
q = fopen("one.txt", "r");
printf("Enter Name and Age");
scanf("%s%d", e.name, &e.age);
fprintf(p,"%s %d", e.name, e.age);
fclose(p);
do
{
fscanf(q,"%s %d", e.name, e.age);
printf("%s %d", e.name, e.age);
}
while( !feof(q) );
getch();
}
```

In this program, we have create two FILE pointers and both are refering to the same file but in different modes. **fprintf()** function directly writes into the file, while **fscanf()** reads from the file, which can then be printed on console usinf standard **printf()** function.

Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in thefile.

Reading and Writing in a Binary File

A Binary file is similar to the text file, but it contains only large numerical data. The Opening modes are mentioned in the table for opening modes above.

fread() and **fwrite()** functions are used to read and write is a binary file.

fwrite(data-element-to-be-written, size_of_elements,number_of_elements, pointer-to-file);

fread() is also used in the same way, with the same arguments like fwrite() function.

Below mentioned is a simple example of writing into a binary file

```
const char *mytext = "The quick brown fox jumps over the lazy dog";
FILE *bfp= fopen("test.txt", "wb");
if (bfp)
{
fwrite(mytext, sizeof(char), strlen(mytext), bfp) ;
fclose(bfp) ;
}
```

fseek(), ftell() and rewind() functions

    fseek() - It is used to move the reading control to different positions using fseek function.
    ftell() - It tells the byte location of current position of cursor in file pointer.
    rewind() - It moves the control to beginning of the file.

File opening modes

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 ofthe 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

File opening modes

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 :

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 for declaring thefile pointer.

FILE *fp;

Step 2 : Opening file hello.txt

fp = fopen ("filename","mode");

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 Modefp = fopen("hello.txt","r");
Writing Mode fp = fopen("hello.txt","w");
Append Mode fp = fopen("hello.txt","a");
```

Opening the File: Yet Another Example

```
#include<stdio.h>
void main()
{
FILE *fp;
char ch;
```

169

```
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 + Appending	New data is appended at the end of file	Create New 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 indicator until 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);

Way of Writing feof Function :

with if statement :

if(feof(fptr) == 1) // as if(1) is TRUE

printf("End of File");

Way 2 : In

While Loop

while(!feof(fptr)

)

{

--- ---

--- ---

}

Input And Output Operations With Files

File i/o functions

When working with files, we need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fptr;

C provides a number of functions that helps to perform basic file operations. Following are the functions

fopen():

create a new file or open a existing file

*fp = FILE *fopen(const char *filename, const char *mode);

fclose() :
function is used to close an already opened file.

    int fclose(FILE *fp);

getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

Example of Reading and Writing to File using fprintf() and fscanf() functions

```c
#include<stdio.h>
struct emp
{
char name[10];
int age;
};
void main()
{
struct emp e; FILE *p,*q;
p = fopen("one.txt", "a");
q = fopen("one.txt", "r");
printf("Enter Name and Age:");
scanf("%s %d", e.name, &e.age);
fprintf(p,"%s %d", e.name, e.age);
fclose(p);
do
{
fscanf(q,"%s %d", e.name, e.age);
printf("%s %d", e.name, e.age);
} while(!feof(q));
}
```

File Status Functions:

- fseek()
- ftell()
- rewind()
- fseek()

It is used to set file pointer to any position. Prototype is:

int fseek (FILE * stream, long int offset, int origin);

173

Parameters:

Stream: pointer to a file.
Offset: Number of bytes or characters from the origin.
Origin: The original position is set here. From this position, using the offset, the file pointer is set to a new position. Generally, three positions are used as origin:

SEEK_SET - Beginning of file
SEEK_CUR - Current position of the file pointer
SEEK_END - End of file

Return

Type: Integer
Value: On success Zero (0) On failure Non-Zero

Example
```
int main()
{
FILE *fp;
fp = fopen ( "file.txt" , "w" );
if (fp==NULL)
printf ("Error in opening file");
 else
{
fputs("I am supporter of France.",fp );
fseek (fp , 18 , SEEK_SET );
fputs ("Brazil" , fp );
fseek(fp , 0 , SEEK_CUR );
fputs(" and Portugal" , fp );
fclose ( fp );
}
return 0;
}
```

Then using SEEK_SET and offset, the word France is replaced by Brazil. Then by using SEEK_CUR, and position is appended with the string.

ftell()

It is used to get current position of the file pointer. The function

prototype is: long int ftell (FILE * stream);

Parameters

stream: Pointer to a file.

174

Return

Type: long integer.

Value: On success value of current position or
offset bytes On failure -1. System specific
error no is set

Example

Code:

```
int main ()
{
FILE * fp;
ong int len;
fp = fopen ("file.txt","r");
if (fp==NULL)
printf ("Error in opening file");
else
{
fseek (fp, 0, SEEK_END);
len=ftell (fp);
fclose (fp);
printf ("The file contains %ld characters.\n",len);
}
return 0;
}
```

In this example, file.txt is opened and using fseek(), file pointer is set to the end of the file. Then,
ftell() is used to get the current position, i.e. offset from the beginning of the file.

rewind()

It is used to set the file pointer at the beginning of the file.

Function prototype: void rewind (FILE * stream);

Parameters
stream: Pointer to a file.

In any stage of the program, if we need to go back to the starting point of the file, we can use

rewind() and send the file pointer as the parameter.

Example
Code:

```
int main ()
{
int n; FILE * fp;
fp = fopen ("file.txt","w+");
if(fp==NULL)
printf ("Error in opening file");
else { fputs ("France is my favorite team",fp);
rewind (fp);
fputs("Brazil",fp);
fclose (fp);
}
return 0;
}
```

In the above example, first, some string is written in the file. Then rewind() is used to set the file pointer at the beginning of the file. Then overwrite a word.

File Position functions :

The C library function **int fseek(FILE *stream, long int offset, int whence)** sets the file position of the **stream** to the given **offset**.

Following is the declaration for fseek() function.

int fseek(FILE *stream, long int offset, int whence)

Parameters

- ❖ **stream** − This is the pointer to a FILE object that identifies thestream.

- ❖ **offset** − This is the number of bytes to offset fromwhence.

- ❖ **whence** − This is the position from where offset is added. It is specified by one ofthe following constants−

Sr.No.	Constant & Description
1	**SEEK_SET** Beginning of file
2	**SEEK_CUR** Current position of the file pointer
3	**SEEK_END** End of file

Return Value:

This function returns zero if successful, or else it returns a non-zero value.

Example:

The following example shows the usage of fseek() function.

```c
#include <stdio.h>
int main ()
{
FILE *fp;
fp = fopen("file.txt","w+");
fputs("This is tutorialspoint.com", fp);
fseek( fp, 7, SEEK_SET );
fputs(" C Programming Language", fp);
fclose(fp);
return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content. Initially program creates the file and writes *This is tutorialspoint.com* but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content –

This is C Programming Language
Now let's see the content of the above file using the following program −

```c
#include <stdio.h>
int main ()
{
FILE *fp; int c;
fp = fopen("file.txt","r");
while(1)
{
c = fgetc(fp);
if( feof(fp) )
{
break;
}
printf("%c", c);
}
fclose(fp);
return(0);
}
```

Let us compile and run the above program to produce the following result −

This is C Programming Language

Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line argumentsand many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside thecode.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```c
#include <stdio.h>

int main( int argc, char *argv[] )
{
if( argc == 2 )
{
printf("The argument supplied is %s\n", argv[1]);
}
else if( argc > 2 )
{
printf("Too many arguments supplied.\n");
}
else
{
```

182

```
printf("One argument expected.\n");
}

}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
printf("Program name %s\n", argv[0]);
if( argc == 2 )
{
printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 )
{
```

```
printf("Too many arguments supplied.\n");
}
else
{
printf("One argument expected.\n");
}
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

$./a.out "testing1 testing2"

Progranm name ./a.out
The argument supplied is testing1 testing2

Searching

Searching is one of the most common problems that arise in computing. Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found. Search operations are usually carried out on a keyfield.

Well, to search an element in a given array, there are two popular algorithms available:

1. LinearSearch
2. BinarySearch

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list ofelements.

2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that goodtoo.

It has a very simple implementation.

Linear search C program

```c
#include <stdio.h>
 int main()
{
  int array[100], search, c, n;
  printf("Enter the number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
 for (c = 0; c < n; c++)
 {
 scanf("%d", &array[c]);
 printf("Enter a number to search\n"); scanf("%d", &search);
   printf("%d isn't present in the array.\n", search);

  return 0;
}
if (array[c] == search)
  If required element is found */
 printf("%d is present at location %d.\n", search, c+1);
break;
  }
```

Binary Search

1. BinarySearchis used withsorted arrayorlist. In binarysearch, wefollowthefollowingsteps:

2. We start by comparing the element to be searched with the element in the middle of thelist/array.

3. If we get a match, we return the index of the middleelement.

4. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middleelement.

5. Iftheelement/numbertobesearchedisgreaterin valuethanthemiddlenumber,thenwepickthe elements on the right side of the middle element(as the list/array is sorted, hence on the right, we willhave all the numbersgreaterthan themiddle number),andstartagainfromthestep1.

6. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sortedarrays.
   - It has a time complexity of **O(log n)** which is a very good timecomplexity

2. It has a simpleimplementation.

Binary search C program

```c
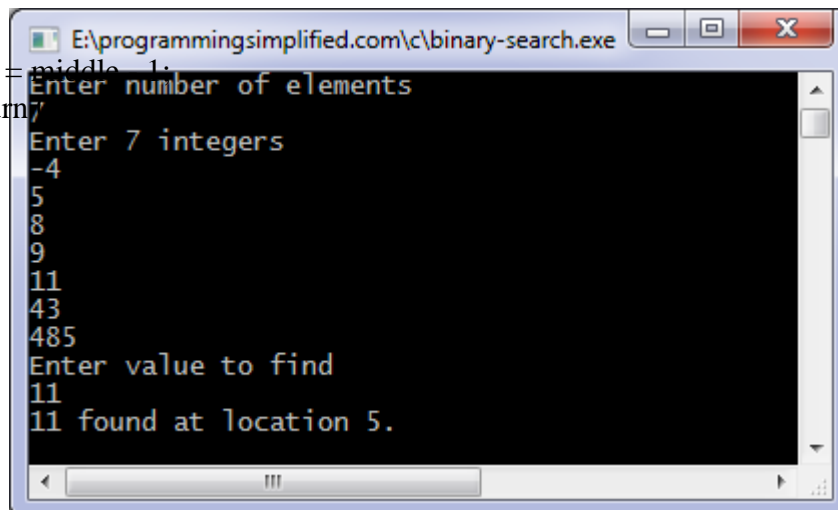#include <stdio.h>
 int main()
{
  int c, first, last, middle, n, search, array[100];
  printf("Enter number of elements\n");
  scanf("%d",&n);
  printf("Enter %d integers\n", n);
 for (c = 0; c < n; c++)
    scanf("%d",&array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
   first = 0;
  last = n - 1;
  middle =(first+last)/2;
   while (first <= last){
    if (array[middle] <search)
      first = middle +1;
    else if (array[middle] == search)
{
    printf("%d found at location %d.\n", search, middle+1);
      break;
  middle = (first + last)/2;
    }
     if (first > last) printf("Not   found!
  %d isn't present in the list.\n", search);
    }
    else
    last = middle - 1;
    return;
    }
```



```
E:\programmingsimplified.com\c\binary-search.exe
Enter number of elements
Enter 7 integers
-4
5
8
9
11
43
485
Enter value to find
11
11 found at location 5.
```

Sorting

Sorting is the basic operation in computer science. Sorting is the process of arranging data in some given sequence or order (in increasing or decreasing order).

For example you have an array which contain 10 elements as follow;
10, 3 ,6 12, 4, 17, 5, 9

After shorting value must be;
3, 4, 5, 6, 9, 10, 12, 17

Above value sort by apply any sorting technique. C language have following technique to sort values;

- BubbleSort

- SelectionSort

- InsertionSort

Bubble Sort in C

Bubble sort is a simple sorting algorithm in which each element is compared with adjacent element and swapped if their position is incorrect. It is named as bubble sort because same as like bubbles the lighter elements come up and heavier elements settledown.

Both worst case and average case complexity is O

(n^2). **Example Program for Bubble Sort**

```c
#include<stdio.h>

int main()
{
    int a[50],n,i,j,temp;
    printf("Enter the size of array: ");
    scanf("%d",&n);
    printf("Enter the array elements: ");

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

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

183

```c
            a[j]=a[j+1];
            a[j+1]=temp;
        }

    printf("\nArray after sorting: ");
    for(i=0;i<n;++i)
        printf("%d ",a[i]);

    return 0;
}
```
Output
Enter the size of array: 4
Enter the array elements: 3 7 9 2
Array after sorting: 2 3 7 9

Selection Sort in C

One of the simplest techniques is a selection sort. As the name suggests, selection sort is the selection of an element and keeping it in sorted order. In selection sort, the strategy is to find the smallest number in the array and exchange it with the value in first position of array. Now, find the second smallest element in the remainder of array and exchange it with a value in the second position, carry on till you have reached the end of array. Now all the elements have been sorted in ascending order.ep by StepProcess

The selection sort algorithm is performed using following steps...

- ➢ **Step 1:** Select the first element of the list (i.e., Element at first position in thelist).

- ➢ **Step 2:** Compare the selected element with all other elements in thelist.

- ➢ **Step 3:** For every comparision, if any element is smaller than selected element(for

    Ascending order), then these two areswapped.

- ➢ **Step 4:** Repeat the same procedure with next position in the list till the entire list issorted.

Example Program for Selection Sort

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

void main(){

    int size,i,j,temp,list[100];
    clrscr();

    printf("Enter the size of the List: ");
```

184

```c
   scanf("%d",&size);

   printf("Enter %d integer values: ",size);
   for(i=0; i<size; i++)
      scanf("%d",&list[i]);

   //Selection sort logic

   for(i=0; i<size; i++){
      for(j=i+1; j<size; j++){
          if(list[i] >list[j])
             {
             temp=list[i];
             list[i]=list[j];
             list[j]=temp;
          }
      }
   }

   printf("List after sorting is: ");
   for(i=0; i<size; i++)
      printf(" %d",list[i]);

   getch();
}
```

Insertion Sort in C

The insertion sort inserts each element in proper place. The strategy behind the insertion sort is similar to the process of sorting a pack of cards. You can take a card, move it to its location in sequence and move the remaining cards left or right as needed.

In insertion sort, we assume that first element A[0] in pass 1 is already sorted. In pass 2 the next second element A[1] is compared with the first one and inserted into its proper place either before or after the first element. In pass 3 the third element A[2] is inserted into its proper place and so on.

Example Program for Insertion Sort

```c
#include<stdio.h>
 int main()
{
   int i,j,n,temp,a[30];
   printf("Enter the number of elements:");
   scanf("%d",&n);
   printf("\nEnter the elements\n");
```

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

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

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

      while((temp<a[j])&&(j>=0))
      {
        a[j+1]=a[j];   //moves element forward
        j=j-1;
      }
      a[j+1]=temp;   //insert element in properplace
    }

    printf("\nSorted list is as follows\n");
    for(i=0;i<n;i++)
    {
      printf("%d ",a[i]);
    }

    return 0;
}
```

Algorithm complexity through example programs

Algorithm	Time Complexity		
	Best	**Average**	**Worst**
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$