



Problem Solving Using C

Course Code: 24BTPHY104

02/08/2024

TABLE OF CONTENTS

Module 1	4
1. INTRODUCTION TO PROGRAMMING CONCEPTS	4
1.1 Computer Languages	4
1.2 Programming Language Translators	6
1.3 Software	6
1.4 Modular Approach in Programming	8
1.5 Structured programming	9
1.6 Algorithm	10
1.7 Flowchart	16
1.8 Overview of C Language	20
1.9 History of C	20
1.10 Character set of C language	22
1.11 C Tokens	23
1.12 Keywords	23
1.13 Identifiers	24
1.14 Constants	24
1.15 Data types	27
1.16 Variables	30
1.17 Format Specifiers	33
1.18 Operators and Expressions	38
1.19 C Operators	40
1.20 Type Conversion	59
1.21 Library functions in C	62
Module 2	65
2. MANAGING INPUT AND OUTPUT OPERATIONS	65
2.1 Unformatted I/O statements	65
2.2 Formatted I/O Functions	68
2.3 Decision Making Statements	72
2.4 Ternary operator	90
2.5 Looping structure in C	91
2.6 Break statement	97
2.7 Continue statement	97
2.8 Goto statement	98
2.9 C Programs	99

Module 3	102
3.ARRAYS – STRINGS – STORAGE CLASSES	102
3.1 Arrays	102
3.2 Single Dimensional Array.	103
3.3Two Dimensional arrays (Multidimensional array).....	109
3.4 Strings	115
3.5 Storage Classes	121
Module 4	125
4.FUNCTIONS AND POINTERS	125
4.1 Function Definition.....	127
4.2 Function Prototyping	128
4.3 Types of functions	130
4.4 Passing arguments to functions	134
4.5 Passing arrays to functions	134
4.6 Passing strings to functions	137
4.7 Nested Functions.....	139
4.8 Call by value	140
4.9 Call by reference.....	141
4.10 Recursive functions.....	142
4.11 Pointers	145
4.12 Pointers and Arrays.....	158
4.13 Arrays of Pointers	161
4.14 Pointers and Structures	163
4.15 Meaning of static and dynamic memory allocation	164
4.16 Memory Allocation Functions	165
4.17 Macros.....	176
Module 5	184
5. STRUCTURES AND UNIONS.....	184
5.1 Structures.....	184
5.2 Unions	186
5.3 Nested Structures in C.....	188
5.4 Arrays of Structures in C	191
5.5 Passing structures to functions.....	195
5.6 typedef in C.....	199
5.7 Enum (Enumerated Data type) in C.....	201

5.8 Bit Fields in C	203
5.9 Command Line arguments	205
5.10 C pre-processor directives	206
5.11 Files in C	208
5.11.3 Text and Binary File	218

Course: Problem Solving Techniques using C**Course Code: 24BTPHY104****Module 1****1. INTRODUCTION TO PROGRAMMING CONCEPTS****Programming Language**

A language that is acceptable to a computer system is called a computer language or programming language and the process of creating a sequence of instructions in such a language is called programming or coding. A program is a set of instructions, written to perform a specific task by the computer. A set of large programs is called software. To develop software, one must have knowledge of a programming language.

A programming language is a set of symbols, grammars and rules with the help of which one is able to translate algorithms to programs that will be executed by the computer. The programmer communicates with a machine using programming languages.

Before moving on to any programming language, it is important to know about the various types of languages used by the computer. Let us first know what the basic requirements of the programmers were & what difficulties they faced while programming in that language.

1.1 Computer Languages

Languages are a means of communication. Normally people interact with each other through a language. On the same pattern, communication with computers is carried out through a language. This language is understood both by the user and the machine. Just as every language like English, Hindi has its own grammatical rules; every computer language is also bounded by rules known as syntax of that language. The user is bound by that syntax while communicating with the computer system.

Computer languages are broadly classified as:

1.1.1 Low Level Language: The term low level highlights the fact that it is closer to a

language which the machine understands.

The low-level languages are classified as:

Machine Language: This is the language (in the form of 0's and 1's, called binary numbers) understood directly by the computer. It is machine dependent. It is difficult to learn and even more difficult to write programs.

Assembly Language: This is the language where the machine codes comprising of 0's and 1's are substituted by symbolic codes (called mnemonics) to improve their understanding. It is the first step to improve programming structure. Assembly language programming is simpler and less time consuming than machine level programming, it is easier to locate and correct errors in assembly language than in machine language programs. It is also machine dependent. Programmers must have knowledge of the machine on which the program will run.

1.1.2 High Level Language: Low level language requires extensive knowledge of the hardware since it is machine dependent. To overcome this limitation, high level language has been evolved which uses normal English, which is easy to understand to solve any problem. High level languages are computer independent and programming becomes quite easy and simple. Various high-level languages are given below:

- a. BASIC (Beginners All Purpose Symbolic Instruction Code): It is widely used, easy to learn general purpose language. Mainly used in microcomputers in earlier days.
- b. COBOL (Common Business Oriented language): A standardized language used for commercial applications.
- c. FORTRAN (Formula Translation): Developed for solving mathematical and scientific problems. One of the most popular languages among scientific community.
- d. C: Structured Programming Language used for all purpose such as scientific application, commercial application, developing games etc.
- e. C++: Popular object-oriented programming language, used for general purpose.

1.2 Programming Language Translators

As you know that high level language is machine independent and assembly language though it is machine dependent yet mnemonics that are being used to represent instructions are not directly understandable by the machine. Hence to make the machine understand the instructions provided by both the languages, programming language translators are used. They transform the instruction prepared by programmers into a form which can be interpreted & executed by the computer. Following are the various tools to achieve this purpose:

1.2.1 Compiler: The software that reads a program written in high level language and translates it into an equivalent program in machine language is called as compiler. The program written by the programmer in high level language is called source program and the program generated by the compiler after translation is called as object program.

1.2.2 Interpreter: it also executes instructions written in a high-level language. Both compiler & interpreter have the same goal i.e. to convert high level language into binary instructions, but their method of execution is different. The compiler converts the entire source code into machine level program, while the interpreter takes 1 statement, translates it, executes it & then again takes the next statement.

1.2.3 Assembler: The software that reads a program written in assembly language and translates it into an equivalent program in machine language is called as assembler.

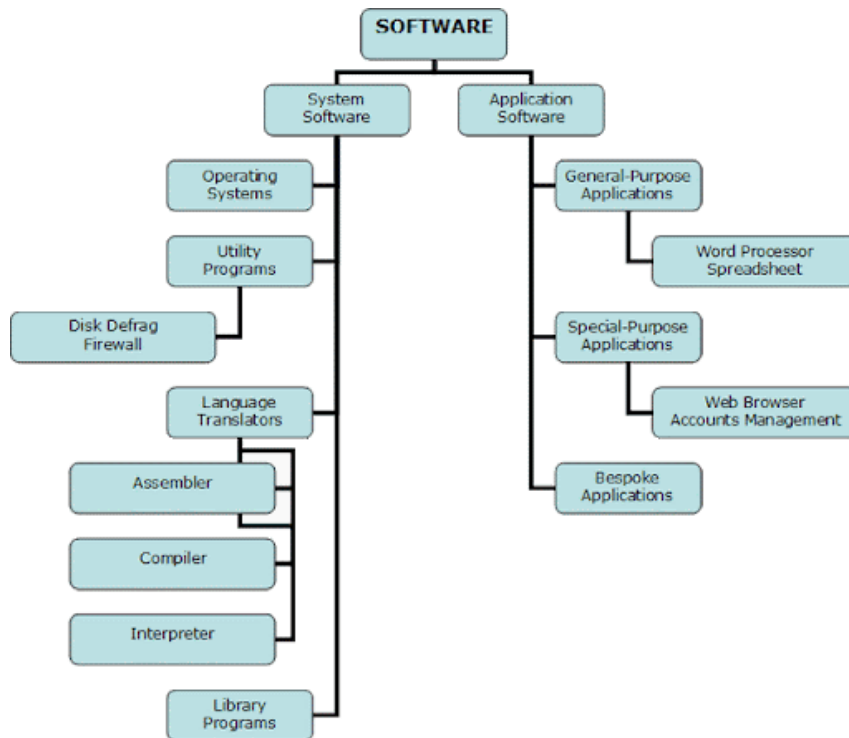
1.2.4 Linker: A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

1.3 Software

Software is a set of instructions, data or programs used to operate computers and execute specific tasks. It is the opposite of hardware, which describes the physical aspects of a computer. Software is a generic term used to refer to applications, scripts and programs that run on a device.

This software is further classified as system software & application software

1.3.1 Classification of software



System Software:

System Software is necessary to manage computer resources and support the execution of application programs. Software like operating systems, compilers, editors and drivers, etc., come under this category. A computer cannot function without the presence of these. Operating systems are needed to link the machine-dependent needs of a program with the capabilities of the machine on which it runs. Compilers translate programs from high-level language to machine language.

Application Software:

Application software is designed to fulfill the user's requirement by interacting with the user directly. It could be classified into two major categories are generic or customized. Generic Software is software that is open to all and behaves the same for all of its users. Its function is limited and not customized as per the user's changing requirements. However, on the other hand, customized software is the software products designed per the client's requirement, and are not available for all.

System Software	Application Software
System software is used for operating the computer hardware.	The user uses application software to perform a specific task.
System software is installed on the computer when the operating system is installed.	Application software is installed according to user requirements.
System software provides the platform for running application software.	Application software cannot run without the presence of system software.
System software works/runs independently.	At the same time, application software cannot work or run independently.
The user does not interact with system software because it works in the background.	In the application software, the user interacts directly.
System software runs automatically when the computer is turned ON and STOP when the computer shutdown.	Application software runs when the user requests it.

1.4 Modular Approach in Programming

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of modular programming approached.

- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

Points which should be taken care of prior to modular program development:

- Limitations of each and every module should be decided.
- In which way a program is to be partitioned into different modules.
- Communication among different modules of the code for proper execution of the entire program.

Advantages of Using Modular Programming Approach –

- **Ease of Use:** This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines of code in one go we can access it in the form of modules. This allows ease in debugging the code and is prone to less error.
- **Reusability:** It allows the user to reuse the functionality with a different interface without typing the whole program again.
- **Ease of Maintenance:** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

1.5 Structured programming

Structured programming, or modular programming, is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming, but the mechanisms of support -- like the syntax of the programming languages -- vary.

In Structured Programming, the code is divided into functions or modules. It is also known as modular programming. Modules or functions are a set of statements which performs a sub task. As each task is a separate module, it is easy for the programmer to test and debug. It is also easy to do modifications without changing the whole program.

When changing the code, the programmer has to concentrate only on the specific module. C and Pascal are some examples of Structural Programming languages.

Unstructured Programming

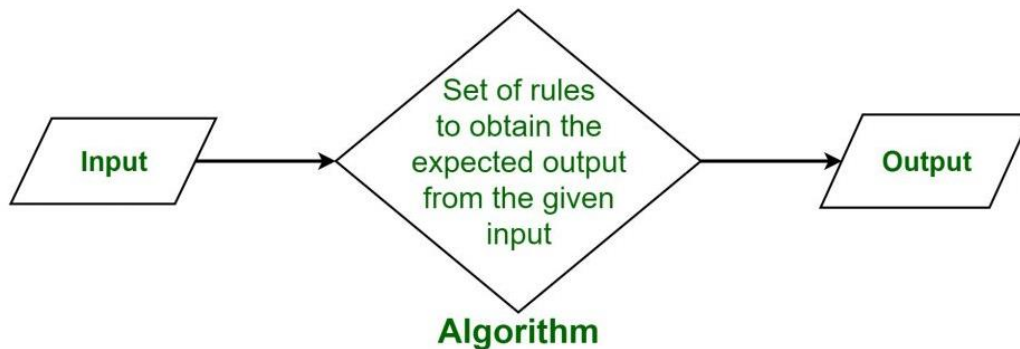
In Unstructured Programming, the code is written as a single whole block. The whole program is taken as a single unit. It is harder to do changes in the program. This paradigm was used in earlier versions of BASIC, COBOL, and FORTRAN. Unstructured programming languages have a limited number of data types like numbers, arrays, strings.

<i>Structure oriented</i>	<i>Object oriented</i>	<i>Non structure</i>
In this type of language, large programs are divided into small programs called functions	In this type of language, programs are divided into objects	There is no specific structure for programming this language
Prime focus is on functions and procedures that operate on the data	Prime focus is in the data that is being operated and not on the functions or procedures	N/A
Data moves freely around the systems from one function to another	Data is hidden and cannot be accessed by external functions	N/A
Program structure follows “Top Down Approach”	Program structure follows “Bottom UP Approach”	N/A
Examples: C, Pascal, ALGOL and Modula-2	C++, JAVA and C# (C sharp)	BASIC, COBOL, FORTRAN

1.6 Algorithm

The word “algorithm” relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique. Software Engineer commonly uses an algorithm for planning and solving the problems. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time

What is Algorithm?



Algorithm has the following characteristics.

- **Input:** An algorithm may or may not require input
- **Output:** Each algorithm is expected to produce at least one result
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps

The algorithm and flowchart include following three types of control structures.

1. **Sequence:** In the sequence structure, statements are placed one after the other and the execution takes place starting from up to down.
2. **Branching (Selection):** In branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. In the case of TRUE, one of the two branches are explored; but in the case of FALSE condition, the other alternative is taken. Generally, the 'IF-THEN' is used to represent branch control.
3. **Loop (Repetition):** The Loop or Repetition allows a statement(s) to be executed repeatedly based on certain loop condition e.g. WHILE, FOR loops.

Advantages of algorithm

- It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
- An algorithm uses a definite procedure.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.

Before writing an algorithm for a problem, one should find out what is/are the inputs to the algorithm and what is/are expected output after running the algorithm. Now let us take some exercises to develop an algorithm for some simple problems: While writing algorithms we will use following symbol for different operations:

'+' for Addition

'-' for Subtraction

'*' for Multiplication

'/' for Division and

'=' for assignment. For example, $A = X * 3$ means A will have a value of $X * 3$.

Example of Algorithm

Problem 1: Find the area of a Circle of radius r.

Inputs to the algorithm:

Radius r of the Circle.

Expected output:

Area of the Circle

Algorithm:

Step1: Read\input the Radius r of the Circle

Step2: $Area = \pi * r * r$ // calculation of area

Step3: Print Area

Step4: End

Problem2: Write an algorithm to read two numbers and find their sum.

Inputs to the algorithm:

First num1.

Second num2.

Expected output:

Sum of the two numbers.

Algorithm:

Step1: Start

Step2: Read\input the first num1.

Step3: Read\input the second num2.

Step4: Sum num1+num2 // calculation of sum

Step5: Print Sum

Step6: End

Problem3: Write an algorithm to find the largest of three numbers.

Inputs to the algorithm:

First num1.

Second num2.

Third num3.

Expected output:

Largest of three numbers.

Algorithm:

Step1: Start

Step2: Read\input the first num1.

Step3: Read\input the second num2.

Step4: Read\input the third num3.

Step5: If a>b

 If a>c

 Display a is the largest number.

```

    Else
        Display c is the largest number.
Else
    If(b>c)
        Display b is the largest number
    Else
        Display c is the largest number
Step5: Stop
    
```

Problem 4: An algorithm to calculate even numbers between 0 and 99

1. Start
2. $I \leftarrow 0$
3. Write I in standard output
4. $I \leftarrow I+2$
5. If ($I \leq 98$) then go to line 3
6. End

Problem5: Design an algorithm which gets a natural value n as its input and calculates odd numbers equal or less than n. Then write them in the standard output:

1. Start
2. Read n
3. $I \leftarrow 1$
4. Write I
5. $I \leftarrow I + 2$
6. If ($I \leq n$) then go to line 4
7. End

Problem6: Design an algorithm which generates even numbers between

1000 and 2000 and then prints them in the standard output. It should also print total sum:

1. Start
2. $I \leftarrow 1000$ and $S \leftarrow 0$
3. Write I
4. $S \leftarrow S + I$
5. $I \leftarrow I + 2$
6. If ($I \leq 2000$) then go to line 3
else go to line 7
7. Write S
8. End

Properties of Algorithm

Donald Ervin Knuth has given a list of five properties for a, algorithm, these properties are:

- 1) Finiteness:** An algorithm must always terminate after a finite number of steps. It means after every step one reaches closer to the solution of the problem and after a finite number of steps algorithm reaches to an end point.
- 2) Definiteness:** Each step of an algorithm must be precisely defined. It is done by well thought actions to be performed at each step of the algorithm. Also, the actions are defined unambiguously for each activity in the algorithm.
- 3) Input:** Any operation you perform needs some beginning value/quantities associated with different activities in the operation. So, the value/quantities are given to the algorithm before it begins.
- 4) Output:** One always expects output/result (expected value/quantities) in terms of output from an algorithm. The result may be obtained at different stages of the algorithm. If some result is from the intermediate stage of the operation then it is known as intermediate result and result obtained at the end of the algorithm is known as end result. The output is expected value/quantities always have a specified relation to the inputs
- 5) Effectiveness:** Algorithms to be developed/written using basic operations. Actually,

operations should be basic, so that even they can in principle be done exactly and in a finite amount of time by a person, by using paper and pencil only.

1.7 Flowchart

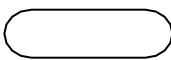

The first design of flowchart goes back to 1945 which was designed by John Von Neumann. Unlike an algorithm, Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flow chart one can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

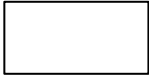
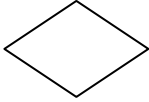
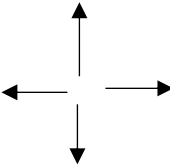



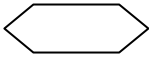
Advantages of flowchart:

- Flowchart is an excellent way of communicating the logic of a program.
- Easy and efficient to analyze problem using flowchart.
- During program development cycle, the flowchart plays the role of a blueprint, which makes program development process easier.
- After successful development of a program, it needs continuous timely maintenance during the course of its operation. The flowchart makes program or system maintenance easier.

It is easy to convert the flowchart into any programming language code

Flowchart is diagrammatic /Graphical representation of sequence of steps to solve a problem. To draw a flowchart following standard symbols are used

Symbol Name	Symbol	Function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation

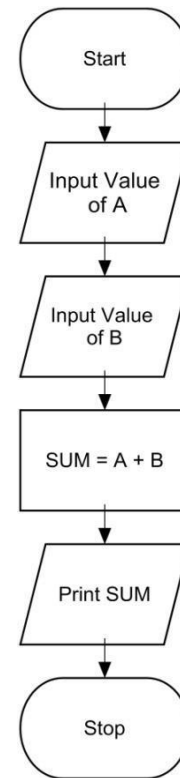
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
Off page connector		Connects two parts of a flowchart which are spread over different pages
Predefined Process		Used to invoke a subroutine or an interrupt program
Hexagon		Preparation Indicates preparation taken for the following step

Algorithm

Step - 1 Start

Step - 2 Input first numbers say A Step – 3
Input second number say B Step - 4
 $SUM = A + B$

Step – 5 Display SUM Step – 6 Stop



Algorithm & Flowchart to convert temperature from Celsius to Fahrenheit

C: temperature in Celsius

F: temperature
Fahrenheit

Algorithm

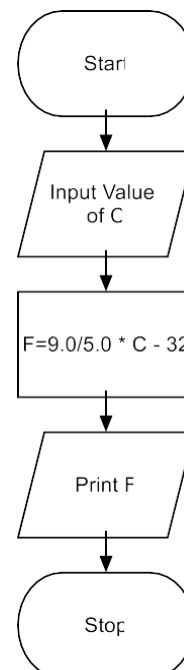
Step -1 Start

Step -2 Input temperature in Celsius C

Step - 3 $F = (9.0/5.0 \times C) + 32$

Step - 4 Display Temperature in Fahrenheit
F

Step - 5 Stop



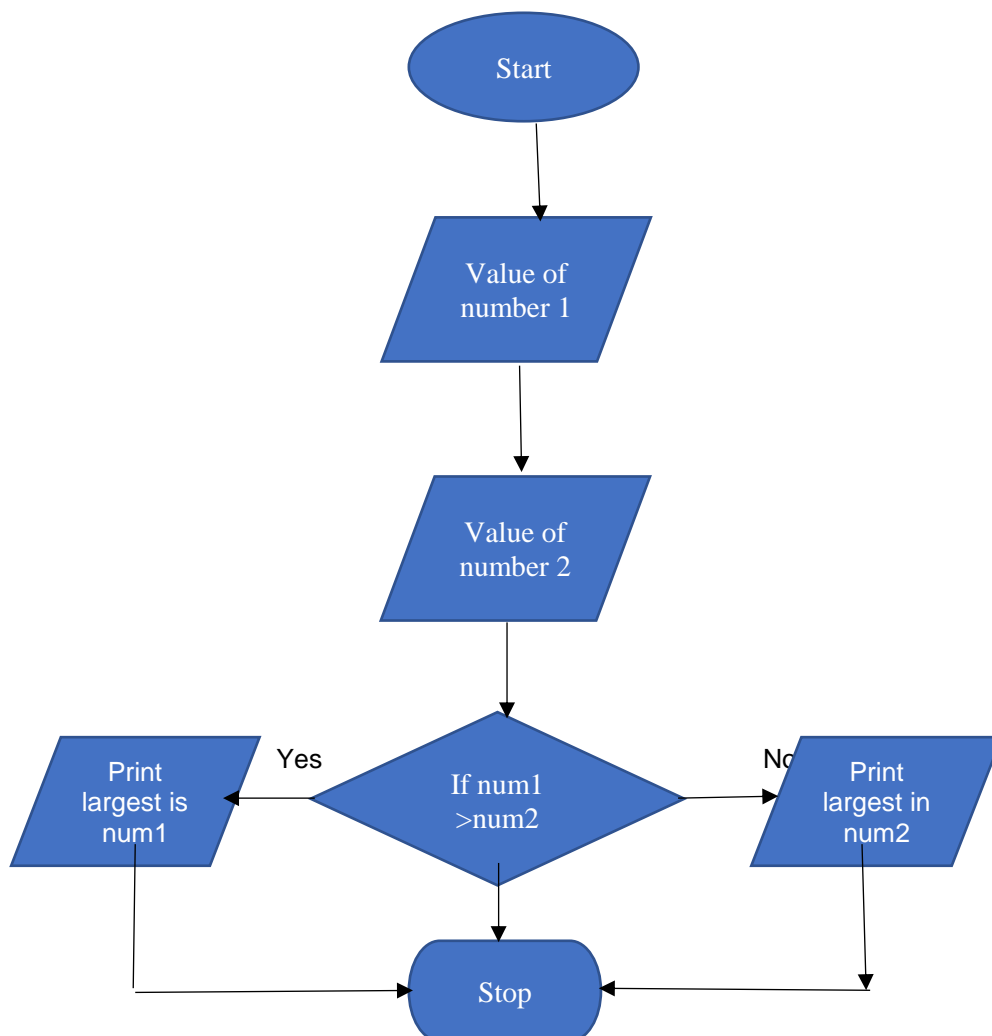
Algorithm & Flowchart to find the largest of two numbers**Algorithm**

Step-1 Start

Step-2 Input two numbers say
NUM1, NUM2

Step-3 IF NUM1 > NUM2 THEN
print largest is NUM1 ELSE
print largest is NUM2

Step-4 Stop



Algorithm & Flowchart to find Even number between 1 to 50**Algorithm**

Step-1 Start

Step-2 $I = 1$

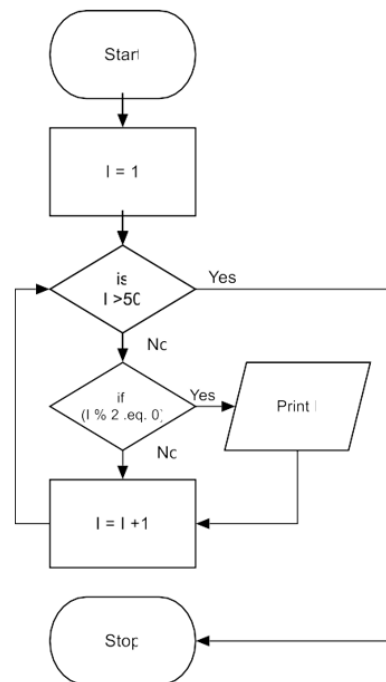
Step-3 IF ($I > 50$) THEN
 GO TO Step-7
 ENDIF

Step-4 IF ($(I \% 2) = 0$) THEN
 Display I
 ENDIF

Step-5 $I = I + 1$

Step-6 GO TO Step-3

Step-7 Stop



1.8 Overview of C Language

1.9 History of C

- C is a general-purpose programming language developed by **Dennis Ritchie** at AT&T Bell laboratories in 1972.

Advantages/Features of C Language

C language is very popular language because of the following features:

1. C is structured Programming Language
2. It is considered a high-level language because it allows the programmer to solve a problem without worrying about machine details.
3. It has a wide variety of operators using which a program can be written easily to solve a given problem.
4. C is more efficient which increases the speed of execution and management of memory compared to low level languages.
5. C is machine independent. The program written on one machine will work on another machine.

6. C can be executed on many different hardware platforms.

1.1 Pseudocode: A solution to Problem

- **Definition:**

- It is a series of steps to solve a given problem written using a mixture of English and C language.
- It acts as a problem-solving tool.
- It is the first step in writing a program.

Purpose:

- Is to express a solution to a given problem using a mixture of English language and c like code.

- **Advantage:**

- Easy to write and understand
- It is relatively easy to convert English description solutions of small programs to C programs.

Ex 1: Addition of two numbers

- Get the numbers [a, b]
- Compute addition [Sum= a + b]
- Print the results [Sum]

Ex 2: Area of Circle

1. Get the radius[r]
2. Compute area[Area = 3.141*r*r]
3. Print the results [Area]

Disadvantage:

- It is very difficult to translate the solution of lengthy and complex problem in English to C

C Programming Concepts

- **Program:** A program is a group of instructions given by the programmer to perform a specific task.

1.10 Character set of C language

➤ **Definition:** A symbol that is used while writing a program is called a character.

➤ A character can be:

□ **Alphabets/Letters (Lowercase a-z, Uppercase A-Z)**

□ **Digits (0-9)**

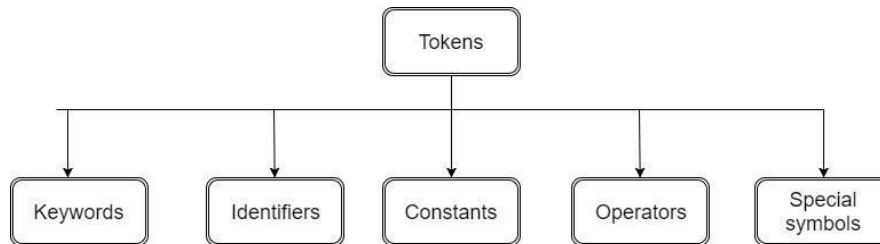
□ **Special Symbols (~ ' ! @ # % & * () - + / \$ = \ {**

} [] : ; " " ? etc)

Symbol	Name	Symbol	Name	Symbol	Name
~	Tilde		Vertical bar	[Left bracket
#	Hash	(Left parenthesis]	Right bracket
\$	Dollar sign)	Right parenthesis	:	Colon
^	Caret	+	Plus, sign	;	Semicolon
&	Ampersand	{	Left brace	<	Less than
*	Asterisk	}	Right brace	>	Greater than
'	Single quote	.	Dot	=	Assignment
,	Comma	\	backslash	/	Division

1.11 C Tokens

- Tokens are the smallest or basic units of a C program.
- One or more characters are grouped in sequence to form meaningful words. These meaningful words are called tokens.
- A token is a collection of **characters**.
- Tokens are classified into **5 types** as below:



1.12 Keywords

- The tokens which have predefined meaning in C language are called **keywords**.
- They are reserved for specific purposes in C language they are called **Reserved Words**.
- There are totally **32 keywords** supported in C they are:

Auto	double	if	Static
Break	else	int	Struct
Case	enum	long	Switch
Char	extern	near	Typedef
Const	float	register	Union
Continue	for	return	Unsigned
Default	volatile	short	Void
Do	goto	signed	While

Rules for keywords

1. Keywords should not be used as variables, function names, array names etc.
2. All keywords should be written in lowercase letters. Keywords meaning cannot be changed by the users.

1.13 Identifiers

Definition:

- Identifiers are the names given to program elements such as variables, constants, function names, array names etc.
- It consists of one or more letters or digits or underscores.

Rules for identifiers

1. The First character should be an **alphabet** or an **underscore** _
Then the First character is followed by any number of **letters or digits**.
No extra symbols are allowed other than **letters, digits and Underscore**
2. Keywords cannot be used as an identifier
3. The length can be **31 characters** for external, **63** for internal.
4. Identifiers are case sensitive

Identifier	Reasons for invalidity
india06	Valid
_india	Valid
india_06	Valid
india_06_king	Valid
_india	valid
06india	not valid as it starts from digits
int	not valid since int is a keyword
india 06	not valid since there is space between india and 06
india@06	not valid since @ is not allowed

1.14 Constants

Definition:

Constants refers to **fixed values** that **do not change** during the execution of a program
The different types of constants are:

1. Integer constant
2. Real constant/Floating Pointing constant
3. Enumeration constant
4. Character constant
5. String constant

Integer constant

- Definition: An integer is whole number without any decimal point. no extra characters are allowed other than + or _ .

Three types of integers

- **Decimal integers:** are constants with a combination of digits 0 to 9, optional + or -

Ex: 123, -345, 0, 5436, +79

- **Octal integers:** are constants with a combination of Digits from 0 to 7 but it has a prefix of 0

Ex: 027, 0657, 0777645

- **Hexadecimal integers:** Digits from 0 to 9 and characters from a to f, it has to start with 0X or 0x

Ex: 0X2 0x56 0X5fd 0xbdae

Real constants/Floating point

Floating point constants are base 10 numbers that are represented by fractional parts, such as 10.5. They can be positive or negative.

Two forms are:

i. Fractional form:

- A floating-point number represented using fractional form has an integer part followed by dot and a fractional part.

Ex: 0.0083

- 215.5

- -71.

- +0.56 etc...

ii. Exponential form:

- A floating-point number represented using **Exponent form has 3 parts**
- **mantissa e exponent**
- Mantissa is either real number expressed in decimal notation or integer.
- **Exponent must be integer with optional + or –**
- Ex 9.86 E 3 => 9.86*10³
- Ex 9.86 E -3 => 9.86*10⁻³

Enumeration constant

A set of named integer constants defined using the keyword enum are called enumeration constants.

Syntax:

enum identifier{enum list};

enum Boolean{NO,YES}; NO is assigned with 0

YES is assigned with value 1

enum days{ mon,tue,wed};

mon is assigned with 0

tue is assigned with 1

wed is assigned with

2

Character constant

A symbol enclosed within pair of single quotes is called character constant.

Each character is associated with unique value called ASCII value.

Ex: '9'

'\$'

Backslash constants (Escape sequence character)

- Definition: An escape sequence character begins with backslash and is followed by one character.
- A backslash along with a character give rise to special print effects.
- It always starts with backslash; hence they are called as backslash constants.

Character	Name	Meaning
\a	Bell	Beep sound

\b	Backspace	Cursor moves towards left by one Position
\n	Newline	Cursor moves to next line
\t	Horizontal tab	Cursor moves towards right by 8 Position
\r	Carriage return	Cursor moves towards beginning of the same line
\"	Double quotes	Double quotes
\'	Single quotes	Single quotes
\?	Question mark	Question mark
\\	Backslash	Backslash
\0	NULL	

String constant

A sequence of characters enclosed within pair of double quotes is called string constant.

The string always ends with a NULL character.

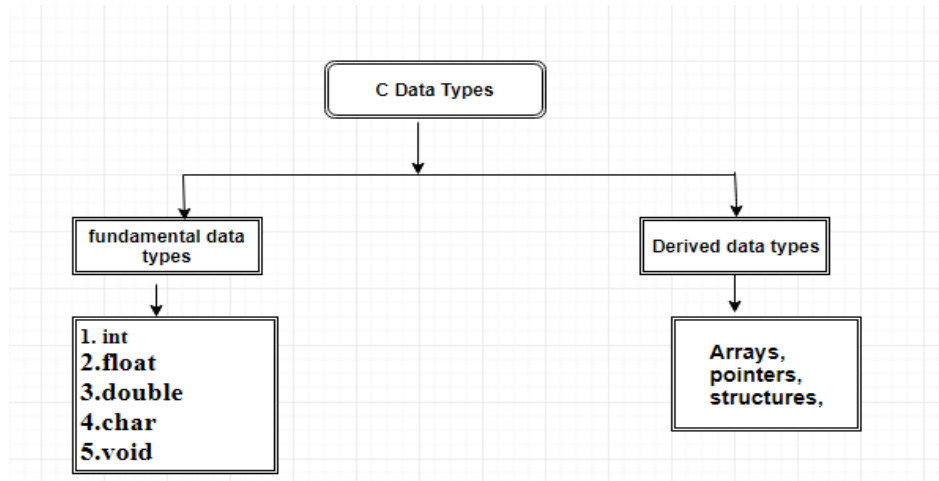
Ex: "9"

"SNPS"

1.15 Data types

Definition: The data type defines the type of data stored in memory location or the type of data the variable can hold.

- The classification of data types are:



➤ There are few basic data types supported in C.

1. **int**
2. **float**
3. **double**
4. **char**
5. **void**

Integer data type(int)

- It stores an integer value or integer constant.
- An int is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as integer.
- C supports 3 different types of integer:
 - short int,
 - int,
 - long int

n-bit machine	Type	Size	Range of unsigned int 0 to 2^n-1	Range of signed int be -2^n to $+2^n-1$
16-bit Machine	short int	2 bytes	0 to $2^{16}-1$ 0 to 65535	-2^{15} to $+2^{15}-1$ -32768 to +32767

	Int	2 bytes	0 to $2^{16}-1$ 0 to 65535	-2^{15} to $+2^{15}-1$ -32768 to +32767
	long int	4 bytes	0 to $2^{32}-1$ 0 to 4294967295	-2^{31} to $+2^{31}-1$ -2147483648 to + 2147483647

Floating data type (float)

A float is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as floating point number.

The size is 4 bytes.

The range is **-3.4e38 to +3.4e38**.

Float can hold the real constant accuracy up to 6 digits after the decimal point.

Double data type (double)

A double is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as long floating-point number.

The size is 8 bytes.

The range is from **1.7e-308 to 1.7 e+308**.

Double can hold real constant up to 16 digits after the decimal point.

Character data type (char)

Char is a keyword which is used to define single character or a sequence of character in c language capable of holding one character from the local character set.

The size of the character variable is 1 byte.

The range is from **-128 to +127**.

Each character stored in the memory is associated with a unique value termed as ASCII (American Standard Code for Information Interchange).

It is used to represent character and strings.

n-bit machine	Type	size	Range of unsigned char 0 to 2^n-1	Range of signed char be -2^n to $+2^n-1$
16-bit machine	short int	2 bytes	0 to 2^8-1 0 to 255	-2^7 to $+2^7-1$ -128 to +127

void data type (void)

It is an empty data type.

No memory is allocated.

Mainly used when there are no return values.

1.16 Variables

- A variable is a name given to memory location where data can be stored.
- Using variable name data can be stored in a memory location and can be accessed or manipulated very easily.

- **Rules for variables**

- The First character should be an **alphabet** or an **underscore** _
- Then the First character is followed by any number of **letters or digits**.
- No extra symbols are allowed other than **letters, digits and Underscore**
- Keywords cannot be used as an identifier

Example:

Sum –

valid For1-

valid

for -invalid (it is a keyword)

Declaration of variables:

- Giving a name to a memory location is called **declaring a variable**.
- Reserving the required memory space to store the data is called **defining a variable**.
- General Syntax:

data type variable; (or)

datatype variable1, variable2,... variableN;

example: **int a;**
 float x, y;
 double sum;

From the examples we will come to know that “a” is a variable of type integer and allocates 2 bytes of memory. “x” and “y” are two variables of type float which will be allocated 4 bytes of memory for each variable. “sum” is a double type variable which will be allocated with 8 bytes of memory for each.

Variable Initialization

- Variables are not initialized when they are declared and defined, they contain garbage values (meaningless values)
- The method of giving initial values for variables before they are processed is called variable initialization.
- General Syntax:

Var_name = expr;

Where,

Var_name is the name of the variable.

expr is the value of the expression.

- The “expr” on the right-hand side is evaluated and stored in the variable name (Var_name) on the left-hand side.
- The expression on the right-hand side may be a constant, variable or a larger formula built from simple expressions by arithmetic operators.

Examples:

- `int a=10;`
 // assigns the value 10 to the integer variable a
- `float x;`
 `x=20;`
 // creates a variable y of float type and assigns value 20 to it.

- `int a=10,b,b=a;`
`// creates two variables a and b. "a" is assigned with value 10, the value of "a" is assigned to variable "b". Now the value of b will be 10.`
- `price = cost*3;`
`//assigns the product of cost and 3 to price.`
- `Square = num*num;`
`// assigns the product of num with num to square.`

OUTPUT FUNCTION

Displaying Output using printf

- `printf` is an output statement in C used to display the content on the screen.
- `print`: Print the data stored in the specified memory location or variable.
- `Format`: The data present in memory location is formatted in to appropriate data type.
- There are various forms of `printf` statements.

Method 1: `printf(" format string");`

- `Format string` may be any character. The characters included within the double quotes will be displayed on the output screen
- Example: `printf("Welcome to India");`

Output:

Welcome to India

Method 2:

`printf(" format string", variable list);`

- `Format string` also called as `control string`.
- `Format string` consist of **format specifier** of particular data type

- Format specifiers start with % sign followed by **conversion code**.
- variable lists are the **variable names** separated by **comma**.
- Example:

```
int a=10;
float
b=20;
printf(" integer =%d, floating=%f",a,b);
```

output:

integer=10, floating=20.00000

- Number of format specifiers must be equal to the number of variables.
- While specifying the variable name make sure that it matches the **format specifiers** within the double quotes.

1.17 Format Specifiers

- Format specifiers are the character string with % **sign** followed with a character.
- It specifies the type of data that is being processed.
- It is also called **conversion specifier or conversion code**.
- There are many format specifiers defined in C.

Symbols	Meaning
%d	Decimal signed integer number
%f	float point number
%c	Character
%o	octal number
%x	hexadecimal integer (Lower case letter x)

%X	hexadecimal integer (Upper case letter X)
%e	floating point value with exponent (Lower case letter e)
%E	floating point value with exponent (Upper case letter E)
%ld	long integer
%s	String
%lf	Double

Input Function (scanf)

Inputting Values Using scanf

- To enter the input through the input devices like keyboard we make use of scanf statement.

General Syntax:

```
scanf("format string", list of address of variables);
```

Where, Format string consists of the access specifiers/format specifiers.

- Format string also called as control string.
- Format string consist of format specifier of particular data type
- Format specifiers starts with % sign followed by conversion code.
- List of addresses of variables consist of the variable name preceded with & symbol (address operator).

Example: int a;
 float
 b;
 scanf("%d%f",&a,&b);

Rules for scanf

No escape sequences or additional blank spaces should be specified in the format specifiers.

- Ex: **scanf("%d %f",&a,&b);** //invalid
 scanf("%d\n%f",&a,&b); //invalid
- & symbol is must to read the values, if not the entered value will not be stored in the variable specified. Ex: **scanf("%d%f",a,b);**//invalid.

A Simple C Program

Example 1: Let us discuss a simple program to print the Hello SNPS

/*program to print Hello snps*/

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

Output: Hello SNPS

Example 2: Consider another example program to find the simple interest

```
#include<stdio.h>
void main ( )
{
    float p,t,r;
    float si;
    printf ("enter the values of p,t,r\n");
    scanf("%f%f%f",&p,&t,&r);
    si = (p*t*r)/100;
    printf ("Simple Interest=%f",si);
}
```

The above program illustrates the calculation of simple interest.

- Firstly, we have declared the header files `stdio.h` for including standard input and output which will be `printf` and `scanf` statements.
- Next, we start with the main program indicated by `void main()`
- Within the main program we declare the required variables for calculating the simple interest. So, we declare `p`, `t` and `r` as float type and `si` as float type.
- After which the values to `p`, `t` and `r` are read from keyboard using `scanf`.
- Computed the simple interest by the formula $(p*t*r)/100$ and the result is assigned to `si`.
- Display the result `si`.

Structure of C Program

Comments/Documentation Section
Preprocessor Directives
Global declaration section
<pre>void main() [Program Header] { Local Declaration part Execution part Statement 1 ----- ----- Statement n }</pre>
User defined Functions

- **Comments/Documentation Section**

- Comments are short explaining the purpose of the program or a statement.
- They are non-executable statements which are ignored by the compiler.

The comment begins **with /* and ends with */**.

- The symbols /* and */ are called **comment line delimiters**.
- We can put any message we want in the comments.

Example: /* Program to compute Quadratic Equation */

Note: Comments are ignored by C compiler, i.e. everything within comment delimiters

- **Preprocessor Directives**

- Preprocessor Directives begins with a # symbol
- Provides instructions to the compiler to include some of the files before compilation starts.
- Some examples of header files are

```
#include <stdio.h>
#include
<conio.h>
#include
<string.h>
#include <math.h>
#include <stdlib.h> Etc...
```

- **Global Declaration Section**

- There will be some variable which has to be used in anywhere in program and used in more than one function which will be declared as global.

- **The Program Header**

- Every program must contain a main () function.
- The execution always starts from main function.
- Every C program must have only one main function.

- **Body of the program**

- Series of statements that performs specific task will be enclosed within braces { and }

- It is present immediately after program header. It consists of two parts

1. **Local Declaration part:** Describes variables of program

```
Ex: int sum =
    0; int a , b;
    float b;
```

2. **Executable part:** Task done using statements.
3. All statements should end with semicolon(;)

Subroutine Section: All user defined functions that are called in main function should be defined.

1.18 Operators and Expressions

Operators:

- “An operator is a symbol that specifies the operation to be performed on various types of operands”. Or in other words “An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations”.
- Operators are used in programs to manipulate data and variable.

Example: +, -, *

Operand:

- The entity on which a operator operates is called an operand. Here the entity may be a constant or a variable or a function.
- An operator may have one or two or three operands.

Example: $a+b-c*d/e\%f$

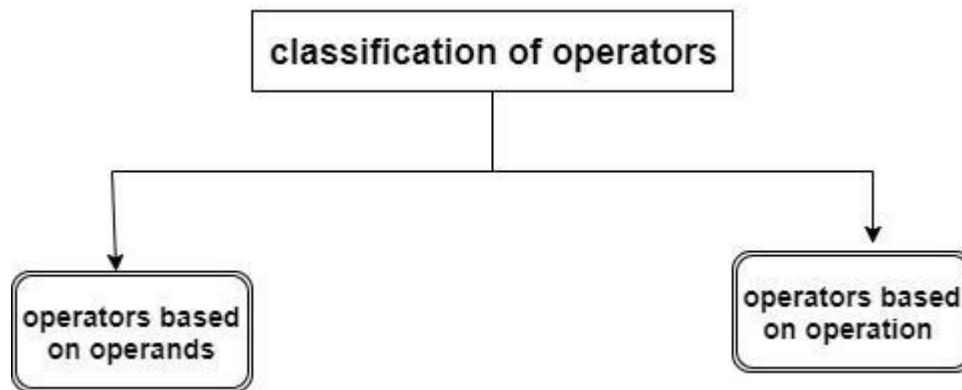
In this example there are 6 operands ie a, b, c, d, e, f and 5 operators i.e +, -, *, / and %.

Expression:

- A combination of operands and operators that reduces to a single value is called an expression.

Eg: $a+b/d$

Classification of operators



Classification of operators based on operands

- The Operators are classified into four major categories based on the number of operands as shown below:
- Unary Operators
 - 1) Binary Operators
 - 2) Ternary Operators
 - 3) Special Operators

1. Unary Operators: An operator which acts on only one operand to produce the result is called unary operator. The Operators precede the operand.

Examples: -10
 -a
 --b

2. Binary Operators: An operator which acts on two operands to produce the result is called a binary operator. In an expression involving a binary operator, the operator is in between two operands.

Examples: $a + b$ $a * b$

3. Ternary Operator: An operator which acts on three operands to produce a result is called a ternary operator.

- Ternary operator is also called a conditional operator.

Example: $a ? b : c$

- Since there are three operands a, b and c (hence the name ternary) that are associated with operators? and it is called ternary operator.

1.19 C Operators

C operators can be classified based on type of operation

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

Arithmetic operators

- C provides all basic arithmetic operators.

The operators that are used to perform arithmetic operation such as addition, subtraction, multiplication, division etc are called arithmetic operators.

- The operators are +, -, *, /, %.
- Let a=10, b=5. Here, a and b are variables and are known as operands.

Description	Symbo l	Example	Res ult	Priorit y	Associativity
Addition	+	$a+b = 10 + 5$	15	2	L-R
Subtraction	-	$a-b = 10 - 5$	5	2	L-R

Multiplication	*	$a * b = 10 * 5$	50	1	L-R
Division (Quotient)	/	$a/b = 10/5$	2	1	L-R
Modulus (Remainder)	%	$a \% b = 10 \% 5$	0	1	L-R

/*Program to illustrate the use of arithmetic operators is given below*/

```
#include<stdio.h>
void main()
{
    int a, b, sum,diff,mul,divres,modres;
    printf("enter a and b value\n");
    scanf("%d %d",&a,&b);
    sum=a+b;
    diff=a-b;
    mul=a*b;
    divres=a/b;
    modres=a%
    b;
    printf("%d, %d, %d ,%d ,%d\n", sum,diff,mul,divres,modres);
}
```

Output: 35,25,150,6,0

Converting Mathematical Expression into C Expression

Mathematical Expression	C expression
$a \div b$	a/b
$S = \frac{a+b+c}{2}$	$S=(a+b+c)/2$
$Area = \sqrt{s(s-a)(s-b)(s-c)}$	$Area=sqrt(s*(s-a)*(s-b)*(s-c))$
ax^2+bx+c	$a*x*x+b*x+c$
$e^{ a }$	$exp(abs(a))$

Relational Operators

- We often compare two quantities depending on their relation, take certain decisions.
- For example, we compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators.
- The operators that are used to find the relationship between two operands are called as relational expression.
- An expression such as
 $a < b$ or $1 < 20$
- The value of relational expression is either one or zero.
- It is **one** if the specified relation is **true** and zero if the relation is **false**. Example: $10 < 20$ is true.
 $20 < 10$ is false.

Relational operator in C

Operator	Meaning	Precedence	Associativity	Example	Result
<	is less than	1	L->R	$20 < 10$	F
<=	is less than or equal to	1	L->R	$20 <= 10$	F
>	is greater than	1	L->R	$20 > 10$	T
>=	is greater than or equal to	1	L->R	$20 >= 10$	T
==	is Equal to	2	L->R	$20 == 10$	F
!=	is not equal to	2	L->R	$20 != 10$	T

Logical Operators

- The operator that are used to combine two or more relational expression is called logical operators.
- Result of logical operators is always true or

false.

Logical Operators in C are:

Operator	Meaning	Precedence	Associativity
&&	logical AND	2	L->R
	logical OR	3	L->R
!	logical NOT	1	L->R

Logical AND (&&)

The output of logical operation is true if and only if both the operands are evaluated to true.

operand 1	AND	operand 2	result
True (1)	&&	True(1)	True(1)
True (1)	&&	False(0)	False(0)
False(0)	&&	True(1)	False(0)
False(0)	&&	False(0)	False(0)

Example:

36 && 0

=1 && 0

=0(False)

Logical OR (||)

The output of logical operation is false if and only if both the operands are evaluated to false.

operand 1	OR	operand 2	Result
True (1)		True(1)	True(1)
True (1)		False(0)	True(1)
False(0)		True(1)	True(1)
False(0)		False(0)	False(0)

Example:

36 || 0

=1 || 0

=0(True)

Logical NOT

It is a unary operator. The result is true if the operand is false and the result is false if the operand is true.

operand	! operand
True (1)	False(0)
False(0)	True (1)

Example:

(a) !0=1

(b) !36

!1=0

NOTE: The logical operators && and || are used when we want to test more than one condition and make decisions.

An example: a>b && x==10

The above logical expression is true only if a>b is true and x==10 is true. If either (or

both) of them are false, the expression is false.

Assignment Operators

An operator which is used to assign the data or result of an expression into a variable is called Assignment operators. The usual assignment operator is =.

Types of Assignment Statements:

1. Simple assignment statement
2. Short Hand assignment statement
3. Multiple assignment statement

1. Simple assignment statement

Syntax: Variable=expression;

Ex:

a= 10; //RHS is constant

a=b; // RHS is variable.

a=b+c; //RHS is an expression

So,

1. A expression can be a constant, variable or expression.
2. The symbol = is assignment operator
3. The expression on right side of assignment is evaluated and the result is converted into type of variable on left hand side of Assignment operator.
4. The converted data is copied into variable which is present on left hand side (LHS) of assignment operator.

Example:

```
int a;
```

```
a=100;    // the value 100 is assigned to variable a
```

```
int
```

```
a=100;
```

```
int b;
b=a;    // a value i.e 100 is assigned to b
```

2. Short Hand assignment Statement

C has a set of “**shorthand**” assignment operators of the form.

var op = exp;

where

v is a variable, exp is an expression and op is a C library arithmetic operator.

The operator **op=** is known as the shorthand assignment operator.

The assignment statements

var op= exp; is equivalent to $v = \text{var op}(\text{exp})$. with var evaluated only once.

The operators such as +=, -=, *=, /= are called assignment operators. Example:

The statement

$x = x + 10$; can be written as $x += 10$;

Example:

Solve the equation $x^* = y + z$ when $x=10$, $y=5$ and $z=3$ Solution: $x^* = y + z$ can be written as

$x = x^*(y + z)$

$x = 10^*(5 + 3)$

$x = 10^*8$

$x = 80$

shorthand assignment	Meaning	Description
$a += 2$	$a = a + 2$	Find $a + 2$ and store result in a

$a-=2$	$a=a-2$	Find $a-2$ and store result in a
$a*=2$	$a=a*2$	Find $a*2$ and store result in a
$a/=2$	$a=a/2$	Find $a/2$ and store result in a

3. Multiple Assignment Statement

A statement in which a value or set of values are assigned to more than one variable is called multiple assignment statement.

Example:

```
i=10;
```

```
j=10; k=10;
```

can be written as:

```
i=j=k=10; //Multiple assignment statement
```

Note: Assignment operator is right associative operator. Hence in the above statement 10 is assigned to k first, the k is assigned to j and j is assigned to i .

Increment and Decrement Operators

- C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

$++$ and $--$

- The operator $++$ adds 1 to the operand, while $--$ subtracts 1. The increment and decrement operators are Unary operators.
- The increment and decrement operators are classified into two categories:

1. Post Increment and Post Decrement:

- If increment or decrement operator is placed immediately after the operand is called Post Increment and Post Decrement.
- As the name indicates Post Increment and Post Decrement means increment or

decrement the operand value is used.

- So, operand value is used first and then the operand value is incremented or decremented by 1.

Example:

Post Increment: $a++$ equivalent to
 $a=a+1$ Post Decrement: $a--$
 equivalent to $a=a-1$

Post

Increment:

Eg1:

```
#include<stdio.h>

void main()
{
    int a=20;
    21. a++;
    printf("%d",a);
}
```

Eg2:

Output:21

Description: Initial value of a is 20 after execution of $a++$, the value of a will be

```
#include<stdio.h>

void main( )
{
    int a=20,b;
    b=a++;
    printf("a=%d",a);
    printf("b=%d",b);
}
```

Output : a= 21

b=20

Post

Decrement: Eg

1:

```
#include<stdio.h>

void main()
{
    int a=20;
```

Output:19

Description: Initial value of a is 20 after execution of $a--$, the value of a will be

```

19. a--;
    printf("%d",a);
}

```

2. Pre-Increment and Pre-Decrement

- If increment or decrement operator is placed before the operand is called Pre Increment and Pre Decrement.
- As the name indicates Pre-Increment and Pre-Decrement means increment or decrement before the operand value is used.
- So, the operand value is incremented or decremented by 1 first then operand value is used.

Example:

Post Increment: ++a equivalent to
a+1=a Post Decrement: a--
equivalent to a-1=a

Pre-Increment:

Eg 1:

```

#include<stdio.h>

void main()
{
    int a=20;
    ++a;
    printf("%d",a);
}

```

Output:21

Description: Initial value of a is 20 after execution of ++a, the value of a will be 21.

Pre-Decrement:

Eg 1:

```

#include<stdio.h>

void main()
{
    int a=20;
    --a;
}

```


Output:19

Description: Initial value of a is 20 after execution of --a, the value of a will be 19.

```
printf("%d",a);
}
```

Consider the following (let $a=20$)


Post increment

b=a++  current value of a used, b=a //b=20 a
is incremented by 1 a=a+1
//a=21

Pre-increment

b=++a → a is incremented by 1, a=a+1 //a=21
 → incremented value of a used, b=a //b=21

Post Decrement (let a=10)

b=a--  current value of a used, a is decremented by 1 b=a //b=10
a=a-1 //a=9

Pre-decrement (let a=10)

```

b=--a      a is decremented by 1,    a=a-1 // a=9
           decremented value of a used, b=a // b=9

```

Conditional Operator (OR) Ternary Operator

The operator which operates on 3 operands are called Ternary operator or conditional operator.

Syntax: (exp1)? exp2: exp3

where exp1, exp2, and exp3 are expressions and

- exp1 is an expression evaluated to true or false
- if exp1 is evaluated to true exp2 is executed
- if exp1 is evaluated false exp3 is executed

For example, consider the following statements.

```
a=10;  
b=15;  
Max = (a>b)? a:b;
```

In this example, Max will be assigned the value of b i.e b=15. This can be achieved using the if... else statements as follows:

```
if (a>b)
x=a;
else
x=b;
```

Bitwise Operators

Bitwise operators are used to manipulate the bits of given data. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning	Precedence	Associativity
~	Bitwise Negate	1	R->L
<<	Bitwise Left Shift	2	L->R
>>	Bitwise Right Shift	2	L->R
&	Bitwise AND	3	L->R
^	Bitwise XOR (exclusive OR)	4	L->R
	Bitwise OR	5	L->R

1) Bitwise AND (&)

If the corresponding bit positions in both the operand are 1, then AND operation results in 1; otherwise 0.

operand 1	AND	operand 2	result
0	&	0	0
0	&	1	0
1	&	0	0
1	&	1	1

a=10 00001010

b=6 00000110 (a&b)

c=2 00000010

2)Bitwise OR (|)

If the corresponding bit positions in both the operands are 0, then OR operation **results** in 0; otherwise 1.

operand 1	OR	operand 2	Result
0		0	0
0		1	1
1		0	1
1		1	1

a=10 00001010

b=6 00000110 (a|b)

c=2 00001110

Bitwise XOR (^)

If the corresponding bit positions in both the operand are different, then XOR operation results in 1; otherwise the result is 0.

operand 1	XOR	operand 2	result
0	^	0	0
0	^	1	1
1	^	0	1
1	^	1	0

a=10 00001010

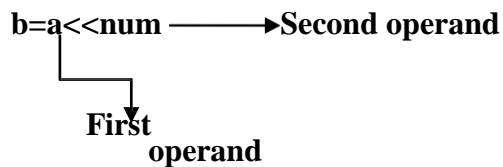
b=6 00000110 (a^b)

c=12 00001100

Left shift operator (<<)

The operator that is used to shift the data by a specified number of bit positions towards left is called left shift operator.

After left shift MSB should be discarded and LSB will be empty and should be filled with "0".



The first operand is the value to be shifted.

The second operand specifies the number of bits to be shifted.

The content of the first operand is shifted towards the left by the number bit positions specified in the second operand.

Example:

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

OUTPUT: 5<<1=10

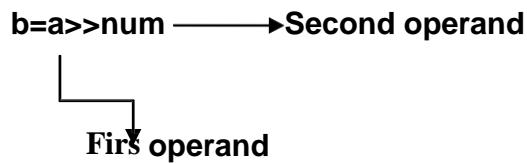
```
void main()
```

```
{   int a,b; a=5; b=a<<1;
    printf(" %d<<1=%d",a,b);
}
```

Right shift operator (>>)

The operator that is used to shift the data by a specified number of bit positions towards right is called right shift operator.

After right shift operation, LSB bit should be discarded and MSB is empty and should be filled with 0.



The first operand is the value to be shifted.

The second operand specifies the number of bits to be shifted.

The content of first operand are shifted towards right by the number bit positions specified in second operand.

Example:

```
#include<stdio.h>

void main()
{
    int a,b;
    a=10;
    b=a>>1
    ;
    printf(" %d>>1=%d",a,b);
}
```

OUTPUT: 10>>1=5

Bitwise negate:

The operator that is used to change every bit from 0 to 1 and 1 to 0 in the specified operand is called bitwise negate operator. It is used to find one's compliment of an operand.

Example:

```
#include<stdio.h>
void main()
{int a=10,b; b=~a;
    printf("Negation of a is %d",b);
}
```

Tracing:

a=10 0 0 0 0 1 0 1 0

b=245 1 1 1 1 0 1 0 1

Special Operators

C supports some special operators of interest such as comma operator, sizeof operator.

The Comma Operator

- ❖ The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated from left to right. The comma operator has least precedence among all operator.
- ❖ It is a binary operator which is used to:

1. Separate items in the list.

E.g. a=12,35,678;

2. Combine two or more statements into a single statement.

E.g. int a=10;
int b=20;
int c=30;

can be combined using comma operator as
follows: int a=10, b=20, c=30;

Example, the statement: value = (x=10, y=5, x+y)

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

The sizeof () Operator

The sizeof () is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, or a constant or a data type qualifier.

Syntax: sizeof(operand)

Examples:

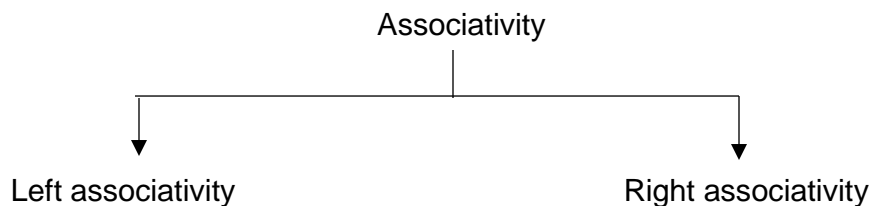
- | | | |
|----|---|---------------|
| 1. | <code>sizeof(char);</code> | Output:1 Byte |
| 2 | <code>int a;</code>
<code>sizeof(a)</code> | Output:2 Byte |

Precedence [Priority]

It is the rule that specifies the order in which certain operations need to be performed in an expression. For a given expression containing more than two operators, it determines which operations should be calculated first.

Associativity

If all the operators in an expression have equal priority then the direction or order chosen left to right or right to left to evaluate an expression is called associativity.



Left associativity:

In an expression if two or more operators having same priority are evaluated from left to right then the operators are called left to right associative operator, denoted as **L->R**

Right associativity:

In an expression if two or more operators having same priority are evaluated from right to left then the operators are called right to left associative operator, denoted as **R->L**

BODMAS Rule can be used to solve the expressions, where

B: Brackets O: Operators D: Division M: Multiplication A: Addition S: Subtraction

The Precedence (Hierarchy) of Operator

Operator Category	Operators in Order of Precedence (Highest to Lowest)	Associativity
Innermost brackets/Array elements	(), [], {}	Left to Right(L->R)

Reference		
Unary Operators	++, --, sizeof(), ~, +, -	Right to Left(R->L)
Member Access	-> or *	L->R
Arithmetic Operators	*, /, %	L->R
Arithmetic Operators	-, +	L->R
Shift Operators	<<, >>	L->R
Relational Operators	<, <=, >, >=	L->R
Equality Operators	==, !=	L->R
Bitwise AND	&	L->R
Bitwise XOR	^	L->R
Bitwise OR		L->R
Logical AND	&&	L->R
Logical OR		L->R
Conditional Operator	?:	R->L
Assignment Operator	=, +=, -=, *=, /=, %=	R->L
Comma Operator	,	L->R

Modes of Arithmetic Expressions

- An expression is a sequence of operands and operators that reduces to a single value. Expressions can be simpler or complex. An operator is a syntactical token that requires an action to be taken. An operand is an object on which an operation is performed.

- Arithmetic expressions are divided into 3 categories:

1. Integer Expressions
2. Floating Point Expressions
3. Mixed mode Expressions

1. Integer Expressions

- If all the operands in an expression are integers then the expression is called Integer Expression.
- The result of integer expression is always integer. Ex: $4/2=2$

2. Floating Point Expressions

- If all the operands in an expression are float numbers then the expression is called floating point Expression.
- The result of floating-point expression is always float Ex: $4.0/3.0=1.3333$

3. Mixed Mode Expressions

- An expression that has operands of different types is called Mixed Mode Expression. Ex: $4.0/2$ i.e float/int

Evaluation of Expressions involving all the operators

The rules to be followed while evaluating any expressions are shown below.

- Replace the variables if any by their values.
- Evaluate the expressions inside the parentheses
- Rewrite the expressions with increment or decrement operator as shown below:
 - a) Place the pre-increment or pre-decrement expressions before the expression being evaluated and replace each of them with corresponding variable.
 - b) Place the post-increment or post- decrement expressions after the expression being evaluated and replace each of them with

corresponding values.

- Evaluate each of the expression based on precedence and associativity.

1.20 Type Conversion

The process of converting the data from one data type to another data type is called Type conversion.

Type conversion is of two types

(i) Implicit type conversion

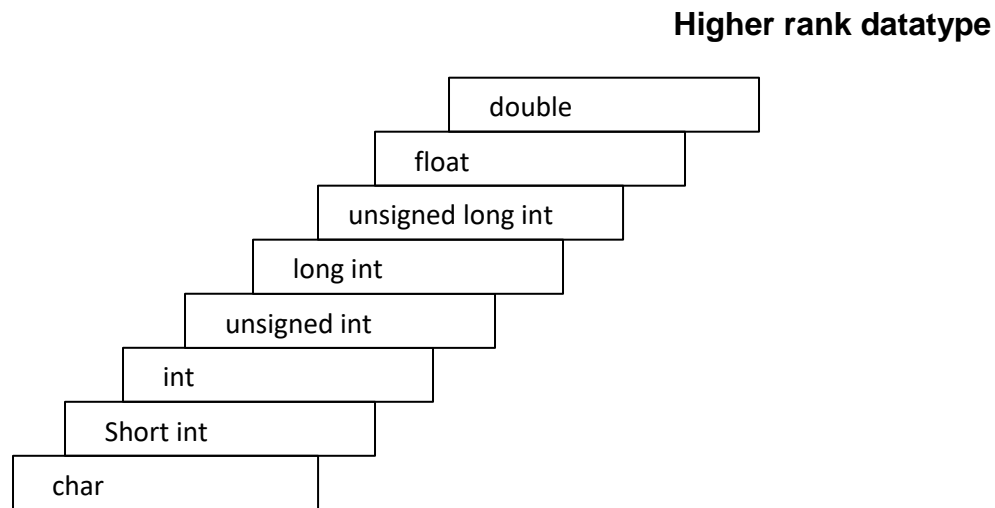
(ii) Explicit type conversion

i). Implicit Conversion

- C can evaluate the expression if and only if the data types of two operands are same.
- If operands are of different type, it is not possible to evaluate expression.
- In such situation to ensure that both operands are of same type, C compiler converts data from lower rank to higher rank automatically.
- This process of converting data from lower rank to higher rank is called as Implicit type conversion (ITC).

For example,

- If one operand type is same as other operand type, no conversion takes place and type of result remains same as the operands
i.e $\text{int} + \text{int} = \text{int}$, $\text{float} + \text{float} = \text{float}$ etc.
- If one operand type is int and other operand type is float, the operand with type int is promoted to float.

Promotion of hierarchy is as follows:**Lower rank datatype****Examples:**

- i. $5 + 3 = 8$
 $\text{int} + \text{int} = \text{int}$
- ii. $5 + 3.5$
 $= 8.5$
 $\text{int} + \text{float} = \text{float}$

ii) Explicit type conversion

- If the operands are of same datatype no conversion takes place by compiler. Sometimes the type conversion is required to get desired results.
- In such situation programmer himself has to convert the data from one type to another. This forcible conversion from one datatype to another is called as explicit type conversion (ETC).

Syntax:

Example: $(\text{int}) 9.93 = 9$

The data conversion from higher data type to lower data type is possible using explicit type conversion. The following example gives the explicit type conversion.

“What is type casting? Explain with example?”

Definition: C allows programmers to perform typecasting by placing the type name in parentheses and placing this in front of the value.

<code>x=(int) 7.5</code>	7.5 is converted to integer by truncation
<code>a=(int)21.3 / (int) 4.5</code>	Evaluated as 21/4 and the result would be 5.
<code>b=(double) sum/n</code>	Division is done in floating point mode
<code>y=(int)(a+b)</code>	The result $a + b$ is converted to integer
<code>z=(int)a +b</code>	a is converted to integer and then added to b

```
main()
{
float a;
a = (float)5 / 3;
}
```

Gives result as 1.666666. This is because the integer 5 is converted to floating point value before division and the operation between float and integer results in float.

Difference between ITC and ETC

ITC	ETC
<ol style="list-style-type: none"> 1. It is performed by Compiler. 2. ITC performs only lower to higher data conversion 3. ITC is performing when both the operands are of different type. 4. Example $2+3.4=5.4$ 	<ol style="list-style-type: none"> 1. It is performed by programmer. 2. ETC perform both lower to higher and higher to lower. 3. ETC is performed when both the operands are of same type. 4. Example float a=10.6 (int)a; Ans:10

1.21 Library functions in C

Library functions in C are also inbuilt functions in C language. These inbuilt functions are located in some common location, and it is known as the library. All the functions are used to execute a particular operation. These library functions are generally preferred to obtain the predefined output.

C Standard library functions or simply C Library functions are inbuilt functions in C programming. The prototype and data definitions of these functions are present in their respective header files. To use these functions, we need to include the header file in our program. For example, If you want to use the printf() function, the header file <stdio.h> should be included.

```
#include <stdio.h>
int main()
{
printf("Catch me if you can.");
}
```

If you try to use printf() without including the stdio.h header file, you will get an error.

Example: Square root using sqrt() function

Suppose, you want to find the square root of a number. To compute the square root of a number, you can use the sqrt() library function. The function is defined in the math.h header file.

```
#include <stdio.h>
#include <math.h>
int main()
{
    float num, root;
    printf("Enter a number: ");
    scanf("%f", &num);
    // Computes the square root of num and stores in root.
    root = sqrt(num);
    printf("Square root of %.2f = %.2f", num, root);
    return 0;
```

//program to illustrate Pow() function

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

// Driver code
int main()
{
    double base, power, result;
    base = 10.0;
    power = 2.0;

    // Calculate the result
    result = pow(base, power);
    printf("%.1lf^%.1lf = %.2lf",
           base, power, result);
    return 0;
}
```

Output

10.0^2.0 = 100.00

// program to illustrate log() functions

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

// Driver code
int main()
{
```



```

double num = 5.6, result;
result = log(num);
printf("log(%.1f) = %.2f",
      num, result);
return 0;
}

```

Output

log(5.6) = 1.72

S No.	Header Files	Description
1	<assert.h>	It checks the value of an expression that we expect to be true under normal circumstances. If the expression is a nonzero value, the assert macro does nothing.
2	<complex.h>	A set of functions for manipulating complex numbers.
3	<float.h>	Defines macro constants specifying the implementation-specific properties of the floating-point library.
4	<limits.h>	These limits specify that a variable cannot store any value beyond these limits, for example- An unsigned character can store up to a maximum value of 255.
5	<math.h>	The math.h header defines various mathematical functions and one macro. All the Functions in this library take double as an argument and return double as the result.
6	<stdio.h>	The stdio.h header defines three variable types, several macros, and various function for performing input and output.
7	<time.h>	Defines date and time handling functions.
8	<string.h>	Strings are defined as an array of characters. The difference between a character array and a string is that a string is terminated with a special character '\0'.

Course: Problem Solving Techniques using C

Course Code: 24BTPHY104

Module 2

2. MANAGING INPUT AND OUTPUT OPERATIONS

Reading, processing and writing of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data. We have two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements like `x=5`, `a=0` and so on. Another method is to use the input function `scanf`, which can read data from a keyboard. We have used both the methods in programs. For outputting results, we have used extensively the function `printf`, which sends results out to a terminal.

Input – Output functions:

The program takes some I/P- data, process it and gives the O/P.

We have two methods for providing data to the program

- i) Assigning the data to the variables in a program.
- ii) By using I/P-O/P statements.

C language has 2 types of I/O statements; all these operations are carried out through function calls.

- 1. Unformatted I/O statements
- 2. Formatted I/O statements

2.1 Unformatted I/O statements

getchar():- It reads single character from standard input device. This function donot require any arguments.

Syntax: - `char variable_name = getchar();`

Ex: - `char x;`
`x = getchar();`

putchar ():- This function is used to display one character at a time on the standard output

device.

Syntax:- putchar(char_variable);

Ex:- char x;

putchar(x);

program:

```
main( )
{
char ch;
printf(—enter a char||);
ch = getchar( );
printf(—entered char
is||); putchar(ch);
}
```

Output:

enter a char a

entered char is a

getc() :- This function is used to accept single character from the file.

Syntax: char variable_name = getc();

Ex:- char c;

c = getc();

putc():- This function is used to display single character.

Syntax :- putc(char variable_name);

Ex:- char c;

putc(c);

These functions are used in file processing.

gets():- This function is used to read group of characters(string) from the standard I/P device.

Syntax:- gets(character array variable);

Ex:- gets(s);

puts():- This function is used to display string to the standard O/P device.

Syntax:- puts(character array variables);

Ex:- puts(s);

program:

```
main()
{
char s[10];
puts(—enter namell);
gets(s);
puts(—print namell);
puts(s);
}
```

Output:

```
enter name ramu
print name ramu
```

getch(): - This function reads a single character directly from the keyboard without displaying on the screen. This function is used at the end of the program for displaying the output (without pressing (Alt-F5).

Syntax: char variable_name = getch();

Ex: - char c

c = getch();

getche(): - This function reads a single character from the keyboard and echoes(displays) it to the current text window.

Syntax:- char variable_name = getche();

Ex:- char c

c = getche();

program:

```
main()
{
char ch, c;
printf(—enter
charll); ch = getch();
printf(—%c\\, ch);
printf(—enter
charll); c = getche();
printf(—%c\\,c);
```

```
}
```

Output: -

enter character a

enter character b

b

Character test functions: -

Function Test

isalnum(c) -Is c an alphanumeric character?

isalpha(c) -Is c an alphabetic character

isdigit(c)- Is c a digit?

islower(c) -Is c a lower-case letter?

isprint(c) -Is c a character?

ispunct(c) -Is c a punctuation mark?

isspace(c) -Is c a white space character?

isupper(c)- Is c an upper-case letter?

tolower(c) -Convert ch to lower case

toupper(c) -Convert ch to upper case

program:

```
main()
{
    char a;
    printf("—enter\n");
    a = getchar();
    if (isupper(a))
    {
        x= tolower(a);
        putchar(x);
    }
    else
        putchar(toupper(a));
}
```

Output: - enter char A

a

2.2 Formatted I/O Functions

Formatted I/O refers to input and output that has been arranged in a particular format.

Formatted I/P functions**scanf() , fscanf()**

Formatted O/P functions---

printf() , fprintf()

scanf() : -scanf() function is used to read information from the standard I/P device.

Syntax: - scanf(—controlstringll, &variable_name);

Control string (also known as format string) represents the type of data that the user is going to accept and gives the address of variable. (char-%c , int-%d , float - %f , double-%lf).Control string and variables are separated by commas. Control string and the variables going to I/P should match with each other.

Ex: - int n;
scanf(—%dll,&n);

Inputting Integer Numbers:

The field specification for reading an integer number is:
%w d

The percentage sign (%) indicates that a conversion specification follows. w is an integer number that specifies the field width of the number to be read and d known as data type character indicates that the number to be read is in integer mode.

Ex: - scanf(—%2d,%5dll,&a&b);

The following data are entered at the console:

50 31426

Here the value 50 is assigned to a and 31426 to b

Inputting Real Numbers:

The field width of real numbers is not to be specified unlike integers. Therefore, scanf reads real numbers using the simple specification %f.

Ex: `scanf(—%f %fll,&x,&y);`

Suppose the following data are entered as input:

23.45 34.5

The value 23.45 is assigned to x and 34.5 is assigned to y.

Inputting character strings:

The field specification for reading character strings is
%ws or %wc

%c may be used to read a single character.

Ex: `scanf(—%sll,name1);`

Suppose the following data is entered as input:

Griet

Griet is assigned to name1.

Formatted Output:

printf(): This function is used to output any combination of data. The outputs are produced in such a way that they are understandable and are in an easy to use form. It is necessary for the programmer to give clarity of the output produced by his program.

Syntax:- `printf(—control stringll, var1,var2.....);`

Control string consists of 3 types of items

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the O/P format for display of each item.
3. Escape sequence chars such as \n, \t and \b.....

The control string indicates how many arguments follow and what their types are.

The var1, var2 etc... are variables whose values are formatted and printed according to the specifications of the control string.

The arguments should match in number, order and type with the format specifications.

O/P of integer number: - Format specification : %wd

Format

O/P

printf(—%dl, 9876)

9	8	7	6
---	---	---	---

printf(—%6dl,9876)

		9	8	7	6
--	--	---	---	---	---

printf(—%2dl,9876)

9	8
---	---

printf(—%-6dl,9876)

9	8	7	6		
---	---	---	---	--	--

printf(—%06dl,9876)

0	0	9	8	7	6
---	---	---	---	---	---

O/P of real number: %w.pf

w---- Indicates minimum number of positions that are to be used for display of the value.

p----- Indicates number of digits to be displayed after the decimal point.

Format **y=98.7654**

O/P

printf(—%7.4fl,y)

9	8	.	7	6	5	4
---	---	---	---	---	---	---

printf(—%7.2fl,y)

		9	8	.	7	7
--	--	---	---	---	---	---

printf(—%-7.2fl,y)

9	8	.	7	7		
---	---	---	---	---	--	--

printf(—%10.2el,y)

		9	.	8	8	e	+	0	1
--	--	---	---	---	---	---	---	---	---

O/P of single characters and string:

%wc

%ws

Format

O/P

%s

R	a	J	U		R	a	j	e	s	h		R	a	N	i
---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---

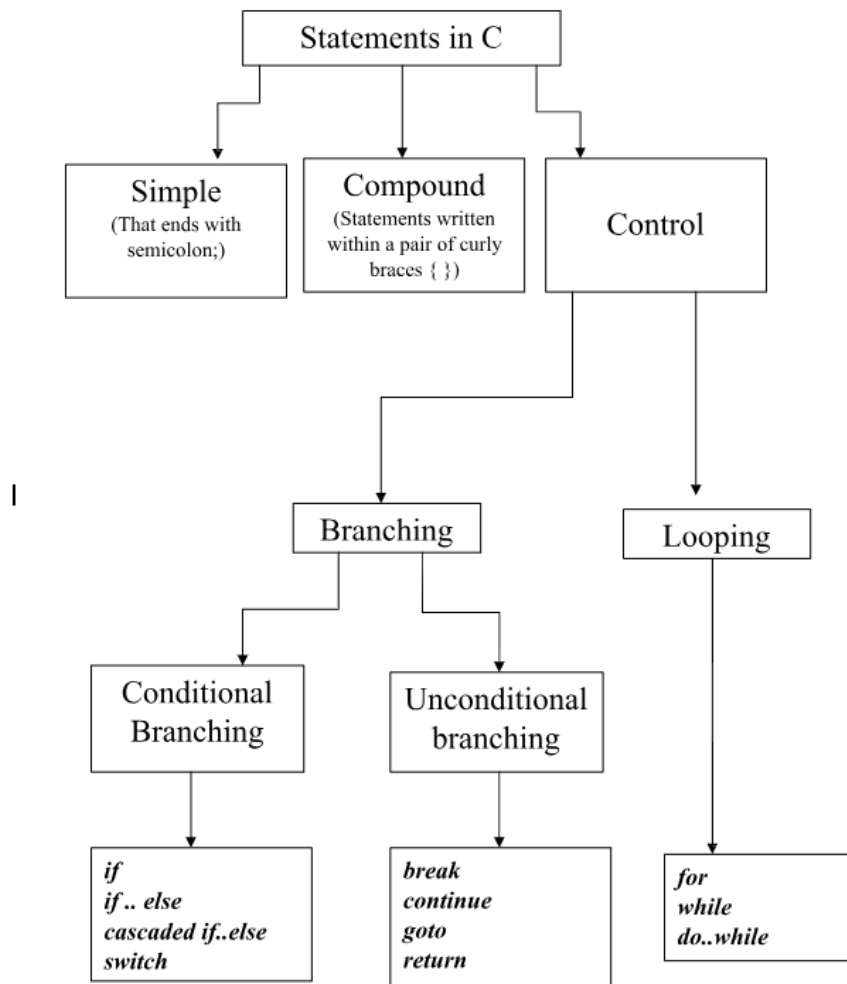
%18s

	R	a	J	U		R	a	J	E	s	h		R	a	n	i
--	---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---

%18s

R	A	j	U		R	a	j	E	S	h		R	a	n	i	
---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	---	--

Decision making, Branching and Looping



2.3 Decision Making Statements

If and Switch Statements

We have a number of situations where we may have to change the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

2.3.1 The if Statement

This is basically a “one-way” decision statement. This is used when we have only one

alternative.

The syntax is shown below:

```

if(expression)
{
    statement1;
}

```

Firstly, the expression is evaluated to *true* or *false*.

If the expression is evaluated to *true*, then statement1 is executed. If the expression is evaluated to *false*, then statement1 is skipped. The flow diagram is shown below:

Example 1: Program to illustrate the use of if statement

```

#include<stdio.h>
main()
{
    int a,b;
    printf("Enter two numbers");
    scanf("%d%d",&a,&b);
    if a>b
        printf(" a is greater");
    if b>a
        printf("b is greater");
}

```

Example 2: Program to illustrate the use of if statement.

```

#include<stdio.h>void
main()
{
    int n;
    printf("Enter any non-zero integer: \n");
    scanf("%d", &n)
    if(n>0)
        printf("Number is positive number");if(n<0)
        printf("Number is negative number");
}

```

Output:

```

Enter any non-zero integer:7
Number is positive number

```

2.3.2 The if else Statement

This is basically a “two-way” decision statement.

This is used when we must choose between two alternatives. The syntax is shown below:

```

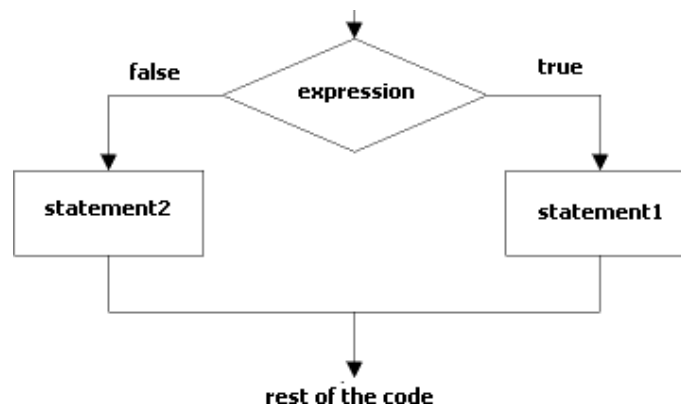
if(expression)
{
    statement1;
}
else
{
    statement2;
}

```

Firstly, the expression is evaluated to true or false.

If the expression is evaluated to *true*, then statement1 is executed. If the expression is evaluated to *false*, then statement2 is executed.

The flow diagram is shown below:



Example: Program to illustrate the use of if else statement.

```

#include<stdio.h>
void main()
{
    int n;
    printf("Enter any non-zero integer: \n");
    scanf("%d", &n)
    if(n>0)
        printf("Number is positive number");
    else

```

```
}
```

```
printf("Num  
ber is  
negative  
number")
```

Output:

Enter any non-zero integer:7
Number is positive number

Example 2: Program to illustrate if else statement to find greatest of two numbers.

```
#include<stdio.h>
main()
{
    int a,b;
    printf("Enter two numbers");
    scanf("%d%d",&a,&b);
    if a>b
        printf("a is greater")
    else
        printf("b is greater");
}
```

Example 3: program to check whether an integer is odd or even

```
#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
        printf("%d is an odd integer.",number);
    }

    return 0;
}
```

Output:

Enter an integer: 7
7 is an odd integer.

Example 4: Program to check vowel or consonant

```
#include <stdio.h>
int main() {
    char c;
    int lowercase_vowel, uppercase_vowel;
    printf("Enter an alphabet: ");
    scanf("%c", &c);

    // evaluates to 1 if variable c is a lowercase vowel
    lowercase_vowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');

    // evaluates to 1 if variable c is a uppercase vowel
    uppercase_vowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    // evaluates to 1 (true) if c is a vowel
    if (lowercase_vowel || uppercase_vowel)
        printf("%c is a vowel.", c);
    else
        printf("%c is a consonant.", c);
    return 0;
}
```

OUTPUT:

Enter an alphabet: G
G is a consonant.

Example 5: Program to Find the Largest Number Among Three Numbers

```
#include <stdio.h>

int main() {

    double n1, n2, n3;

    printf("Enter three different numbers: ");
    scanf("%lf %lf %lf", &n1, &n2, &n3);

    // if n1 is greater than both n2 and n3, n1 is the largest
    if (n1 >= n2 && n1 >= n3)
        printf("%.2f is the largest number.", n1);

    // if n2 is greater than both n1 and n3, n2 is the largest
    if (n2 >= n1 && n2 >= n3)
        printf("%.2f is the largest number.", n2);

    // if n3 is greater than both n1 and n2, n3 is the largest
    if (n3 >= n1 && n3 >= n2)
        printf("%.2f is the largest number.", n3);
}
```

```
    return 0;
}
```

OUTPUT:

Enter three different numbers: 4 2 7
7.00 is the largest number.

Example 6: Program to check whether the number is positive or negative

```
#include <stdio.h>

int main() {

    double num;
    printf("Enter a number: ");
    scanf("%lf", &num);
    if (num <= 0.0) {
        if (num == 0.0)
            printf("You entered 0.");
        else
            printf("You entered a negative number.");
    }
    else
        printf("You entered a positive number.");

    return 0;
}
```

Output:

Enter a number: 4
You entered a positive number.

Example 7: Program to find the factorial of n

```
#include <stdio.h>
int main() {
    int n, i;
    unsigned long long fact = 1;
    printf("Enter an integer: ");
    scanf("%d", &n);

    // shows error if the user enters a negative integer
    if (n < 0)
        printf("Error! Factorial of a negative number doesn't exist.");
    else {
        for (i = 1; i <= n; ++i) {
            fact *= i;
        }
    }
}
```

```

    printf("Factorial of %d = %llu", n, fact);
}

return 0;
}

```

OUTPUT:

Enter an integer: 10

Factorial of 10 = 3628800

2.3.3 The nested if Statement

An if-else statement within another if-else statement is called nested if statement.

This is used when an action has to be performed based on many decisions. Hence, it is called as multi-way decision statement.

The syntax is shown below:

```

if(expr1)
{
    if(expr2)
        statement1
    else
        statement2
}
else
{
    if(expr3)
        statement3
    else
        statement4
}

```

- Here, firstly expr1 is evaluated to true or false.

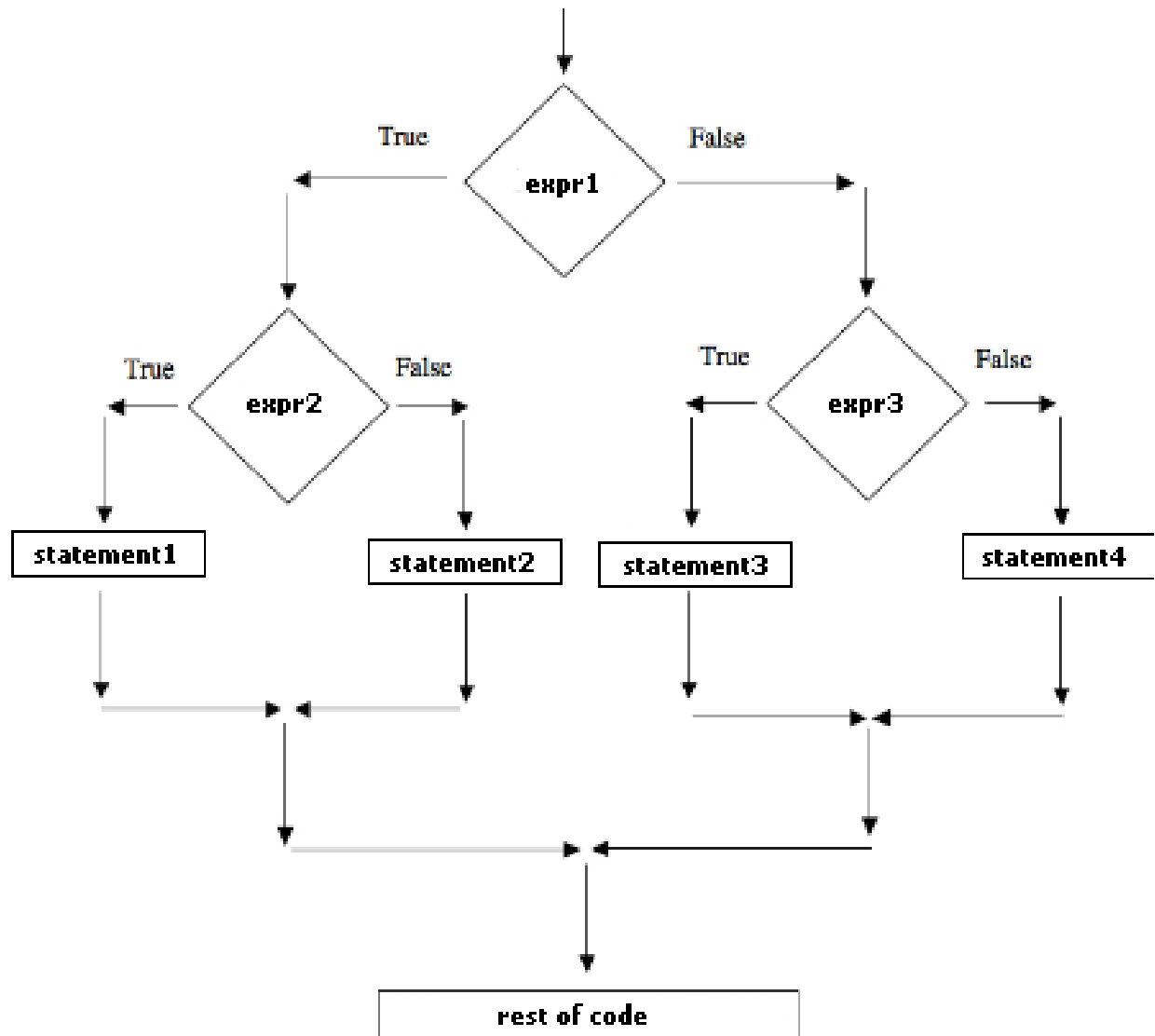
If the expr1 is evaluated to *true*, then expr2 is evaluated to true or false. If the expr2 is evaluated to *true*, then statement1 is executed.

If the expr2 is evaluated to *false*, then statement2 is executed.

If the expr1 is evaluated to *false*, then expr3 is evaluated to true or false. If the expr3 is evaluated to *true*, then statement3 is executed.

If the expr3 is evaluated to *false*, then statement4 is executed.

The flow diagram is shown below:



Example1: Program to select and print the largest of the 3 numbers using nested “if-else” statements.

```

#include<stdio.h>void main()
{
int a,b,c;
printf("Enter Three Values: \n");
scanf("%d %d %d ", &a, &b, &c);
printf("Largest Value is: ") ;
    if(a>b)
    {
        if(a>c)
            printf(" %d ", a);
        else
            printf(" %d ", c);
    }
    else
    {
        if(b>c)
            printf(" %d", b);
        else
            printf(
                " %d",
                c)
    }
}

```

Example 2:

Problem Statement: Determine the grade of a student based on their score.

- If the score is above 90, it's an 'A'.
 - If the score is between 80 and 90, check if it's above 85 for an 'A-' or else 'B+'.
- Below 80, if it's above 70, it's a 'C', otherwise, it's a 'D'.

```
#include <stdio.h>
int main() {
    int score = 88;
    // Check if score is greater than 90
    if (score > 90) {
        printf("Grade: A\n");
    } else {
        // Score is 90 or below, check if score is greater than 80
        if (score > 80) {
            // Score is between 81 and 90, check if score is above 85
            if (score > 85) {
                printf("Grade: A-\n");
            } else {
                printf("Grade: B+\n");
            }
        } else {
            // Score is 80 or below, check if score is above 70
            if (score > 70) {
                printf("Grade: C\n");
            } else {
                printf("Grade: D\n");
            }
        }
    }
    return 0;
}
```

2.3.4 The Cascaded if-else statement (The else if ladder statement)

This is basically a “multi-way” decision statement.

This is used when we must choose among many alternatives. The syntax is shown below:

```
if(expression1)
{
    statement1;
```

```

    }
    else if(expression2)
    {
        statement2;
    }
    else if(expression3)
    {
        statement3
    }

    else
    {

```

Default statement

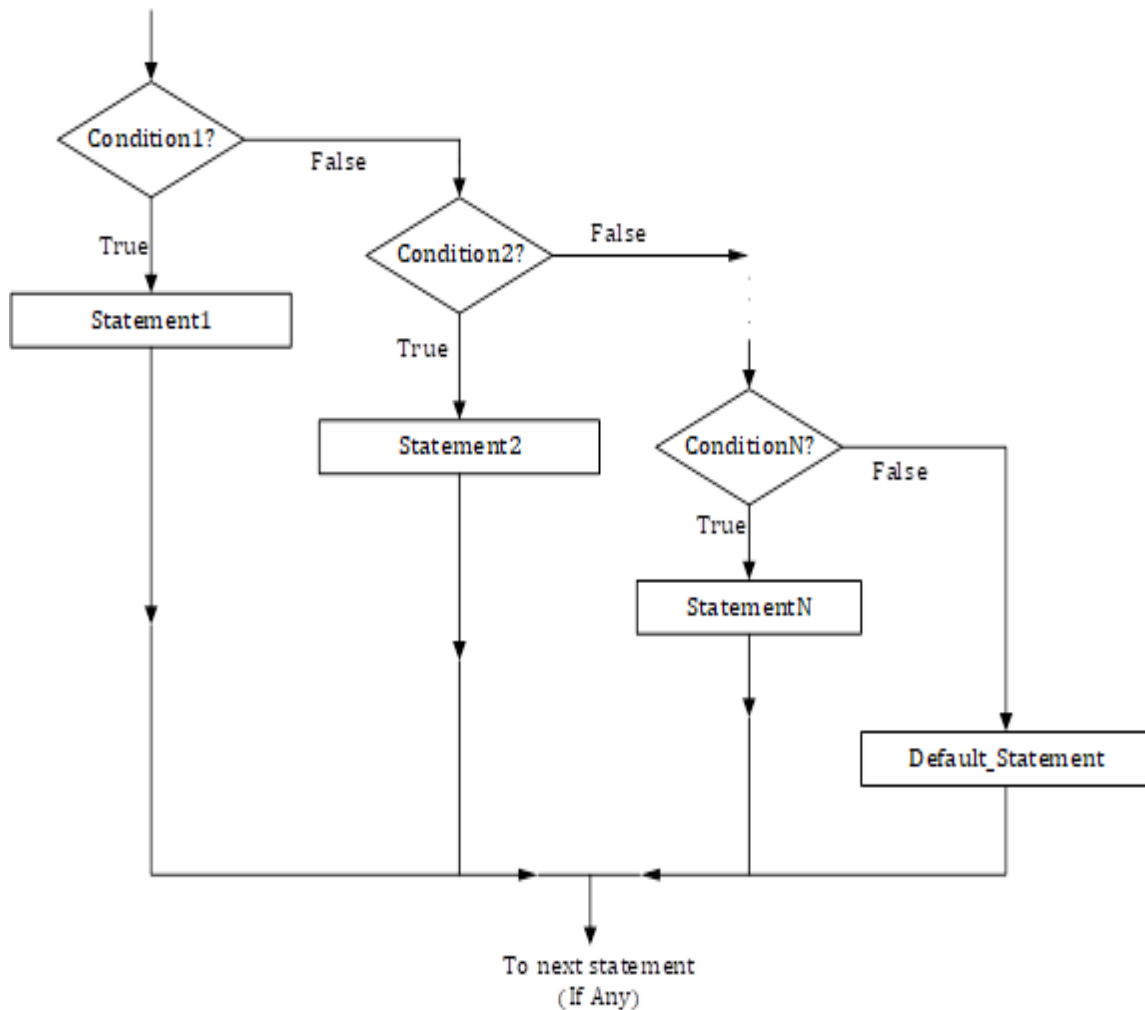
The expressions are evaluated in order (i.e. top to bottom). If an expression is evaluated to true, then

- Statement associated with the expression is executed &
- Control comes out of the entire else if ladder

For ex, if expression1 is evaluated to *true*, then statement1 is executed.

If all the expressions are evaluated to false, the last statement4 (default case) is executed.

Flowchart of if-else-if Ladder statement



Example: Program to find largest from three numbers given by user to explain working of if-else-if statement or ladder.

```

#include<stdio.h>
int main()
{
    int a,b,c;
    printf("Enter three numbers: \n");
    scanf("%d%d%d", &a, &b, &c);
    if(a>b && a>c)
    {
        printf("Largest = %d", a);
    }
    else if(b>a && b>c)
    {
        printf("Largest = %d", b);
    }
}
  
```

```
    else
    {
        printf("Largest = %d", c);
    }
    return(0);
}
```

Output

Enter three numbers:

12

33

17

Largest = 3

2.3.5 The Switch Statement

This is a multi-branch statement similar to the if - else ladder (with limitations) but clearer and easier to code.

Syntax:

```

switch (expression)
{
    case constant1:    statement1;
                      break;

    case constant2:    statement2;
                      break;

    ...

    default:          statement;
}

```

The value of expression is tested for equality against the values of each of the constants specified in the **case** statements in the order written until a match is found. The statements associated with that case statement are then executed until a break statement or the end of the switch statement is encountered.

When a break statement is encountered execution jumps to the statement immediately following the switch statement.

The default section is optional -- if it is not included the default is that nothing happens and execution simply falls through the end of the switch statement.

The switch statement however is limited by the following

- Can only test for equality with **integer constants** in case statements.
- No two case statement constants may be the same.
- Character constants are automatically converted to integer.

Rules for switch statement in C language

- The *switch expression* must be of an integer or character type.
- The *case value* must be an integer or character constant.
- The *case value* can be used only inside the switch statement.
- The *break statement* in switch case is not must. It is optional.
- If there is no break statement found in the case, all the cases will be executed present after the matched case.
- It is known as *fall through* the state of C switch statement.
- We are assuming that there are following variables.

int x,y,z;

char a,b;

float f;

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Example: - Program to simulate a basic calculator.

```

#include <stdio.h>
void
main()
{
    double num1, num2, result ;char
    op ;

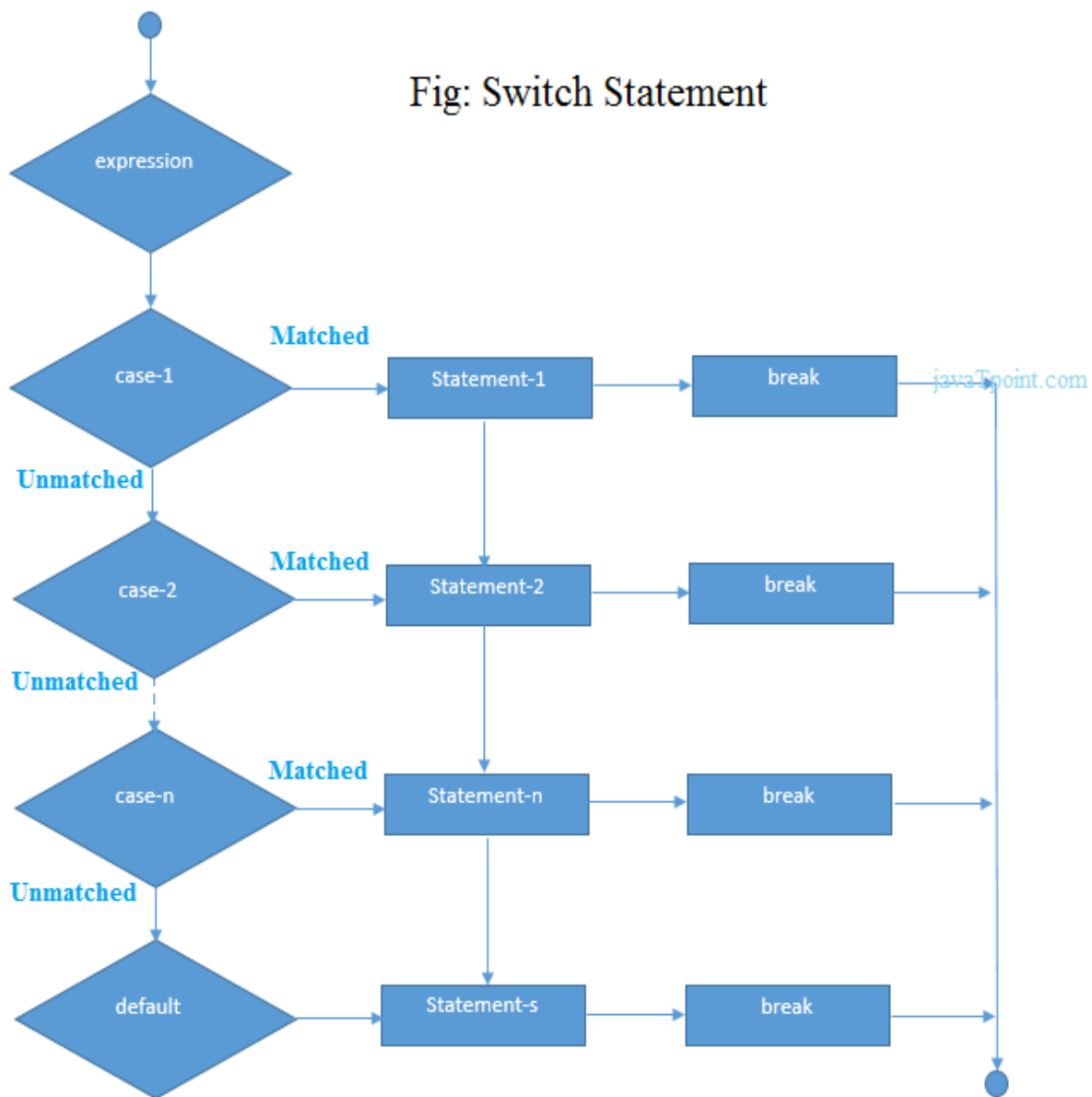
    printf ( " Enter number operator number\n" ) ;scanf ("%f %c
    %f", &num1, &op, &num2 ) ; switch ( op )
    {
        case '+' : result = num1 + num2 ;
                    break ;
        case '-' : result = num1 - num2 ;
                    break ;
        case '*' : result = num1 * num2 ;
                    break ;
        case '/' : if ( num2 != 0.0 ) {
                    result = num1 / num2 ;break ;
                    }
        // else we allow to fall through for error message

        default : printf ("ERROR -- Invalid operation or division
                    by 0.0" ) ;
    }
    printf( "%f %c %f = %f\n", num1, op, num2, result) ;
}

```

Note : The break statement need not be included at the end of the case statement body if it is logically correct for execution to fall through to the next case statement (as in the case of division by 0.0) or to the end of the switch statement (as in the case of default :).

Fig: Switch Statement



/* program to implement switch */

```

#include<stdio.h>
main()
{
int marks,index;
char grade[10];
printf("Enter your marks");
scanf("%d",&marks);

```

```

index=marks/10;
switch(index)
{
case 10 :
case 9:
case 8:
case 7:
case 6: grade="first";
break;
case 5 : grade="second";
break;
case 4 : grade="third";
break;
default : grade ="fail";
break;
}
printf("%s",grade);
}

```

2.4 Ternary operator

This is a special shorthand operator in C and replaces the following segment.

```

if (condition)
expr_1 ;
else
expr_2 ;

```

with the more elegant:

Syntax: *condition? expr_1: expr_2;*

The? operator is a ternary operator in that it requires three arguments. One of the advantages of the? operator is that it reduces simple conditions to one simple line of code which can be thrown unobtrusively into a larger section of code.

For Example: - to get the maximum of two integers, x and y, storing the larger in max.

```
max = x >= y ? x : y ;
```

The alternative to this could be as follows

```

if ( x >= y )
    max = x ;
else
    max = y ;

```

giving the same result but the former is a little bit more sufficient.

2.5 Looping structure in C

2.5.1 For statement

The for statement is most often used in situations where the programmer knows in advance how many times a particular set of statements are to be repeated. The for statement is sometimes termed a counted loop.

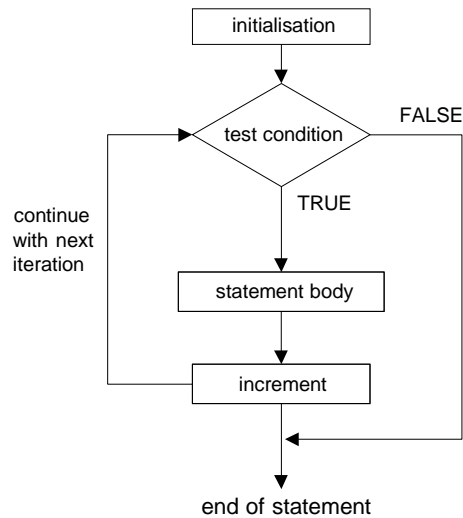
Syntax: **for ([initialisation] ; [condition] ; [increment])**
 {
 [statement body];
 }

Initialisation :- this is usually an assignment to set a loop counter variable for example.

Condition :- determines when loop will terminate.

Increment :- defines how the loop control variable will change each time the loop is executed.

Statement body :- can be a single statement, no statement or a block of statements. The *for* statement executes as follows :-



Note: The square braces above are to denote optional sections in the syntax but are not part of the syntax. The semi-colons must be present in the syntax.

For Example: Program to print out all numbers from 1 to 100.

```

#include <stdio.h>void
main()
{
    int x ;

    for ( x = 1; x <= 100; x++ )printf( "%d\n", x ) ;
}
    
```

Curly braces are used in C to denote code blocks whether in a function as in main() or as the body of a loop.

For Example: - To print out all numbers from 1 to 100 and calculate their sum.

```

#include <stdio.h>void
main()
{
    int x, sum = 0 ;

    for ( x = 1; x <= 100; x++ )
    {
        printf( "%d\n", x ) ;sum +=
        x ;
    }
    printf( "\n\nSum is %d\n", sum ) ;
}
    
```

Multiple Initialization

C has a special operator called the **comma operator** which allows separate expressions to be tied together into one statement.

For example, it may be tidier to initialize two variables in a for loop as follows: -

```
for (x = 0, sum = 0; x <= 100; x++)
{
    printf("%d\n", x) ;sum += x
;
}
```

Any of the four sections associated with a for loop may be omitted but the semi-colons must be present always.

For Example: -

```
for ( x = 0; x < 10;          )
    printf( "%d\n", x++ ) ;
...
x = 0 ;
for ( ; x < 10; x++ ) printf( "%d\n",
    x ) ;
```

An infinite loop may be created as follows

```
for ( ; ; )
    statement body ;
```

or indeed by having a faulty terminating condition.

Sometimes for statement may not even have a body to execute as in the following example where we just want to create a time delay.

```
for ( t = 0; t < big_num ; t++ ) ;
```

or we could rewrite the example given above as follows

```
for ( x = 1; x <= 100; printf( "%d\n", x++ ) ) ;
```

The initialization, condition and increment sections of the for statement can contain any valid C expressions.

```
for ( x = 12 * 4 ; x < 34 / 2 * 47 ; x += 10 )printf(
"%d ", x ) ;
```

It is possible to build a nested structure of for loops, for example the following creates a large time delay using just integer variables.

```
unsigned int x, y;

for (x = 0; x < 65535; x++) for (y = 0; y < 65535;
    y++);
```

Example :

Program to produce the following table of values

```
#include <stdio.h> void main()
{
    int j, k ;

    for ( j = 1; j <= 5; j++ )
    {
        for ( k = j ; k < j + 5; k++ )
        {
            printf( "%d  ", k ) ;
        }
        printf( "\n" ) ;
    }
}
```

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

C Program to Display Fibonacci Sequence

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

    int i, n;

    // initialize first and second terms
    int t1 = 0, t2 = 1;

    // initialize the next term (3rd term)
    int nextTerm = t1 + t2;
    // get no. of terms from user
    printf("Enter the number of terms: ");
    scanf("%d", &n);
```

```

// print the first two terms t1 and t2
printf("Fibonacci Series: %d, %d, ", t1, t2);

// print 3rd to nth terms
for (i = 3; i <= n; ++i) {
    printf("%d, ", nextTerm);
    t1 = t2;
    t2 = nextTerm;
    nextTerm = t1 + t2;
}

return 0;
}

```

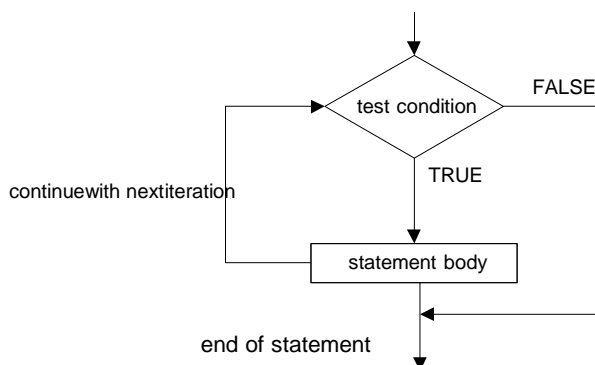
Output

Enter the number of terms: 10
 Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

2.5.2 While statement

The while statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax: **while (condition)**
 {
 statement body;
 }



Example 1: C program to print numbers from 1 to 5 using while loop

```

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

```



```

while (i <= 5)
{
    printf("%d\n", i);
    ++i;
}
return 0;
}

```

Output:

```

1
2
3
4
5

```

Example 2: Program to sum all integers from 100 down to 1.

```

#include <stdio.h>
void main()
{
    int sum = 0, i = 100 ;

    while ( i )
        sum += i-- ; // note the use of postfix decrement operator!
    printf ("Sum is %d \n", sum);
}

```

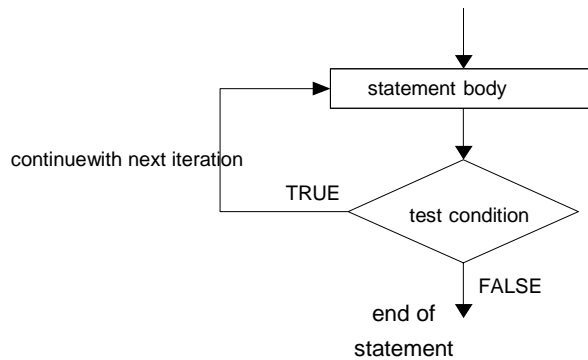
where it should be recalled that any non-zero value is deemed TRUE in the condition section of the statement.

2.5.3 Do while

The terminating condition in the “for” and “while” loops are always tested before the body of the loop is executed -- so of course the body of the loop may not be executed at all.

In the **do while** statement on the other hand the statement body is always executed at least once as the condition is tested at the end of the body of the loop.

Syntax: **do**
 {
 statement body;
 } while (condition);



Example: To read in a number from the keyboard until a value in the range 1 to 10 is entered.

```

int i ;do
{
    scanf( "%d\n", &i ) ;
} while ( i < 1 && i > 10 ) ;
  
```

In this case we know at least one number is required to be read so the do-while might be the natural choice over a normal while loop.

2.6 Break statement

When a break statement is encountered inside a while, for, do/while or switch statement the statement is immediately terminated, and execution resumes at the next statement following the statement.

For Example: -

```

...
for ( x = 1 ; x <= 10 ; x++ )
{
    if ( x > 4 )
        break;

    printf( "%d ", x ) ;
}
printf( "Next executed\n" ); //Output : "1          2    3    4    NextExecuted"
  
```

2.7 Continue statement

The continue statement terminates the current iteration of a while, for or do/while statement and resumes execution back at the beginning of the loop body with the next iteration.

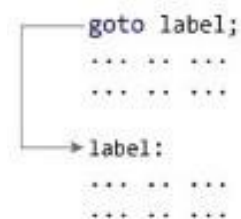
For Example: -

```
for ( x = 1; x <= 5; x++ )
{
    if ( x == 3 )
        continue ; printf( "%d ", x ) ;
}
printf( "Finished Loop\n" ) ; // Output : "1 2 4 5 Finished Loop"
```

2.8 Goto statement

goto statement can be used to branch unconditionally from one point to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by a colon (:). The label is placed immediately before the statement where the control is to be transferred.

The syntax is shown below:



Example: Program to detect the entered number as to whether it is even or odd. Use goto statement.

```
#include<stdio.h>void main()
{
    int x;
    printf("Enter a Number: \n"); scanf("%d", &x);
    if(x % 2 == 0)
        goto even;
    else
        goto odd;
even:printf("%d is Even Number");return;
odd:printf(" %d is Odd Number");
}
```

Output:

```
Enter a Number:5
5 is Odd Number.
```

2.9 C Programs

2.9.1 C Program to Print Pyramids and Patterns

Example 1: Half Pyramid of *

```
*
* *
* * *
* * * *
* * * * *
```

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i) {
        for (j = 1; j <= i; ++j) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Example 2: Half Pyramid of Numbers

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i) {
        for (j = 1; j <= i; ++j) {
            printf("%d ", j);
        }
        printf("\n");
    }
    return 0;
}
```

Example 3: Inverted half pyramid of *

```
* * * * *
* * * *
* * *
* *
*
```

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (i = rows; i >= 1; --i) {
        for (j = 1; j <= i; ++j) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

2.9.2 C Program to Check Whether a Number is Palindrome or Not

```
#include <stdio.h>
int main() {
    int n, reversed = 0, remainder, original;
    printf("Enter an integer: ");
    scanf("%d", &n);
    original = n;

    // reversed integer is stored in reversed variable
    while (n != 0) {
        remainder = n % 10;
        reversed = reversed * 10 + remainder;
        n /= 10;
    }

    // palindrome if original and reversed are equal
    if (original == reversed)
        printf("%d is a palindrome.", original);
    else
        printf("%d is not a palindrome.", original);

    return 0;
}
```

Output:

```
Enter an integer: 1001
1001 is a palindrome.
```

2.9.3 C Program to Calculate the Power of a Number

```
#include <stdio.h>
int main() {
    int base, exp;
    long double result = 1.0;
    printf("Enter a base number: ");
    scanf("%d", &base);
    printf("Enter an exponent: ");
    scanf("%d", &exp);

    while (exp != 0) {
        result *= base;
        --exp;
    }
    printf("Answer = %.0Lf", result);
    return 0;
}
```

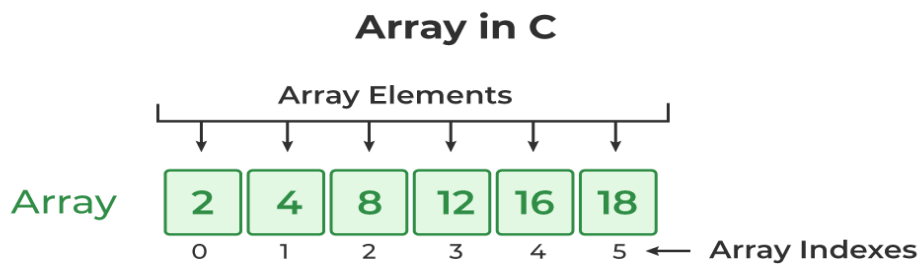
OUTPUT:

```
Enter a base number: 3
Enter an exponent: 4
Answer = 81
```

Course: Problem Solving Techniques using C**Course Code: 24BTPHY104****Module 3****3.ARRAYS – STRINGS – STORAGE CLASSES****3.1 Arrays**

An Array is a collection of similar data elements. Array is one of the most used data structures in C programming. It is a simple and fast way of storing multiple values under a single name.

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.



Pictorial representation of an array of 5 integers

10	20	30	40	50
A[0]	A[1]	A[2]	A[3]	A[4]

- ☐ An array is a collection of similar data items.
- ☐ All the elements of the array share a common name.
- ☐ Each element in the array can be accessed by the subscript (or index) and array name.
- ☐ The arrays are classified as:
 - 1.Single dimensional array
 - 2.Multidimensional array.

3.2 Single Dimensional Array.

- a. A single dimensional array is a linear list of related data items of same data type.
- b. In memory, all the data items are stored in contiguous memory locations.

Declaration of one-dimensional array (Single dimensional array)

Syntax:

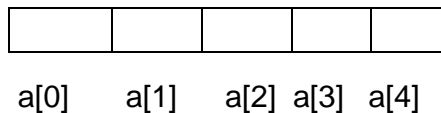
`datatype array_name[size];`

- ✓ **datatype** can be int,float,char,double.
- ✓ **array_name** is the name of the array and it should be a valid identifier.
- ✓ **Size** is the total number of elements in array.

For example:

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

The above statement allocates $5 \times 2 = 10$ Bytes of memory for the array **a**.



```
float b[5];
```

The above statement allocates $5 \times 4 = 20$ Bytes of memory for the array **b**.

- a. Each element in the array is identified using integer number called as **index**.
- b. If n is the size of array, the array index starts from **0** and ends at **n-1**.

Storing Values in Arrays

- i. Declaration of arrays only allocates memory space for array. But array elements are not initialized and hence values have to be stored.
- ii. Therefore, to store the values in array, there are 3 methods

1. Initialization
2. Assigning Values
3. Input values from keyboard through **scanf()**

Initialization of one-dimensional array

- Assigning the required values to an array element before processing is called initialization.

```
data type array_name[expression]={v1,v2,v3...,vn};
```

Where

- ✓ datatype can be char,int,float,double
 - ✓ array name is the valid identifier
 - ✓ size is the number of elements in array
 - ✓ v1, v2, v3.....vn are values to be assigned.
- Arrays can be initialized at declaration time. Example:
int a[5]={2,4,34,3,4};

2	4	34	3	4
a[0]	a[1]	a[2]	a[3]	

The various ways of initializing arrays are as follows:

Initializing all elements of array (Complete array initialization)

1. Partial array initialization.
2. Initialization without size.
3. String initialization

Initializing all elements of array:

- Arrays can be initialized at the time of declaration when their initial values are known in advance.
- In this type of array initialization, initialize all the elements of specified memory size.

➤ Example:

```
int a[5]={10,20,30,40,50};
```

10	20	30	40	50
----	----	----	----	----

Initialization without size

- In the declaration the array size will be set to the total number of initial values specified.
- The compiler will set the size based on the number of initial values.

Example:

```
int a[ ]={10,20,30,40,50};
```

In the above example the size of an array is set to 5

String Initialization

- Sequence of characters enclosed within double quotes is called as string.
- The string always ends with NULL character (**\0**)

```
char s[5]="SNPS";
```

We can observe that string length is 4, but size is 5 because to store NULL character we need one more location.

So pictorial representation of an array **s** is as follows

S	N	P	S	\0
S[0]	S[1]	S[2]	S[3]	S[4]

Assigning values to arrays

Using assignment operators, we can assign values to individual elements of arrays.

For example:

```
int a[3];
a[0]=10;
a[1]=20;
a[2]=30;
```

10	20	30
a[0]	a[1]	a[2]

Reading and writing single dimensional arrays.

To read array elements from keyboard we can use **scanf()** function as follows:

To read 0th element: `scanf("%d",&a[0]);` To read
 1st element: `scanf("%d",&a[1]);` To read 2nd
 element: `scanf("%d",&a[2]);`

.....

.....

To read nth element : `scanf("%d",&a[n-1]);`

In general

To read ith element:

`scanf("%d",&a[i]);` where `i=0; i<n; i++`

To print array elements we can use **printf()** function as follows:

To print 0th element: `printf("%d",a[0]);` To
 print 1st element: `printf("%d",a[1]);` To print
 2nd element :`printf("%d",a[2]);`

.....

.....

To nth element : `printf("%d",&a[n-1]);`

In general

To read ith element:

`printf("%d",a[i]);` where `i=0; i<n; i++`

- **Write a C program to read N elements from keyboard and to print N elements on screen.**

```
/* program to read N elements from keyboard and to print N elements on
screen */
#include<stdio.h>
void main()
{
    int i,n,a[10];
    printf("enter number of array elements\n");
    scanf("%d",&n);
    printf("enter array elements\n");
    for(i=0; i<n;i++)
```

```

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

printf("array elements are\n");
for(i=0; i<n;i++)
{
printf("%d",a[i]);
}
}

```

2. Write a C program to find sum of n array elements.

/* program to find the sum of n array elements. */

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

```
void main()
```

```

{
int i,n,a[10],sum=0;
printf("enter number of array elements\n");
scanf("%d",&n);
printf("enter array elements\n");
for(i=0; i<n; i++)
{
scanf("%d",&a[i]);
}

for(i=0; i<n;i++)
{
sum=sum+ a[i];

}

printf("sum is %d\n",sum):

}

```

3. Write a c program to find largest of n elements stored in an array a.

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

```
void main()
```

```

{
int i,n,a[10],big;
printf("enter number of array elements\n");scanf("%d",&n);
printf("enter array elements\n");for(i=0; i<n;i++)
{
scanf("%d",&a[i]);
}
big=a[0];
for(i=0; i<n;i++)
{
if(a[i]>big)
big=a[i];
}
}

```

```
printf("the biggest element in an array is %d\n",big);
}
```

4. Write a C program to generate Fibonacci numbers using arrays.

```
#include<stdio.h>
void main()
{
    int i,n,a[10];a[0]=0;
    a[1]=1;
    printf("enter n\n");
    scanf("%d",&n); if(n==1)
    {
        printf("%d\t",a[0]);
    }
    if(n==2)
    {
        printf("%d\t %d\t",a[0],a[1]);
    }
    if(n>2)
    {
        printf("%d \t %d\t",a[0],a[1]);
        for(i=2;i<n;i++)
        {
            a[i]=a[i-1]+a[i-2];
            printf("%d\t",a[i]);
        }
    }
}
```

3.3 Two Dimensional arrays (Multidimensional array)

- In two dimensional arrays, elements will be arranged in rows and columns.
- To identify two dimensional arrays, we will use two indices (say i and j) where i index indicates row number and j index indicates column number.

Declaration of two-dimensional array

```
data_type array_name[exp1][exp2];

Or

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

- **data_type** can be int, float, char, double.
- **array_name** is the name of the array.
- **exp1 and exp2** indicates number of rows and columns

For example:

```
int a[2][3];
```

- ✓ The above statements allocate memory for $3 \times 4 = 12$ elements i.e $12 \times 2 = 24$ bytes.

Initialization of two-dimensional array

Assigning or providing the required values to a variable before processing is called initialization.

```
Data_type array_name[exp1][exp2]={
```

```
    {a1,a2,...an},
    {b1,b2, .bn},
    .....
    .....
    {z1,z2,...zn}

};
```

Data type can be int, float etc.

- exp1 and exp2 are enclosed within square brackets.
- both exp1 and exp2 can be integer constants or constant integer expressions (number of rows and number of columns).

a1 to an are the values assigned to 1st row.

- b1 to bn are the values assigned to 2nd row and so on. Example:

```
int a[3][3]={
    {10,20,30},
    {40,50,60},
    {70,80,90}
};
```

10	20	30
40	50	60
70	80	90

Partial Array Initialization

- If the number of values to be initialized is less than the size of array, then the elements are initialized from left to right one after the other.
- The remaining locations are initialized to zero automatically.
- Example:

```
int a[3][3]={
    {10,20},
    {40,50},
    {70,80}
};
```

Write a c program to copy one 2d array in to another 2d array

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int m,n,i,j,a[3][3],b[3][3];
    clrscr();
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array a elements\n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

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

```

        {
            b[i][j]=a[i][j];
        }
    }
    printf("matrix b is \n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d\t",b[i][j]);
        }
        printf("\n");
    }
}

```

Write a c program to find biggest element in a matrix or 2D array.

```

#include<stdio.h>void main()
{
    int m,n,i,j,a[3][3];clrscr();
    printf("enter number of rows and columns\n");scanf("%d %d",&m,&n);
    printf("enter array elements\n");for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    big=a[0][0]; for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            if(big>a[i][j])
                big=a[i][j];
        }
    }
    printf("big is %",big);
}

```


Write a C program to implement Matrix Multiplication

```

#include<stdio.h>void main()
{
    int m,n,i,j,sum,p,q,k,a[3][3],b[3][3],c[3][3];
    printf("enter number of rows and columns of matrix a \n");scanf("%d %d",&m,&n);
    printf("enter number of rows and columns of matrix b \n");scanf("%d %d",&p,&q);
    if(n!=p)
    {
        printf("multiplication not possible\n"):exit(0);
    }

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

    printf("enter array b elements\n");for(i=0;i<p;i++)
    {
        for(j=0;j<q;j++)
        {
            scanf("%d",&b[i][j]);
        }
    }

    for(i=0;i<m;i++)
    {
        for(j=0;j<q;j++)
        {
            {
                c[i][j]=0; for(k=0;k<n;k++)
                {
                    c[i][j]= c[i][j]+a[i][k]*b[k][j];
                }
            }
        }
    }
    printf("resultant matrix a is \n");

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

```

```

{
for(j=0;j<n;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
printf("resultant matrix a is \n");

for(i=0;i<p;i++)
{
for(j=0;j<q;j++)
{
printf("%d\t",b[i][j]);
}
printf("\n");
}
printf("resultant matrix a is \n");

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

```

Write a program to find sum of each row and sum of each column

```

#include<stdio.h>
void main()
{
int m,n,i,j,rsum,csum,a[3][3];
printf("enter number of rows and columns\n");
scanf("%d %d",&m,&n);
printf("enter array elements\n");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&a[i][j]);

```

```

        for(j=0;j<n;j++)
        {
            rsum=rsum+a[i][j];
        }
        printf("sum is %d",rsum);
    }

    for(i=0;i<m;i++)
    {
        csum=0;
        for(j=0;j<n;j++)
        {
            csum=csum+a[i][j];
        }
        printf("sum is %d",csum);
    }
7 }
#include<stdio.h>void main()
{
    int m,n,i,j,sum=0,a[3][3];
    printf("enter number of rows and columns\n");scanf("%d %d",&m,&n);
    printf("enter array elements\n");for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    for(i=0;i<m;i++)
    {
        sum=sum+a[i][j];
    }
    printf("sum is %d",rsum);
}

```

Real-world Applications of 1-D Arrays

- Lists and Sequences: Storing a list of names, numbers, or any type of data that needs to be organized sequentially.
- Counting and Accumulation: Keeping track of counts, scores, or incremental values.
- Data Retrieval: Accessing elements of a database or dataset.
- Mathematical Operations: Performing mathematical calculations using arrays.
- Text Processing: Storing and processing text or characters.

Real-world Applications of 2-D and 3-D Arrays

- Image Processing: Storing and manipulating pixel values in images.
- Game Development: Representing game boards, maps, and 3D environments.
- Scientific Computing: Storing and processing data from experiments and simulations.
- Matrices in Mathematics: Solving linear equations, transformations, and more.
- Databases: Organizing data in tabular form.

3.4 Strings**3.4.1 C Strings**

In C language a string is group of characters (or) array of characters, which is terminated by delimiter `\0` (null).

Thus, C uses variable-length delimited strings in programs.

3.4.2 Declaring Strings

C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.

Syntax: - `char string_name[size];`

The size determines the number of characters in the string name.

Ex: - `char city[10];`
`char name[30];`

3.4.3 Initializing strings

There are several methods to initialize values for string variables.

Ex:- `char str1[6]="HELLO";`

H	E	L	L	O	\0
---	---	---	---	---	----

Ex:- `char month[]="JANUARY";`

J	A	N	U	A	R	Y	\0
---	---	---	---	---	---	---	----

Ex:- `char city[8]="NEWYORK";`

`char city[8]={„N“,“E“,“W“,“Y“,“O“,“R“,“K“,“\0“};`

The string city size is 8 but it contains 7 characters and one-character space is for NULL terminator.

3.4.4 Operations on Strings

In this section, we will learn about different operations that can be performed on character arrays.

But before we start with these operations, we must understand the way arithmetic operators can be applied to characters.

In C, characters can be manipulated in the same way as we do with numbers. When we use a character constant or a character variable in an expression, it is automatically converted into an integer value, where the value depends on the local character set of the system. For example, if we write

```
int i;
```

```
char ch 'A';
```

```
i = ch;
```

```
printf("%d", i);
```

```
// Prints 65, ASCII value of ch is 'A'
```

C also enables programmers to perform arithmetic operations on character variables and character constants. So, if we write

```
int i;
```

```
char ch = 'A';
```

```
i = ch + 10;
```

```
printf("%d", i);
```

```
// Prints 75, ASCII value of ch that is 'A' + 10
```

Character variables and character constants can be used with relational operators as shown in the example below,

```
char ch = 'C';

if (ch >= 'A' && ch <= 'Z')

printf("The character is in upper case");
```

1. Finding the Length of a String

```
1 // C prog C:\Users\chira\OneDrive\Documents\C_DS prog\c length of string.c
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char Str[1000];
8     int i;
9
10    printf("Enter the String: ");
11    scanf("%s", Str);
12
13    for (i = 0; Str[i] != '\0'; ++i);
14
15    printf("Length of Str is %d", i);
16
17    return 0;
18 }
```

Output:

```
Enter the String: Hello
Length of Str is 5
Process returned 0 (0x0)   execution time : 3.391 s
Press any key to continue.
|
```

2. Converting Characters of a String into Upper Case

```
1  #include <stdio.h>
2  #include <conio.h>
3  #include <string.h>
4  int main()
5  {
6      char s[100];
7      int i;
8      printf("Enter a string : ");
9      gets(s);
10     for (i = 0; s[i]!='\0'; i++) {
11         if(s[i] >= 'a' && s[i] <= 'z') {
12             s[i] = s[i] - 32;
13         }
14     }
15     printf("String in Upper Case = %s", s);
16     return 0;
17 }
18
```

Output:

```
Enter a string : sapthagiri nps university
String in Upper Case = SAPTHAGIRI NPS UNIVERSITY
Process returned 0 (0x0)   execution time : 13.072 s
Press any key to continue.
```

3. Write a program to convert characters of a string into lower case.

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4
5  int main ()
6  {
7      int i = 0;
8      char c;
9      char str[] = "SAPTHAGIRI NPS UNIVERSITY";
10
11     while(str[i])
12     {
13         putchar(tolower(str[i]));
14         i++;
15     }
16
17     return(0);
18 }
19

```

Output:

```

sapthagiri nps university
Process returned 0 (0x0)   execution time : 0.483 s
Press any key to continue.

```

The string Input/ Output functions with example:

Reading and Writing strings: -

C language provides several string handling functions for input and output.

String Input/Output Functions: - C provides two basic methods to read and write strings.

Using formatted input/output functions and using a special set of functions.

Reading strings from terminal: -

formatted input function: - scanf can be used with %s format specification to read a string.

Ex: - char name[10];
scanf("%s",name);

Here don't use '&' because name of string is a pointer to array.

The problem with scanf is that it terminates its input on the first white space it finds.

Ex: - NEW YORK

Reads only NEW (from above example).

Unformatted input functions: -

getchar(): - It is used to read a single character from keyboard. Using this function repeatedly we may read entire line of text

Ex: - char ch="Z";
ch=getchar();

(2) gets(): - It is more convenient method of reading a string of text including blank spaces.

Ex: - char line[100];
gets(line);

Writing strings on to the screen: -

(1) Using formatted output functions: - printf with %s format specifier we can print strings in different formats on to screen.

Ex: - char name[10];
printf("%s",name);

Ex: - char name[10];
printf("%0.4",name);

/* If name is JANUARY prints only 4 characters ie., JANU */

J	A	N	U
---	---	---	---

Printf("%10.4s",name);

						J	A	N	U
--	--	--	--	--	--	---	---	---	---

printf("%-10.4s",name);

J	A	N	U						
---	---	---	---	--	--	--	--	--	--

3.4.5 Arrays of strings

✓ Declare and Initialize Array of strings in C:

We have array of integers, array of floating-point numbers, etc... similarly we have array of strings also.

Collection of strings is represented using array of strings.

Declaration: -

Char arr[row][col];

where,

arr - name of the array

row - represents number of strings

col - represents size of each string

Initialization: -

char arr[row][col] = {list of strings};

Example: - char city[5][10] = { "DELHI", "CHENNAI", "BANGALORE", "HYDERABAD", "MUMBAI" };

D	E	L	H	I	\0				
C	H	E	N	N	A	I	\0		
B	A	N	G	A	L	O	R	E	\0
H	Y	D	E	R	A	B	A	D	\0
M	U	M	B	A	I	\0			

In the above storage representation memory is wasted due to the fixed length for all strings.

3.5 Storage Classes

➤ There are following storage classes which can be used in a C Program:

- i. Automatic variables (Local variable)
- ii. External variables
- iii. Static variables
- iv. Register variables

3.5.1 Local variables (automatic variables)

- These are the variables which are defined within a function.
- These variables are also called as automatic variables.
- The scope of these variables is limited only to the function in which they are declared and cannot be accessed outside the function.

Example 1: (Automatic Variables)

```

int main()
{
int n1; // n1 is a local variable to main()
}
void func()
{
int n2; // n2 is a local variable to func()
}

```

- In the above example, n1 is local to main() and n2 is local to func().
- This means you cannot access the n1 variable inside func() as it only exists inside main().
- Similarly, you cannot access the n2 variable inside main() as it only exists inside func().

3.5.2 Global variables (External variable)

- These are the variables which are defined before all functions in global area of the program.
- Memory is allocated only once to these variables and initialized to zero.
- These variables can be accessed by any function and are alive and active throughout the program.
- Memory is deallocated when program execution is over.

Example: Global Variable

```

#include <stdio.h>
void display();
int n = 5; // global variable
int main()
{
    ++n;
    display();
    return 0;
}
void display()
{
    ++n;
    printf("n = %d", n);
}

```

Output: n = 7

3.5.3 Static variables

- The variables that are declared using the keyword static are called static

variables.

- The static variables can be declared outside the function and inside the function. They have the characteristics of both local and global variables.
- Static can also be defined within a function.

Ex:

```
static int a,b;
```

Example Program to illustrate static variables

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

```
void display();
int main()
{
    display();
    display();
}
void display()
{
    static int c = 1;
    c += 5; // c = c + 5
    printf("%d ",c);
}
```

OUTPUT:

```
6 11
```

- ✓ During the first function call, the value of 'c' is initialized to 1. Its value is increased by 5.
- ✓ Now, the value of 'c' is 6, which is printed on the screen.
- ✓ During the second function call, 'c' is not initialized to 1 again. It's because 'c' is a static variable.
- ✓ The value 'c' is increased by 5. Now, its value will be 11, which is printed on the screen.

3.5.4 Register variables

- ii. Any variables declared with the qualifier register is called a register variable.
- iii. This declaration instructs the compiler that the variable under use is to be stored in one of the registers but not in main memory.
- iv. Register access is much faster compared to memory access. Ex: `register int a;`

Example 1: (Register Variables)

```
#include <stdio.h>
int main()
{
    register int a; // variable a is allocated memory in the
                   // CPU register. The initial default value of a
                   // is 0.
    printf("%d",a);
}
```

OUTPUT:

0

Example 2: (Register Variables)

```
#include <stdio.h>
int main()
{
    register int count = 0;
    while (count < 5)
    {
        printf("Count: %d\n", count);
        count++;
    }
    return 0;
}
```

OUTPUT:

Count: 0
 Count: 1
 Count: 2
 Count: 3
 Count:

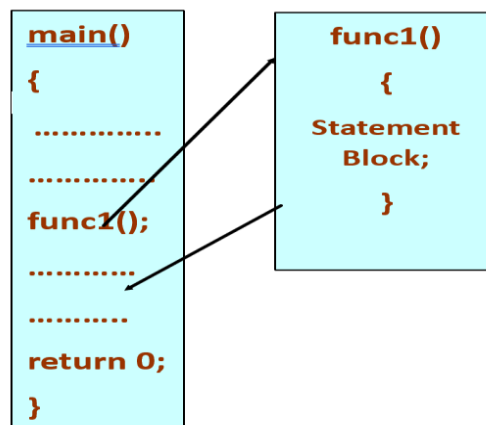
Course: Problem Solving Techniques using C

Course Code: 24BTPHY104

Module 4

4. FUNCTIONS AND POINTERS

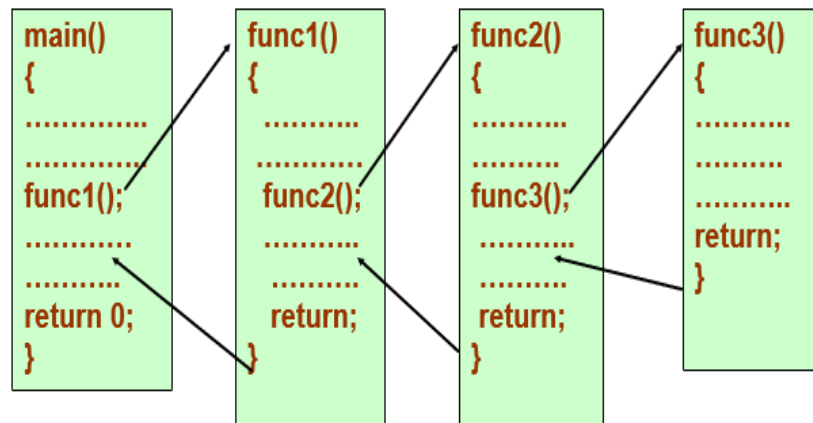
- A large C program is divided into basic building blocks called function. Function contains set of instructions enclosed by “{}” which performs specific operation in a program.
- Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from that of other functions.
- Every function has a name which acts as an interface to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is specified by the function name.



- In the above figure main() calls another function, func1() to perform a well-defined task.
- main() is known as the calling function and func1() is known as the called function.
- When the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function.
- After the called function is executed, the control is returned back to the calling program.
- It is not necessary that the main() can call only one function, it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for

loop, while loop or do-while loop may call the same function multiple times until the condition holds true.

- It is not that only the main() can call another functions. Any function can call any other function. In the fig. one function calls another, and the other function in turn calls some other function.



Basically, there are two categories of function:

- Predefined functions: available in C / C++ standard library such as `stdio.h`, `math.h`, `string.h` etc.
- User-defined functions: functions that programmers create for specialized tasks such as graphic and multimedia libraries, implementation extensions or dependent etc.

Why Do We Need Functions?

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- Understanding, coding and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function (except `main()`), then there will be countless lines in the `main()`.
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been prewritten and pre-tested, so the programmers use them without worrying about their code details. This speeds up program development.

Uses of Functions

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large programs.

4.1 Function Definition

- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.
- When a function defined, space is allocated for that function in the memory.
- The syntax of a function definition can be given as:

```
return_data_type function_name(data_type var1, data_type var2,...)
{
    .....
    statements
    .....
    return(variable);
}
```

- The no. and the order of arguments in the function header must be same as that given in function declaration statement.

- The length of your program can be reduced.
- It becomes easy
- Functions can be called several times within your program.

All variables declared inside a function are local variables and are not accessible outside the function.

Syntax:

```
return-value-type function-name( parameter-list)
{
declarations and statements
```


Where,

- function-name: any valid identifier
- Return-value-type:
 - data type of the result (default int)
 - void – indicates that the function returns nothing
- Parameter-list:
 - Also called formal parameters.
 - A list of variables, comma separated list, declares parameters:
 - A type must be listed explicitly for each parameter unless, the parameter is of type int.
- Declarations and statements: function body (block)
 - Variables can be declared inside blocks (can be nested)
 - You cannot create functions within inside other functions.
- Returning control
 - If nothing returned
 - return;
 - or, until reaches right brace
 - If something returned
 - return expression;

4.2 Function Prototyping

Function prototype is a statement that tells the compiler about the function's name, its return type, numbers and data types of its parameters.

Function prototype works like a function declaration where it is necessary where the function reference or call is present before the function definition but optional if the function definition is present before the function call in the program.

A prototype functions is only used when its implementation comes after the main function.

Syntax

```
return_type function_name(parameter_list);
```

where,

- **return_type**: It is the data type of the value that the function returns. It can be any data type int, float, void, etc. If the function does not return anything, void is used as the return type.

- **function_name:** It is the identifier of the function. Use appropriate names for the functions that specify the purpose of the function.
- **parameter_list:** It is the list of parameters that a function expects in parentheses. A parameter consists of its data type and name. If we don't want to pass any parameter, we can leave the parentheses empty.

Program to illustrate the Function Prototype

```
#include <stdio.h>

// Function prototype
float calculateRectangleArea(float length, float width);

int main()
{
    float length = 5.0;
    float width = 3.0;

    // Function call
    float area = calculateRectangleArea(length, width);

    printf("The area of the rectangle is: %.2f\n", area);

    return 0;
}

// Function definition
float calculateRectangleArea(float length, float width)
{
    return length * width;
}
```

```
}
```

Output

The area of the rectangle is: 15.00

4.3 Types of functions

A function may define with or without parameters, and it may or may not return a value. It entirely depends upon the user requirement. In Programming, as per our requirement, we can define the User-defined functions in multiple ways. The following are the types of Functions.

1. Function with no argument and no Return value.
2. With no argument and with a Return value.
3. With argument and No Return value.
4. With argument and Return value

4.3.1 Function with no argument and no Return value

In this method, we won't pass any arguments to the function while defining, declaring, or calling it. This type of functions in C will not return any value when we call the [function](#) from main() or any sub-function. When we are not expecting any return value, we need some statements to print as output.

Example:

```
#include<stdio.h>
// Declaration
void Addition();
void main()
{
printf("\n ..... \n");
Addition();
}
void Addition()
{
int Sum, a = 10, b = 20;
Sum = a + b;
```

```
printf("\n Sum of a = %d and b = %d is = %d", a, b, Sum);
}
```

Output:

```
Sum of a = 10 and b = 20 is = 30
```

4.3.2 With no argument and with a Return value.

In this method, we won't pass any arguments while defining, declaring, or calling the function. This C type of function will return some value when we call it from the main() or any sub method.

The Data Type of the return value will depend upon the return type of function declaration. For instance, if the return type is int, then the return value will be int.

In these types of functions program, we are going to calculate the multiplication of 2 integers using the user defined functions without arguments and return keywords.

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

```
int Multiplication();
```

```
int main()
```

```
{
    int Multi;
```

```
    Multi = Multiplication();
```

```
    printf("\n Multiplication of a and b is = %d \n", Multi );
```

```
    return 0;
```

```
}
```

```
int Multiplication()
```

```
{
```

```
    int Multi, a = 20, b = 40;
```

```
    Multi = a * b;
```

```

return Multi;
}

```

Output

```
Multiplication of a and b is = 800
```

4.3.3 With argument and No Return value.

This method allows us to pass the arguments to the function while calling it. But, This C type of function will not return any value when we call it from main () or any sub method. If we want to allow the user to pass his data to the arguments, but we are not expecting any return value.

These Types of Functions allows the user to enter 2 integers. Next, we are going to pass those values to the user-defined function to calculate the sum.

```

#include<stdio.h>
void Addition(int, int);
void main()
{
    int a, b;
    printf("\n Please Enter two integer values \n");
    scanf("%d %d",&a, &b);
    //Calling with dynamic values
    Addition(a, b);
}
void Addition(int a, int b)
{
    int Sum;
    Sum = a + b;
    printf("\n Addition of %d and %d is = %d \n", a, b, Sum);}

```

```
Please Enter two integer values
40
90

Addition of 40 and 90 is = 130
```

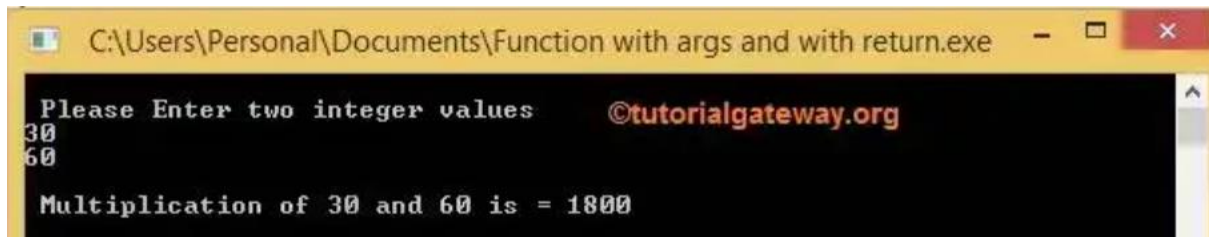
4.3.4 With argument and Return value

This method allows us to pass the arguments to the function while calling it. This type of function in C will return some value when we call it from the main () or any subfunction. The data Type of the return will depend upon the return type of function declaration. For instance, if the return type is int then the return value will be int. This type of user-defined function is called a fully dynamic one, and it provides maximum control to the end user. This Types of Functions allows the user to enter 2 integers. And then, we are going to pass those values to the UDFs to multiply them and return the output using the return keyword.

```
#include<stdio.h>
int Multiplication(int, int);
int main()
{
    int a, b, Multi;
    printf("\n Please Enter two integer values \n");
    scanf("%d %d",&a, &b);
    //Calling the with dynamic values
    Multi = Multiplication(a, b);
    printf("\n Multiplication of %d and %d is = %d \n", a, b, Multi);
    return 0;
}

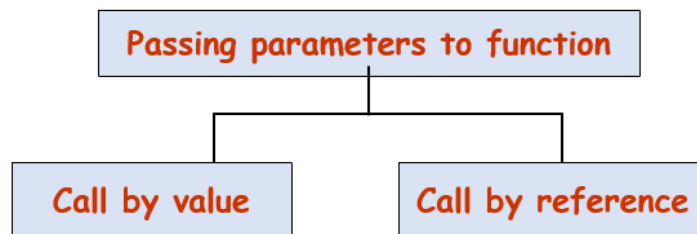
int Multiplication(int a, int b)
{
    int Multi;
```

```
Multi = a * b;  
return Multi;  
}
```



4.4 Passing arguments to functions

- There are two ways in which arguments or parameters can be passed to the called function.



- Call by value in which values of the variables are passed by the calling function to the called function.
- Call by reference in which address of the variables are passed by the calling function to the called function.

4.5 Passing arrays to functions

In C, the whole array cannot be passed as an argument to a function. However, you can pass a pointer to an array without an index by specifying the array's name.

Syntax

Three ways to pass an array as a parameter to the function. In the function definition, use the following syntax:

```
return_type foo ( array_type array_name[size], ...);
```

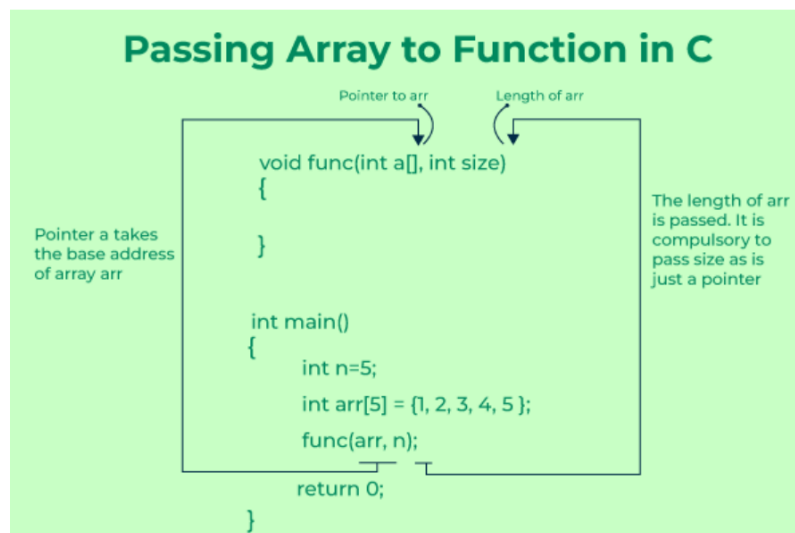
Mentioning the size of the array is optional. So the syntax can be written as:

```
return_type foo ( array_type array_name[], ...);
```

In both of the above syntax, even though we are defining the argument as array it will still be passed as a pointer. So we can also write the syntax as:

```
return_type foo ( array_type* array_name, ...);
```

But passing an array to function results in [array decay](#) due to which the array loses information about its size.



Program to pass the array as a function and check its size

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

```
#include <stdlib.h>
```

```
// Note that arr[] for fun is just a pointer even if square
```

```
// brackets are used. It is same as
```

```
// void fun(int *arr) or void fun(int arr[size])
```



```

void func(int arr[8])
{
    printf("Size of arr[] in func(): %d bytes",
        sizeof(arr));
}

// Drive code
int main()
{
    int arr[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };

    printf("Size of arr[] in main(): %dbytes\n",
        sizeof(arr));

    func(arr);

    return 0;
}

```

Output

Size of arr[] in main(): 32bytes

Size of arr[] in func(): 8 bytes

As we can see,

- The size of the arr[] in the main() function (where arr[] is declared) is 32 bytes which is $\text{sizeof(int)} * 8 = 4 * 8 = 32$ bytes.
- But in the function where the arr[] is passed as a parameter, the size of arr[] is shown as 8 bytes (which is the size of a pointer in C).

It is due to the fact that the array decays into a pointer after being passed as a parameter.

4.6 Passing strings to functions

String Function Arguments

In C, you can pass strings to functions in two ways:

- By passing an array of characters (i.e., a string)
- By passing a pointer to the first character in the string.

Passing an array of characters

Here's an example of passing an array of characters (string) to a function:

```
#include <stdio.h>
void print_string(char str[])
{
    printf("%s\n", str);
}
int main()
{
    char message[] = "Hello, world!";
    print_string(message);
    return 0;
}
```

Output:

Hello, world!

- The `print_string` function takes a single argument, `str`, which is an array of characters (i.e., a string).
- In the `main` function, a string `message` is defined and initialized with the value "Hello, world!". This string is then passed as an argument to the `print_string` function.

- The `print_string` function uses the `printf` function to print the string passed to it, along with a newline character to add a line break after the string.

Passing a pointer to the first character

And here's an example of passing a pointer to a string to a function:

```
#include <stdio.h>
void print_string(char *str)
{
    printf("%s\n", str);
}
int main()
{
    char message[] = "Hello, world!";
    print_string(message);
    return 0;
}
```

Output:

Hello, world!

Explanation

- The `main` function declares a string `message` with the value "Hello, world!", and passes it to the `print_string` function.
- The `print_string` function takes a pointer to a character as an argument and uses the `printf` function to print the string passed to it, along with a newline character.

4.7 Nested Functions

Some programmer thinks that defining a function inside another function is known as “nested function”. But the reality is that it is not a nested function, it is treated as lexical scoping. Lexical scoping is not valid in C because the compiler cannot reach/find the correct memory location of the inner function.

Nested function **is not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it's not a nested function. Because nested functions definitions cannot [access local variables](#) of the surrounding blocks, they can access only global variables of the containing module. This is done so that lookup of global variables doesn't have to go through the directory. As in C, there are two nested scopes: local and global (and beyond this, built-ins). Therefore, nested functions have only a limited use. If we try to approach nested function in C, then we will get compile time error.

// C program to illustrate the concept of Nested function.

```
#include <stdio.h>
int main(void)
{
    printf("Main");
    int fun()
    {
        printf("fun");

        // defining view() function inside fun() function.
        int view()
        {
            printf("view");
        }
        return 1;
    }
    view();
}
```

Output:

Compile time error: undefined reference to `view'

4.8 Call by value

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it.
- Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function.

In the calling function no change will be made to the value of the variables.

```
#include<stdio.h>
void add( int n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add(int n)
{
    n = n + 10;
    printf("\n The value of num in the called function = %d", n);
}
```

Output:

The value of num before calling the function = 2

The value of num in the called function = 12

The value of num after calling the function = 2

The following points are to be noted while passing arguments to a function using the call-by-value method.

- When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.
- The values of the arguments passed by the function are copied into the newly created variables.
- Arguments are called by value when the called function does not need to modify the values of the original variables in the calling function.
- Values of the variables in the calling function remain unaffected when the arguments are passed by using call-by-value technique.
- Therefore, call-by-value method of passing arguments to a function must be used only in two cases:
 - When the called function does not need to modify the value of the actual parameter. It simply uses the value of the parameter to perform its task.
 - When we want the called function should only temporarily modify the value of the variables and not permanently. So, although the called function may modify the value of the variables, these variables remain unchanged in the calling function.

4.9 Call by reference

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.

- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

```
#include<stdio.h>
void add( int *n);
int main()
{
    int num = 2;
    printf("\n The value of num before calling the function = %d", num);
    add(&num);
    printf("\n The value of num after calling the function = %d", num);
    return 0;
}
void add(int *n)
{
    *n = *n + 10;
    printf("\n The value of num in the called function = %d", *n);
}
```

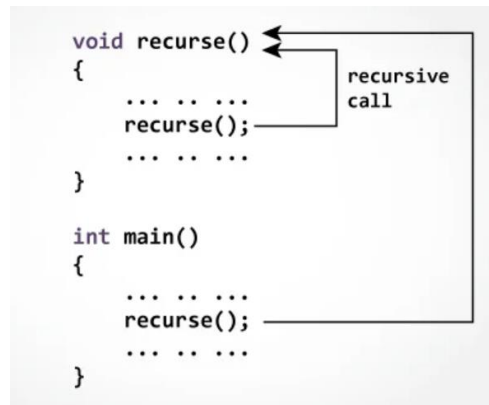
Output:

The value of num before calling the function = 2

The value of num in the called function = 12

4.10 Recursive functions

A recursive function is a function that calls itself either directly or indirectly through another function. We cannot define function within function, but we can call the same function within that function.



The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, [if...else statement](#) (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Classic example for recursive function is sum of Natural number using Recursion

```

#include <stdio.h>
int sum(int n);
int main() {
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum = %d", result);
    return 0;
}
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}

```


Output:

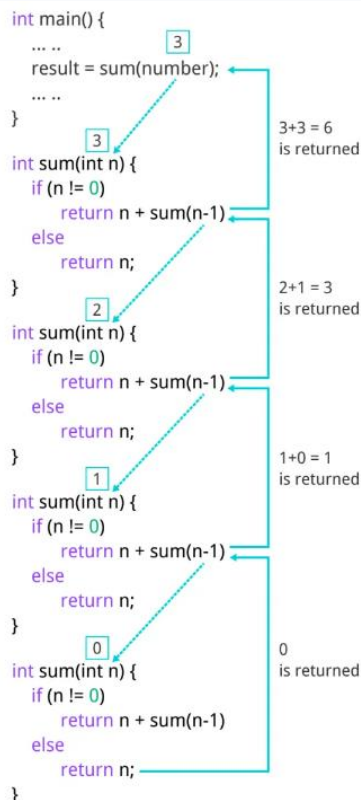
Enter a positive integer:3

sum = 6

Initially, the `sum()` is called from the `main()` function with `number` passed as an argument.

Suppose, the value of `n` inside `sum()` is 3 initially. During the next function call, 2 is passed to the `sum()` function. This process continues until `n` is equal to 0.

When `n` is equal to 0, the `if` condition fails and the `else` part is executed returning the sum of integers ultimately to the `main()` function.



4.11 Pointers

Pointer is a variable that stores the address of another variable. A pointer in c is used to allocate memory dynamically at run time. The Pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Syntax:

```
Data_type*var_name;
```

Example:

```
Int*p;
```

```
Char*p;
```

4.11.1 Declaration of Pointer variable

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

4.11.2 Pointer arithmetic

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers.

The pointer variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language.

The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. These operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

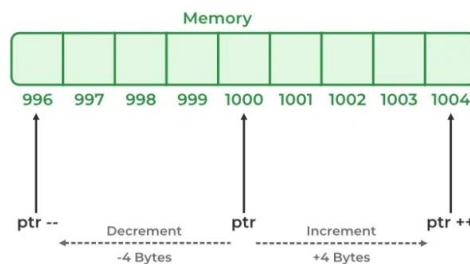
If an integer pointer that stores **address 1000** is incremented, then it will increment by 4(**size of an int**), and the new address will point to **1004**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by 4(**size of an int**), and the new address will point to **996**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.

Pointer Increment & Decrement



Example of Pointer Increment and Decrement

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

```
// pointer increment and decrement
```

```
//pointers are incremented and decremented by the size of the data type they point to
```

```
int main()
```

```
{
```

```
int a = 22;
```

```
int *p = &a;
```

```
printf("p = %u\n", p); // p = 6422288
```

```

p++;
printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes
p--;
printf("p-- = %u\n", p); //p-- = 6422288      -4 // restored to original value

float b = 22.22;
float *q = &b;
printf("q = %u\n", q); //q = 6422284
q++;
printf("q++ = %u\n", q); //q++ = 6422288      +4 // 4 bytes
q--;
printf("q-- = %u\n", q); //q-- = 6422284      -4 // restored to original value

char c = 'a';
char *r = &c;
printf("r = %u\n", r); //r = 6422283
r++;
printf("r++ = %u\n", r); //r++ = 6422284      +1 // 1 byte
r--;
printf("r-- = %u\n", r); //r-- = 6422283      -1 // restored to original value

return 0;
}

```

Output

```

p = 1441900792
p++ = 1441900796
p-- = 1441900792
q = 1441900796
q++ = 1441900800

```

```
q-- = 1441900796
```

```
r = 1441900791
```

```
r++ = 1441900792
```

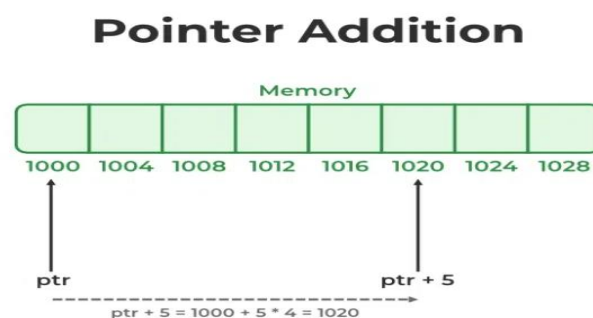
```
r-- = 1441900791
```

Addition of Integer to Pointer

When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we add integer 5 to it using the expression, **ptr = ptr + 5**, then, the final address stored in the ptr will be **ptr = 1000 + sizeof(int) * 5 = 1020**.



Example of Addition of Integer to Pointer

// C program to illustrate pointer Addition

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

```
// Driver Code
```

```
int main()
```

```
{
```

```
    // Integer variable
```

```
    int N = 4;
```

```
    // Pointer to an integer
```

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

```

// Pointer stores the address of N
ptr1 = &N;
ptr2 = &N;
printf("Pointer ptr2 before Addition: ");
printf("%p \n", ptr2);
// Addition of 3 to ptr2
ptr2 = ptr2 + 3;
printf("Pointer ptr2 after Addition: ");
printf("%p \n", ptr2);
return 0;
}

```

Output:

Pointer ptr2 before Addition: 0x7ffca373da9c

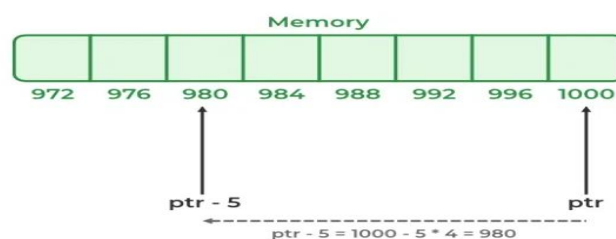
Pointer ptr2 after Addition: 0x7ffca373daa8

Subtraction of Integer to Pointer

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

For Example:

Consider the same example as above where the **ptr** is an **integer pointer** that stores **1000** as an address. If we subtract integer 5 from it using the expression, **ptr = ptr - 5**, then, the final address stored in the ptr will be **ptr = 1000 - sizeof(int) * 5 = 980**.

Pointer Subtraction

Example of Subtraction of Integer from Pointer, below is the program to illustrate pointer

Subtraction:

```
// Program to illustrate pointer Subtraction
#include <stdio.h>
// Driver Code
int main()
{
    // Integer variable
    int N = 4;
    // Pointer to an integer
    int *ptr1, *ptr2;
    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;
    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);
    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);
    return 0;
}
```

Output

Pointer ptr2 before Subtraction: 0x7ffd718ffebc

Pointer ptr2 after Subtraction: 0x7ffd718ffeb0

Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1004)** are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the **increment between ptr1 and ptr2** is given by **(4/4) = 1**.

Example of Subtraction of Two Pointer

```
// Program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
    int x = 6; // Integer variable declaration
    int N = 4;
    // Pointer declaration
    int *ptr1, *ptr2;
    ptr1 = &N; // stores address of N
    ptr2 = &x; // stores address of x
    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
    // %p gives an hexa-decimal value,
    // We convert it into an unsigned int value by using %u

    // Subtraction of ptr2 and ptr1
    x = ptr1 - ptr2;

    // Print x to get the Increment
    // between ptr1 and ptr2
    printf("Subtraction of ptr1 "
           "& ptr2 is %d\n",
           x);
```



```

    return 0;
}

```

Output

```
ptr1 = 2715594428, ptr2 = 2715594424
```

```
Subtraction of ptr1 & ptr2 is 1
```

Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C `>`, `>=`, `<`, `<=`, `==`, `!=`. It returns true for the valid condition and returns false for the unsatisfied condition.

1. **Step 1:** Initialize the integer values and point these integer values to the pointer.
2. **Step 2:** Now, check the condition by using comparison or relational operators on pointer variables.
3. **Step 3:** Display the output.

Example of Pointer Comparison

```

// Program to illustrate pointer comparison
#include <stdio.h>
int main()
{
    // declaring array
    int arr[5];
    // declaring pointer to array name
    int* ptr1 = &arr;
    // declaring pointer to first element
    int* ptr2 = &arr[0];
    if (ptr1 == ptr2) {
        printf("Pointer to Array Name and First Element "
            "are Equal.");
    }
}

```

```

    }
    else {
        printf("Pointer to Array Name and First Element "
            "are not Equal.");
    }
    return 0;
}

```

Output

Pointer to Array Name and First Element are Equal.

4.11.3 Pointers and Functions

- Pointer as a function parameter is used to hold addresses of arguments passed during function call.
- This is also known as call by reference.
- When a function is called by reference any change made to the reference variable will effect the original variable.

Example

```

#include <stdio.h>

void exchange(int *a, int *b);

int main()
{
    int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n);
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n); return 0;
}

void exchange (int *a, int *b)
{

```

```

int temp;
temp = *a;
*a = *b;
*b = temp;
}

```

Output

```

m = 10
n = 20
After Swapping:
m = 20
n = 10

```

4.11.4 Call by value

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Example

```

#include<stdio.h>

void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}

int main() {
    int x=100;

```

```

    printf("Before function call x=%d \n", x);
    change(x); //passing value in function
    printf("After function call x=%d \n", x);
    return 0;
}

```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

Call by Value Example: Swapping the values of the two variables

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

```
void swap(int , int); //prototype of the function
```

```
int main()
```

```
{
```

```
    int a = 10;
```

```
    int b = 20;
```

```
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b
    in main
```

```
    swap(a,b);
```

```
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do
    not change by changing the formal parameters in call by value, a = 10, b = 20
```

```
}
```

```
void swap (int a, int b)
```

```
{
```

```
    int temp;
```

```
    temp = a;
```

```
    a=b;
```

```
    b=temp;
```

```
    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b =
```

```
10
```

```
}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

4.11.5 Call by reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x); //passing reference in function
    printf("After function call x=%d \n", x);
    return 0;
}
```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

Call by reference Example: Swapping the values of the two variables

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

```
void swap(int *, int *); //prototype of the function
```

```
int main()
```

```
{
```

```
int a = 10;
```

```
int b = 20;
```

```
printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
```

```
swap(&a,&b);
```

```
printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
```

```
}
```

```
void swap (int *a, int *b)
```

```
{
```

```
int temp;
```

```
temp = *a;
```

```
*a=*b;
```

```
*b=temp;
```

```
printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
```

```
}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

Difference between call by value and call by reference

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

4.12 Pointers and Arrays

- The address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.
- `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.
- Similarly, `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example:

```
#include<stdio.h>

int main()
{
    int i, x[20], sum = 0,n;
    printf("Enter the value of n: ");
    scanf("%d",&n);
    printf("Enter number one by one\n");
    for(i = 0; i < n; ++i)
```

```

{
/* Equivalent to scanf("%d", &x[i]); */ scanf("%d", x+i);
// Equivalent to sum += x[i]
sum += *(x+i);
}
printf("Sum = %d", sum);
return 0;
}

```

Output

Enter the value of n: 5

Enter number one by one

5

10

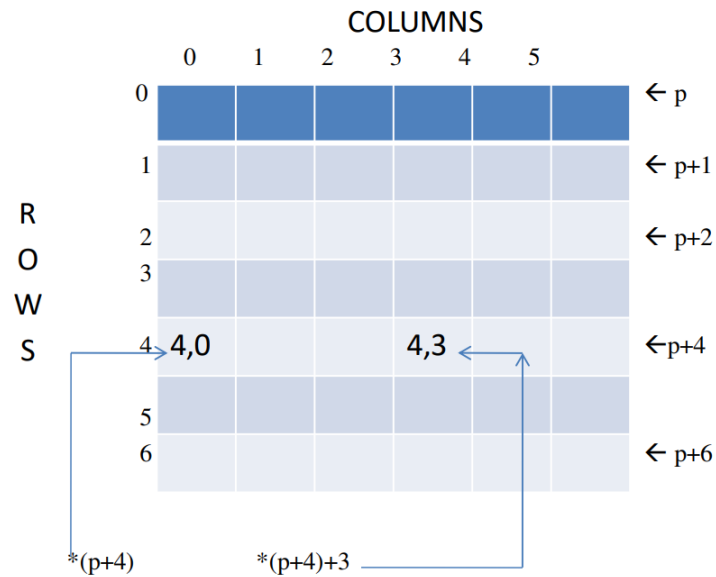
15

20

25

Sum = 75

- Pointers can be used to manipulate two-dimensional arrays also.
- A two-dimensional array can be represented by the pointer expression as follows
- $*(*(a+i)+j)$ or $*(*(p+1)+j)$



p -> pointer to first row

p+i -> pointer to ith row

* (p+i) -> pointer to first element in the ith row

* $(p+i) + j$ -> pointer to j th element in the i th row

((p+i)+j) ->valu stored in the ith row and jth columns.

Example

```
#include
```

```
int main()
```

 $\{$

```
int arr[3][4] = { {11,22,33,44}, {55,66,77,88},{11,66,77,44}};
```

```
int i, j;
```

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

 $\{$

```
printf("Address of %d th array %u \n",i , *(arr + i));
```

```
for(j = 0; j < 4; j++)
```

{

```
printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
```

}

```

printf("\n\n");
}
// signal to operating system program ran fine
return 0;
}

```

Output

```

arr[2][3]=44Address of 0 th array 2692284448
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44
Address of 1 th array 2692284464
arr[1][0]=55
arr[1][1]=66
arr[1][3]=88
Address of 2 th array 2692284480
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44

```

4.13 Arrays of Pointers

A pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in Programming when we want to point at multiple memory locations of a similar data type in program. We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.

- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Example

```
char name[4][25];
```

- The name is a table containing four names, each with maximum of 25 characters.
- The total storage requirements are 75 bytes.
- The individual strings will of equal lengths.

Example

```
char *names[4] = { "Anu", "Banu", "Chandru", "Deepak" };
```

- Declares name to be an array of four pointers to characters, each pointer pointing to a particular name.

```
#include
const int MAX = 4;
int main ()
{
char *names[] = { "Anu", "Banu", "Chandru", "Deepak" };
int i = 0;
for ( i = 0; i < MAX; i++)
{
printf("Value of names[%d] = %s\n", i, names[i] );
}
return 0;
}
```

Output

Value of names[0] = Anu

Value of names[1] = Banu

Value of names[2] = Chandru

Value of names[3] =Deepak

4.14 Pointers and Structures

- We know that the name of an array stands for the address of its zero-th element.
- Also true for the names of arrays of structure variables.

Example

```
struct inventory
```

```
{
    int no;
    char name[30];
    float price;
}
product[5], *ptr ;
```

- The name product represents the address of the zero-th element of the structure array.
- ptr is a pointer to data objects of the type struct inventory.
- The assignment

```
ptr = product ;
```

will assign the address of product [0] to ptr.

- Its member can be access

```
ptr ->name ;
```

```
ptr -> no ;
```

```
ptr -> price;
```

The symbol “->” is called the arrow operator or member selection operator.

- When the pointer ptr is incremented by one (ptr++) :The value of ptr is actually increased by sizeof(inventory).
- It is made to point to the next record.
- We can also use the notation (*ptr).no;
- When using structure pointers, we should take care of operator precedence.

- Member operator “.” has higher precedence than “*”.
- ptr -> no and (*ptr).no mean the same thing.
- ptr.no will lead to error.
- The operator “->” enjoys the highest priority among operators
- ++ptr -> no will increment roll, not ptr.
- (++ptr)-> -> no will do the intended thing.

Example

```
void main ()
{
struct book
{
char name[25];
char author[25];
int edn;
};
struct book b1 = { "Programming in C", "E Balagurusamy", 2 };
struct book *ptr ; ptr = &b1 ;
printf ( "\n%s %s edition %d ", b1.name, b1.author, b1.edn ) ;
printf ( "\n%s %s edition %d", ptr->name, ptr->author, ptr->edn ) ;
}
```

Output

```
Programming in C E Balagurusamy edition 2
Programming in C E Balagurusamy edition 2
```

4.15 Meaning of static and dynamic memory allocation

Static Memory Allocation

When the allocation of memory performs at the compile time, then it is known as static memory. In this, the memory is allocated for variables by the compiler.

OR

In static memory allocation whenever the program executes it fixes the size that the program is going to take, and it can't be changed further. So, the exact memory requirements must be known before. Allocation and deallocation of memory will be done by the compiler automatically. When everything is done at compile time (or) before run time, it is called static memory allocation.

Dynamic Memory Allocation

When the memory allocation is done at the execution or run time, then it is called dynamic memory allocation

OR

In Dynamic memory allocation size initialization and allocation are done by the programmer. It is managed and served with pointers that point to the newly allocated memory space in an area which we call the heap. Heap memory is unorganized and it is treated as a resource when you require the use of it if not release it. When everything is done during run time or execution time it is known as Dynamic memory allocation.

4.16 Memory Allocation Functions

Dynamic Memory allocation is possible by 4 functions of `stdlib.h` header file.

1. `malloc()`
2. `calloc()`
3. `free()`
4. `realloc()`

1. `malloc()`

The “**malloc**” or “**memory allocation**” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

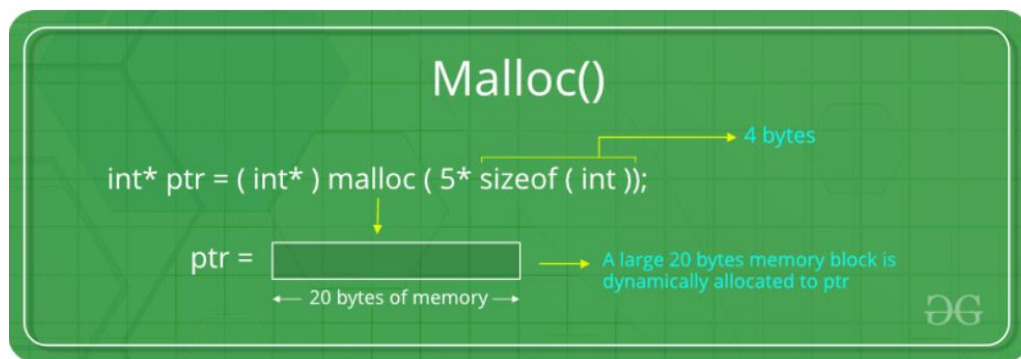
Syntax of `malloc()` in C

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

For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



If space is insufficient, allocation fails and returns a NULL pointer.

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
// This pointer will hold the
```

```
// base address of the block created
```

```
int* ptr;
```

```
int n, i;
```

```
// Get the number of elements for the array
```

```

printf("Enter number of elements:");
scanf("%d",&n);
printf("Entered number of elements: %d\n", n);

// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {
    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }
    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }
}

return 0;
}

```

Output

Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,

2. Calloc() method

1. “**calloc**” or “**contiguous allocation**” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

Syntax of calloc() in C

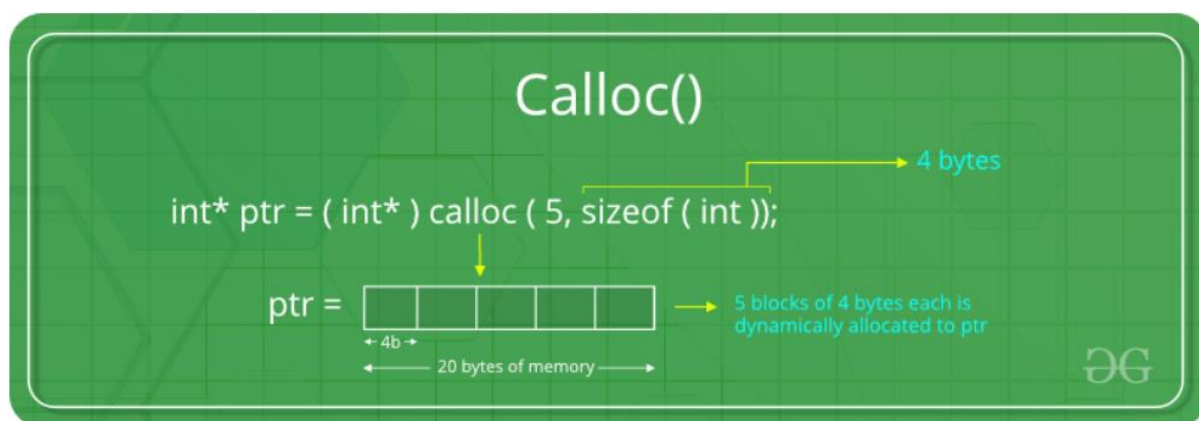
```
ptr = (cast-type*)calloc(n, element-size);
```

here, n is the no. of elements and element-size is the size of each element.

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of calloc()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");
    }
}
```

```
// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

return 0;
}
```

Output

Enter number of elements: 5

Memory successfully allocated using calloc.

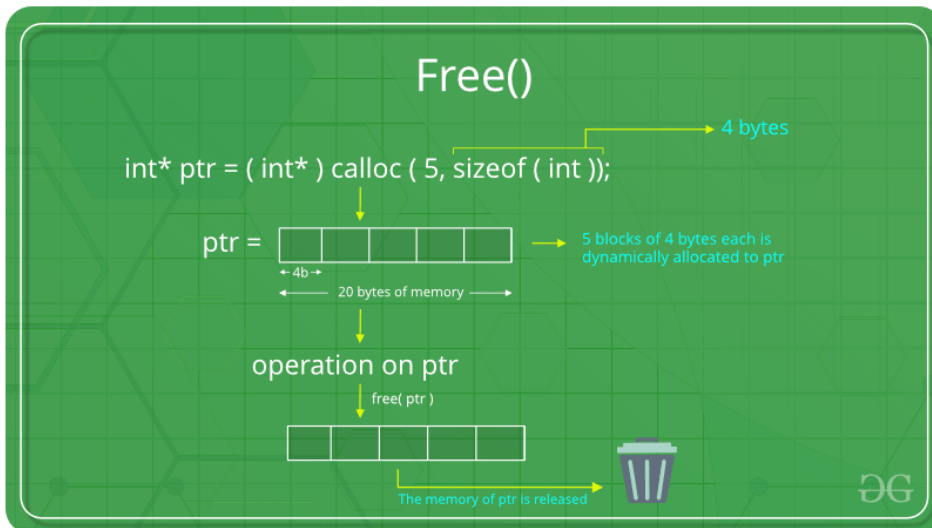
The elements of the array are: 1, 2, 3, 4, 5,

3. free() method

“**free**” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```



Example of free()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
```

```

// Dynamically allocate memory using calloc()
ptr1 = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using malloc.\n");

    // Free the memory
    free(ptr);
    printf("Malloc Memory successfully freed.\n");

    // Memory has been successfully allocated
    printf("\nMemory successfully allocated using calloc.\n");

    // Free the memory
    free(ptr1);
    printf("Calloc Memory successfully freed.\n");
}

return 0;
}

```

Output

Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.

4.realloc() method

“**realloc**” or “**re-allocation**” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.

If space is insufficient, allocation fails and returns a NULL pointer.

Example of realloc()

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
```

```
// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    // Get the new size for the array
```

```

n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);

// Dynamically re-allocate memory using realloc()
ptr = (int*)realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

free(ptr);
}

return 0;
}

```

Output

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Macros: Definition—types of Macros—Creating and implementing user defined header files.

4.17 Macros

The macro in C language is known as the piece of code which can be replaced by the macro value. The macro is defined with the help of **#define preprocessor directive** and the macro doesn't end with a semicolon (;). Macro is just a name given to certain values or expressions it doesn't point to any memory location.

Whenever the compiler encounters the macro, it replaces the macro name with the macro value. Two macros could not have the same name.

In C, we must use the **#define** directive, then the macro's name and value, to define a macro. The following is the syntax to define a macro.

#define MACRO_NAME macro_value

For example, if you want to define a macro for the value of pi, you can write:

#define PI 3.14159

Here, we will have the three components:

1. **#define** - Preprocessor Directive
2. **PI** - Macro Name
3. **3.14** - Macro Value

**Example:**

```
#include<stdio.h>
// This is macro definition
#define PI 3.14
void main()
{
    // declaration and initialization of radius
    int radius = 5;
    // declaration and calculating the area
    float area = PI * (radius*radius);
    // Printing the area of circle
    printf("Area of circle is %f", area);
}
```

Output:

Area of circle is 78.500000

In the above code example, we wrote a program to calculate the area of a circle with the help of a given radius. We defined the PI constant as a macro with the value 3.14

4.17.1 Types of Macros

The macro in C language is classified based on the different types of values that it replaces.

The different types of macros are

Object like Macros

An object like macros in C programming is simply the macros that get replaced by certain values or segments of code.

In the above example, in the introduction, we saw the macro with the name PI and value 3.14 it is an example of an object like macros.

// Examples of object like macros in C language

```
#define MAX 100
```

```
#define MIN 1
```

```
#define GRAVITY 9.8
```

```
#define NAME "Scaler"
```

```
#define TRUE 1
```

```
#define FALSE 0
```

Function Like Macros

In the function like macros are very similar to the actual function in C programming.

We can pass the arguments with the macro name and perform the actions in the code segment.

In macros, there is no type checking of arguments so we can use it as an advantage to pass different datatypes in same macros in C language.

Let's consider the following code example which is the modification of the previous code -

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

```
//object like macro
```

```
#define PI 3.14

// function like macro

#define Area(r) (PI*(r*r))

void main()

{

    // declaration and initialization of radius

    float radius = 2.5;

    // declaring and assigning the value to area

    float area = Area(radius);

    // Printing the area of circle

    printf("Area of circle is %f\n", area);

    // Using radius as int data type

    int radiusInt = 5;

    printf("Area of circle is %f", Area(radiusInt));

}
```

Output:

Area of circle is 19.625000

Area of circle is 78.500000

See in the above code, we added the function-like macro at line no. 7.

The macro name is Area which takes the argument r as a radius that we have passed at line no. 15 and line no. 22.

At the time of preprocessing the value Area(radius) gets replaced with the processed macro value and it is assigned to the area variable.

At line no. 15, we passed the value of radius as a float and at line no. 22 we passed the value of radius as a float of type integer. So, macros gave us the advantage to use the same macro for different datatypes because there is no type checking of arguments.

Chain Like Macros

When we use one macro inside another macro, then it is known as the chain-like macro.

We already saw the example of a chain-like macro in the above code example where we used the PI in Area. Let's discuss it a little more -

```
#define PI 3.14  
#define Area(r) (PI*(r*r))
```

In the above code snippet, we can see we used the object like macro PI inside the function like macro Area.

Here first parent macro gets expanded i.e. the functional macro Area and then the child macro gets expanded i.e. PI. This way when we use one macro inside another macro, it is called a chain macro.

Sr. No.	Macro Name	Description
1	Object Like	Value or code segment gets replaced with macro name
2	Function Like	The macro which takes an argument and acts as a function
3	Chain Like	The macro inside another macro

4.17.2 Creating and Implementing User Defined Header Files

Creating and implementing user-defined header files in C using macros is a fundamental aspect of modular programming. Header files typically contain function prototypes, macro definitions, type declarations, and other declarations required for the implementation of a module. Here's a step-by-step guide on how to create and implement user-defined header files in C using macros:

1. Create the Header File:

Start by creating a header file with a `.h` extension. This file will contain declarations and macro definitions that you want to share across multiple source files.

```
// myheader.h
#ifndef MYHEADER_H
#define MYHEADER_H
// Macro definition
#define SQUARE(x) ((x) * (x))
// Function prototype
int add(int a, int b);
#endif
...
```

In this example, `myheader.h` declares a macro `SQUARE(x)` and a function prototype `add`.

2. Implement the Functions:

Implement the functions declared in the header file in a separate source file (`.c` file).

```
// myfunctions.c
#include "myheader.h"
int add(int a, int b) {
    return a + b;
}
```

Ensure that you include the header file `myheader.h` in the source file that contains the implementations.

3. Include the Header File:

In any source file where you want to use the declarations and macros defined in `myheader.h`, include the header file at the beginning.

```
// main.c
#include <stdio.h>
#include "myheader.h"
int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    int squareResult = SQUARE(5);
    printf("Square Result: %d\n", squareResult);
    return 0;
}
```

4. Compile:

When compiling your program, ensure that you include all relevant source files.

...

```
gcc main.c myfunctions.c -o myprogram
```

...

This command compiles `main.c` and `myfunctions.c` into an executable named `myprogram`.

5. Execute:

Run the compiled program.

```
./myprogram
```

The output is printed in `main.c`, which uses the functions and macros defined in `myheader.h`. By following these steps, you can create and implement user-defined header files in C using macros effectively. This modular approach enhances code organization, readability, and reusability.

Course: Problem Solving Techniques using C**Course Code: 24BTPHY104****Module 5****5. STRUCTURES AND UNIONS**

- Structures and unions are two of many user defined data types in C.
- Both are similar to each other but there are quite some significant differences.

5.1 Structures

- A structure simply can be defined as a user-defined data type which groups logically related items under one single unit. We can use all the different data items by accessing that single unit.
- All the data items are stored in contiguous memory locations.
- It is not only permitted to one single data type items and it can store items of different data items.

5.1.1 Declaring a Structure

```

struct structure_name
{
    data_type variable_name;
    data_type variable_name;
    .....
    data_type variable_name;
};

```

Example 1:

```

struct student
{
    char name[100];
    int roll;
    float marks;
};

```

Example 2:

Suppose we need to store the details of an employee. It includes ID, name, age, salary, designation and several other factors which belong to different data types.

```

struct Empl
{
int emp_id;
float salary;
char designation[20];
int depart_no;
int age_of_emp;
};

```

5.1.2 Initialization of Structure

Like any other data type, structure variable can also be initialized at compile time.

```

struct Patient
{
float height;
int weight;
int age;
};

struct Patient p1 = { 180.75 , 73, 23 }; //initialization

```

[OR]

```

struct Patient p1;
p1.height = 180.75; //initialization of each member separately
p1.weight = 73;
p1.age = 23;

```

Example :

C Program to declare a structure student with field name and roll no. take input from user and display them.

```

#include<stdio.h>
struct student
{
char name[10];

```

```

int roll;
};
void main()
{
    struct student s1;
    clrscr();
    printf("\n Enter student record\n");
    printf("\n student name\t");
    scanf("%s",s1.name);
    printf("\nEnter student roll\t");
    scanf("%d",&s1.roll);
    printf("\nstudent name is %s",s1.name);
    printf("\nroll is %d",s1.roll);
    getch( );
}

```

OUTPUT:

```

Enter student record
Student name yaswanth
Enter student roll 35

```

```

Student name is yaswanth
roll is 35

```

5.2 Unions

- A union is a user-defined data type similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member can hold a value at any given time.
- The size of the union is determined by the size of its largest member.
- Unions are useful for memory management and situations where a variable may take on different types of values at different times.

Syntax of union

```

union UnionName {

```

```

dataType member1;
dataType member2;
// more members
};

```

5.2.1 Declaring Union

```

union Data {
    int i;
    float f;
    char str[20];
};

```

5.2.1 Initialization and Usage

```

union Data data;
data.i = 10;
printf("data.i: %d\n", data.i);

data.f = 220.5;
printf("data.f: %.2f\n", data.f);

strcpy(data.str, "C Programming");
printf("data.str: %s\n", data.str);

```

Defining of Union

- A union has to be defined, before it can be used.
- The syntax of defining a union is

```

union <union_name>
{
    <data_type> <variable_name>;
    <data_type> <variable_name>;
    .....
    <data_type> <variable_name>;
};

```

Example:

To define a simple union of a char variable and an integer variable

```

union shared
{
char c;
int i;
};

```

This union, **shared**, can be used to create instances of a union that can hold either a character value(c) or an integer value(i).

Union Data Type

- A union is a user defined data type like structure.
- The union groups logically related variables into a single unit.
- The union data type allocates the space equal to space needed to hold the largest data member of union.
- The union allows different types of variable to share same space in memory.
- The method to declare, use and access the union is same as structure.

Difference between Structures and Union

- **Memory Allocation:**
 - **Structures:** Each member has its own memory location. The total size of the structure is the sum of the sizes of all members.
 - **Unions:** All members share the same memory location. The size of the union is the size of its largest member.
- **Usage:**
 - **Structures:** Used when you need to store multiple related data items together.
 - **Unions:** Used when you need to store one of several possible data items in the same memory location, optimizing for memory usage.

5.3 Nested Structures in C

A **nested structure** in C is a structure within structure.

Syntax of Nested Structure in C

```

struct outer_struct
{

```

```

// outer structure members
int outer_member1;
float outer_member2;
// nested structure definition
struct inner_struct
{
    // inner structure members
    int inner_member1;
    float inner_member2;
} inner;
};

```

Example of Nested Structure in C:

```

struct school
{
    int numberOfStudents;
    int numberOfTeachers;
    struct student{
        char name[50];
        int class;
        int roll_Number;
    } std;
};

```

In the above example of nested structure in C, there are two structures **Student (depended structure)** and another structure called **School(Outer structure)**.

- The structure School has the data members like numberOfStudents, numberOfTeachers, and the Student structure is nested inside the structure School and it has the data members like name, class, roll_Number.
- To access the members of the school structure, you use dot notation.
- For example, to access the numberOfStudents member of the school structure, you would use the following syntax:

```
struct school s;
```

```
s.numberOfStudents = 100;
```

- To access the members of the nested student structure, you would use the following syntax:
 - strcpy(s.std.name, "John Smith");
 - s.std.class = 10;
 - s.std.roll_Number = 1;
- Note that the dot notation is used to access the members of the std structure, which is a member of the school structure.
- The nested structure student contains three members: name, class, and roll_Number.
- This structure can be useful in situations where you need to store information about a student, such as their name, class, and roll number.
- The structure can be nested in the following ways.
 1. Separate structure
 2. Embedded structure

1. Separate structure

```
struct Date
{
    int dd;
    int mm;
    int yyyy;
};
struct Employee
{
    int id;
    char name[20];
    struct Date doj;
}emp1;
```

- doj (date of joining) is the variable of type Date.
- Here doj is used as a member in Employee structure.

Embedded structure

- The embedded structure enables us to declare the structure inside the structure.
- Hence, it requires less line of codes but it cannot be used in multiple data structures.

struct Employee

```
{
    int id;
    char name[20];
    struct Date
    {
        int dd;
        int mm;
        int yyyy;
    }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

e1.doj.dd

e1.doj.mm

e1.doj.yyyy

5.4 Arrays of Structures in C

- An array whose elements are of type structure is called array of structure.
- It is generally useful when we need multiple structure variables in our program.

- To create an array of structure, first structure is declared and then array of structure is declared just like an ordinary array.

Example: (Array of Structure)

```
struct employee
{
    int emp_id;
    char name[20];
    char dept[20];
    float salary;
};
```

- An array of structure can be created as follows:

```
struct employee emp[10]; /* This is array of structure */
```

- In this example, the first structure employee is declared, then the array of structure created using a new type i.e. struct employee.
- Using the above array of structure, 10 set of employee records can be stored and manipulated.

Accessing Elements from Array of Structure

- To access any structure, index is used.
- For example, to read the emp_id of the first structure we use

```
scanf("%d", emp[0].emp_id);
```

- In C, array indexing starts from 0.
- Similar array of structure can also be declared as follows:

```
struct employee
{
    int emp_id;
    char name[20];
    char dept[20];
    float salary;
```

```
}emp[10];
```

- The **typedef** can be used to create similar array of structure as:

```
typedef struct
{
    int emp_id;
    char name[20];
    char dept[20];
    float salary;
}employee;
employee emp[10];
```

Example:

C program to read records of three different students in structure having member name, roll and marks, and displaying it.

```
#include<stdio.h>
/* Declaration of structure */
struct student
{
    char name[30];
    int roll;
    float marks;
};
int main()
{
    /* Declaration of array of structure */
    struct student s[3];
    int i;
    for(i=0;i<3;i++)
    {
        printf("Enter name, roll and marks of student:\n");
```

```
scanf("%s%d%f",s[i].name, &s[i].roll, &s[i].marks);  
}  
printf("Inputted details are:\n");  
for(i=0;i<3;i++)  
{  
    printf("Name: %s\n",s[i].name);  
    printf("Roll: %d\n", s[i].roll);  
    printf("Marks: %0.2f\n\n", s[i].marks);  
}  
return 0;  
}
```

OUTPUT:

Enter name, roll and marks of student:

Gopinath

17

80.5

Enter name, roll and marks of student:

Mohan

18

90.00

Enter name, roll and marks of student:

Manjula

19

78.00

Inputted details are:

Name: Gopinath

Roll: 17

Marks: 80.50

Name: Mohan

Roll: 18

Marks: 90.00

Name: Manjula

Roll: 19

Marks: 78.00

5.5 Passing structures to functions

Structures can be passed as arguments to the functions. This can be done in three ways. They are,

- Passing the members of the structures as an argument.
- Passing the entire structure as an argument.
- Passing the address of the structure as arguments.

Pass Structure Members (Variables) to Functions

- Sometimes we don't want to pass the entire structure to the function.
- We want to pass only a few members of the structure.
- We can use the dot (.) operator to access the individual members of the structure and pass them to the function.

Example:

- Let us create a structure to hold the details of a student, such as the name of the student, roll number, and marks.
- We want to print only the roll number and marks using a function.
- In this case, passing the entire structure to the function is unnecessary when we want to print only a few structure members.

// C Program to pass structure members to functions

```
#include <stdio.h>
struct student
{
    char name[50];
    int per, rno; // declare percentage and roll number as
```

```

        // integer data type
};
void display(int a, int b);
int main()
{
    struct student s1;
    printf("Enter name: ");
    scanf("%s",&s1.name);
    printf("Enter the roll number: ");
    scanf("%d",&s1.rno);
    printf("Enter percentage: ");
    scanf("%d", &s1.per);
    display(s1.rno,s1.per);
    return 0;
}
void display(int a, int b)
{
    printf("\nDisplaying information\n");
    printf("Roll number: %d", a);
    printf("\nPercentage: %d", b);
}

```

- ✓ In the above example, we created a structure to hold the name, roll number, and percentage of the student.
- ✓ The input from the user is stored in the structure.
- ✓ A function named display() is created, which takes the roll number and the percentage of the student as the parameter.
- ✓ Using the dot (.) operator, we accessed the member of the structure and passed it to the function.

OUTPUT:

The output of the above code is as follows:

Enter name: Shankar

Enter the roll number: 42

Enter percentage: 98

Displaying information

Roll number: 42

Percentage: 98

Pass Structure by Reference (Address)

- ✓ Passing the parameter as a value will make a copy of the structure variable, passing it to the function.
- ✓ Imagine we have a structure with a huge number of structure members.
- ✓ Making a copy of all the members and passing it to the function takes a lot of time and consumes a lot of memory.
- ✓ To overcome this problem, we can pass the address of the structure.
- ✓ Pointers are the variables that hold the address of other variables.
- ✓ We can use pointers to pass the structure by reference.

Example:

// C program to pass the structure by reference

```
#include<stdio.h>

struct car
{
    char name[20];
    int seat;
    char fuel[10];
};

void print_struct(struct car *);
```

```

int main()
{
    struct car tata;
    printf("Enter the model name : ");
    scanf("%s", tata.name);
    printf("\nEnter the seating capacity : ");
    scanf("%d", &tata.seat);
    printf("\nEnter the fuel type : ");
    scanf("%s", tata.fuel);
    print_struct(&tata);
    return 0;
}

void print_struct(struct car *ptr)
{
    printf("\n---Details---\n");
    printf("Name: %s\n", ptr->name);
    printf("Seat: %d\n", ptr->seat);
    printf("Fuel type: %s\n", ptr->fuel);
    printf("\n");
}

```

Explanation:

- In the above code, a structure named car and a function named print_struct() are defined.
- The structure stores the model name, seating capacity, and the fuel type of the vehicle.
- In the main() function, we created a structure variable named tata and stored the values.
- Later the address of the structure is passed into the print_struct() function, which prints the details entered by the user.
- The address is passed using the address operator ampersand (&).
- To access the pointer members, we use the arrow operator -> operator.

OUTPUT:

The output of the above code is as follows:

Enter the model name : ALtroz

Enter the seating capacity : 5

Enter the fuel type : Petrol

---Details---

Name: ALtroz

Seat: 5

Fuel type: Petrol

5.6 typedef in C

- The **typedef** is a keyword that is used to provide existing data types with a new name.
- The C typedef keyword is used to redefine the name of already existing data types.
- When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.

Syntax of typedef

typedef <existing_name> <alias_name>

- In the above syntax, '**existing_name**' is the name of an already existing variable while '**alias name**' is another name given to the existing variable.
- For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type.
- To overcome the problem, we use a **typedef** keyword.
- **typedef unsigned int unit;**
- In the above statements, we have declared the **unit** variable of type unsigned int by using a **typedef** keyword.
- we can create the variables of type **unsigned int** by writing the following statement:
- **unit a, b;** instead of writing
 - **unsigned int a, b;**
- the **typedef** keyword provides a shortcut by providing an alternative name for an already existing variable.

- This keyword is useful when we are dealing with the long data type especially, structure declarations.

Example 1 (typedef):

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
j=20;
printf("Value of i is: %d",i);
printf("\nValue of j is: %d",j);
return 0;
}
```

OUTPUT:

Value of i is :10

Value of j is :20

Example 2: (typedef using structures)

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

- In the above structure declaration, we have created the variable of student type by writing the following statement:
 - struct student s1;
- The above statement shows the creation of a variable, i.e., s1, but the statement is quite big.

- To avoid such a big statement, we use the typedef keyword to create the variable of type student.

```
struct student
{
    char name[20];
    int age;
};
typedef struct student stud;
stud s1, s2;
```

- In the above statement, we have declared the variable stud of type struct student.
- Now, we can use the stud variable in a program to create the variables of type struct student.
- The above typedef can be written as:
 - typedef struct student
 - {
 - char name[20];
 - int age;
 - } stud;
 - stud s1,s2;
- The above declarations show that the typedef keyword reduces the length of the code and complexity of data types.

5.7 Enum (Enumerated Data type) in C

- The enum in C is also known as the enumerated type.
- It is a user-defined data type that consists of integer values, and it provides meaningful names to these values.
- The use of enum in C makes the program easy to understand and maintain.
- The enum is defined by using the enum keyword.

Defining the enum:

```
enum flag{const1, const2,.....constN};
```

- In the above declaration, we define the enum named as flag containing 'N' integer constants.
- The default value of integer_const1 is 0, integer_const2 is 1, and so on.
- We can also change the default value of the integer constants at the time of the declaration.

Example:

```
enum fruits{mango, apple, strawberry, papaya};
```

- The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3.
- The default value can be changed by assigning different values as follows:

```
enum fruits
{
mango=2,
apple=1,
strawberry=5,
papaya=7,
};
```

Declaration of enum data type

```
enum boolean {false, true};
```

```
enum boolean check; // declaring an enum variable
```

- A variable **check** of the type **enum boolean** is created.
- The enum variables can also be declared as follows:
 - **enum boolean {false, true} check;**
- The value of false is equal to 0 and the value of true is equal to 1.

Example: (Enumeration type)

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

```
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

```

int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Wednesday;
    printf("Day %d",today+2);
    return 0;
}

```

OUTPUT:

Day 5

5.8 Bit Fields in C

- In C, we can specify the size (in bits) of the structure and union members.
- The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.
- Bit fields in C language are used when the storage of our program is limited.

Need of Bit Fields in C

- Reduces memory consumption.
- To make our program more efficient and flexible.
- Easy to Implement.

Declaration of C Bit Fields

Bit-fields are variables that are defined using a predefined width or size.

Syntax of Bit Fields in C

```

struct
{
    data_type member_name : width_of_bit-field;
};

```

where,

- **data_type:** It is an integer type that determines the bit-field value which is to be interpreted. The type may be int, signed int, or unsigned int.
- **member_name:** The member name is the name of the bit field.
- **width_of_bit-field:** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Example: (Bitfields)

Consider the following example code.

```
struct Date
{
    unsigned int day;
    unsigned int month;
    unsigned int year;
};
```

- Here, the variable of Date structure allocates 6 bytes of memory.
- In the above example structure, the members day and month both does not requires 2 bytes of memory for each.
- Because member day stores values from 1 to 31 only which requires 5 bits of memory, and the member month stores values from 1 to 12 only which required 4 bits of memory.
- So, to save the memory, we use the bitfields.
- Consider the following structure with bitfields.

```
struct Date
{
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year;
};
```

- Here, the variable of Date structure allocates 4 bytes of memory.

Applications of C Bit Fields

1.Limited Storage: Bit-fields are a suitable choice when you have limited memory or storage resources, as they allow you to represent and manipulate data in a compact way, using fewer bits than standard data types.

2.Device Status and Information: In situations where devices need to transmit status or information encoded as multiple bits, bit-fields are an efficient means of representing this data, optimizing the use of available bits and ensuring efficient communication.

3.Encryption Routines: Encryption algorithms often work at the bit level to manipulate and transform data.

Bit-fields can be a valuable tool in encryption routines to access and manipulate specific bits within a byte, making them useful for bitwise operations and encryption-related tasks.

5.9 Command Line arguments

- ✓ In C programming language, command line arguments are the data values that are passed from command line to our program.
- ✓ When the main function of a program contains arguments, then these arguments are known as Command Line Arguments.
- ✓ The main function can be created with two methods: first with no parameters (void) and second with two parameters.
- ✓ The parameters are argc and argv, where argc is an integer and the argv is a list of command line arguments.
- ✓ argc denotes the number of arguments given, while argv[] is a pointer array pointing to each parameter passed to the program. If no argument is given, the value of argc will be 1.
- ✓ The value of argc should be non-negative.
- ✓ Using command line arguments, we can control the program execution from the outside of the program.
- ✓ Generally, all the command line arguments are handled by the main() method. Generally, the command line arguments can be understood as follows.

- ✓ When command line arguments are passed `main()` method receives them with the help of two formal parameters and they are:

- **`int argc`**
- **`char *argv[]`**

- **`int argc`** - It is an integer argument used to store the count of command line arguments are passed from the command line.
- **`char *argv[]`** - It is a character pointer array used to store the actual values of command line arguments are passed from the command line.

Example:

// C Program to illustrate command line arguments

```
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[])
{
    int i;
    if(argc == 1)
    {
        printf("Please provide command line arguments!!!");
        return 0;
    }
    else
    {
        printf("Total number of arguments are - %d and they are\n\n", argc);
        for(i=0; i<argc ; i++)
        {
            printf("%d -- %s \n", i+1, argv[i]);
        }
        return 0;
    }
}
```

5.10 C pre-processor directives

The important functions of a preprocessor are to include the header files that contain the library functions used in the program.

A preprocessor section of the program always appears at the top of the C code.

Each preprocessor statement starts with the hash (#) symbol.

List of Preprocessor Directives

To execute a preprocessor program on a certain statement, some of the preprocessor directives types are:

- **#define:** It substitutes a preprocessor using macro.
- **#include:** It helps to insert a certain header from another file.
- **#undef:** It undefines a certain preprocessor macro.
- **#ifdef:** It returns true if a certain macro is defined.
- **#ifndef:** It returns true if a certain macro is not defined.
- **#if, #elif, #else, and #endif:** It tests the program using a certain condition; these directives can also be nested.
- **#line:** It handles the line numbers on the errors and warnings. It can be used to change the line number and source files while generating output during compile time.
- **#error and #warning:** It can be used for generating errors and warnings.
 1. **#error** can be performed to stop compilation.
 2. **#warning** is performed to continue compilation with messages in the console window.
- **#region and #endregion:** To define the sections of the code to make them more understandable and readable, we can use the region using expansion and collapse features.
- **#pragma:** Issues special commands to the compiler, using a standardized method.

Example:

```
#define MAX_ARRAY_LENGTH 20
```

- This **#define** directive tells the CPP to replace the instances of MAX_ARRAY_LENGTH with 20.
 - **#include <stdio.h>**
- **#include** directive tell the compiler to get "**stdio.h**" from the System Libraries and add the text to the current source file.
 - **#include "myheader.h"**
- **#include** tells compiler to get "myheader.h" from the local directory and add the content to the current source file.
 - **#undef FILE_SIZE**
 - **#define FILE_SIZE 45**

- `#undef` directive tell the compiler to undefine existing `FILE_SIZE` and define it as 45.
 - `#ifndef MESSAGE`
 - `#define MESSAGE "You wish!"`
 - `#endif`
- `#ifndef` directive tells the compiler to define `MESSAGE` only if `MESSAGE` is not already defined.

5.11 Files in C

- File handling in C is the process in which we create, open, read, write, and close operations on a file.
- C language provides different functions such as `fopen()`, `fwrite()`, `fread()`, `fseek()`, `fprintf()` to perform input, output, and many different C file operations in our program.

Need for File handling in C

1. Reusability:

Data stored in files can be accessed, modified, and deleted as needed, providing high reusability of information.

2. Portability:

- Files can be easily transferred between different systems without data loss.
- This feature minimizes the risk of coding errors and ensures seamless operation across platforms.

3. Efficiency:

- File handling in C simplifies the process of accessing and manipulating large amounts of data.
- It allows programs to efficiently retrieve specific information from files with minimal code, saving time and reducing the likelihood of errors.

4. Storage Capacity:

- Files provide a means to store vast amounts of data without the need to hold everything in memory simultaneously.

- This capability is particularly useful for handling large datasets and prevents memory overload in programs.

5.11.1 File Modes in C

- A file can be opened in one of four modes.
- The mode determines where the file is positioned when opened, and what functions are allowed.
- After you close a file, you can reopen the file in a different mode, depending on what you are doing.
- For example, you can create a file in create mode.
- The various modes in file are:
 - **Read mode**
 - **Update mode**
 - **Create mode**
 - **Append mode**

Read mode:

- Opens a file for reading of data.
- Read mode opens a file to the beginning.
- You cannot write to a file opened in read mode.
- You can reposition a binary or stream file in read mode.

Update mode:

- This mode allows both reading and writing of data.
- Update mode opens a file to the beginning.
- You can reposition a binary or stream file in update mode.
- You can update a record by repositioning the file to the beginning of the record, then writing the new data.

Create mode:

- It opens the specified file and positions it to the beginning.
- If a file by that name does not exist, the `openfile()` statement creates the file.
- If a file by that name exists, it overwrites the existing file, except in **VMS (Virtual Machine System)**.
- VMS: 4GL (Fourth Generation Language) creates a new version of the file. It does not overwrite the file unless the file version limit is reached.
- **[Note: A fourth-generation programming language (4GL) is a high-level computer programming language that belongs to a class of languages]**
- To create a new file, open the file in create mode.
- You cannot read, position or rewind a file opened with create mode.

Append Mode:

- It allows writing data to the end of a file.
- You cannot read, position or rewind a file opened with append mode.

5.11.2 File Operations and Functions in C

1. `fopen()` - create a new file or open an existing file.
2. `fclose()` - close a file.
3. `getc()` - reads a character from a file.
4. `putc()` - writes a character to a file.
5. `fscanf()` - reads a set of data from a file.
6. `fprintf()` - writes a set of data to a file.
7. `getw()` - reads a integer from a file.
8. `putw()` - writes a integer to a file.
9. `fseek()` - set the position to desire point.
10. `ftell()` - gives current position in the file.
11. `rewind()` - set the position to the beginning point.

Opening a file

To open a file in C, the `fopen()` function is employed, specifying the filename or file path along with the desired access modes.

Syntax of `fopen()`

```
FILE *fopen(const char *file_name, const char *access_mode);
```

Parameters

- **file_name:** If the file resides in the same directory as the source file, provide its name; otherwise, specify the full path.
- **access_mode:** Specifies the operation for which the file is being opened.

Return Value

- If the file is successfully opened, it returns a file pointer to it.
- If the file fails to open, it returns `NULL`. There are many modes for opening a file:

File opening modes in C

Mode	Description	Example
R	Opens a text file in read-only mode, allowing only reading operations.	Example: <code>fopen("demo.txt", "r")</code>
W	When using the mode "w", <code>fopen()</code> initializes a text file for writing exclusively. If the file already exists, it clears its contents; otherwise, it creates a new file for writing.	Example: <code>fopen("demo.txt", "w")</code>
A	When employing the "a" mode, <code>fopen()</code> enables opening a text file in append mode. This mode permits writing data to the end of the file, preserving existing content.	Example: <code>fopen("demo.txt", "a")</code>
r+	When using the "r+" mode, <code>fopen()</code> facilitates opening a text file for both reading and writing operations. This mode grants the ability to manipulate data at any position within the file.	Example: <code>fopen("demo.txt", "r+")</code>
w+	This mode opens a text file for both reading and writing. If the file with same name already exists, it truncates the file to zero length; otherwise, it creates a new file for both reading and writing operations.	Example: <code>fopen("demo.txt", "w+")</code>

Mode	Description	Example
a+	This mode opens a text file for both reading and writing, enabling data to be appended to the end of the file without overwriting existing content.	Example: <code>fopen("demo.txt", "a+")</code>
Rb	Opens a binary file in read-only mode, allowing reading operations on binary data.	Example: <code>fopen("demo.txt", "rb")</code>
Wb	Opens a binary file in write-only mode, truncating the file to zero length if it exists or creating a new file for writing binary data.	Example: <code>fopen("demo.txt", "wb")</code>
ab	Opens a binary file in append mode, allowing binary data to be written to the end of the file without overwriting existing content.	Example: <code>fopen("demo.txt", "ab")</code>
rb+	Opens a binary file for both reading and writing operations on binary data.	Example: <code>fopen("demo.txt", "rb+")</code>
wb+	Opens a binary file for both reading and writing operations, truncating the file to zero length if it exists or creating a new file for reading and writing binary data.	Example: <code>fopen("demo.txt", "wb+")</code>
ab+	This mode opens a binary file for both reading and writing operations, allowing binary data to be appended to the end of the file without overwriting existing content.	Example: <code>fopen("demo.txt", "ab+")</code>

Example (Opening a file)

// C Program to illustrate file opening

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // Declare a file pointer variable to store the value returned by fopen
    FILE* file_ptr;
    // Open the file "example.txt" in read mode
    file_ptr = fopen("example.txt", "r");

    // Check if the file is opened successfully
    if (file_ptr == NULL)
    {
        printf("Failed to open the file. The program will now exit.");
        exit(1);
    }
}
```

```

    }
    printf("File opened successfully!");
    return 0;
}

```

Create a File in C

- The fopen() function in C not only opens existing files but also creates a new file if it doesn't already exist.
- This behavior is achieved by using specific modes that allow file creation, such as "w", "w+", "wb", "wb+", "a", "a+", "ab", and "ab+".

```
FILE *file_ptr;
```

```
file_ptr = fopen("filename.txt", "w");
```

Example of Creating a File:

// C program to to create a file using fopen() function if it doesn't exist

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
    // Declare a file pointer variable to store the value returned by fopen
```

```
    FILE* file_ptr;
```

```
    // Open or create the file "filename.txt" in write mode
```

```
    file_ptr = fopen("filename.txt", "w");
```

```
    // Check if the file is opened successfully
```

```
    if (file_ptr == NULL)
```

```
    {
```

```
        printf("Failed to create/open the file. The program will now exit.");
```

```
        exit(1);
```

```
    }
```

```
    printf("File created/opened successfully!");
```

```
    return 0;
```

```
}
```

Reading from a File

- To read data from an existing file, we will use “r” mode in file opening.
- To read the file character by character, we use `getc()`. And to read line by line, we use `fgets()`.

Function	Description
<code>fscanf()</code>	Retrieves input from a file using a formatted string and variable argument list.
<code>fgets()</code>	Obtains a complete line of text from the file.
<code>getc()</code>	Reads a single character from the file.
<code>fgetw()</code>	Reads a numerical value from the file.
<code>fread()</code>	Extracts a specified number of bytes from a binary file.

Example: (Read the file character by character)

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE * fp;
    char s;
    fp = fopen("file2.txt", "r");
    if (fp == NULL) {
        printf("\nCAN NOT OPEN FILE");
        exit(1);
    }
    do
    {
        s = getc(fp); // Read file character by character.
        printf("%c", s);
    }
```

```

}
while (s != EOF);
fclose(fp);
return 0;
}

```

In the above program, the `getc()` function is utilized to read the file character by character until it reaches the end of the file (EOF).

Write to a File

- File writing operations in C are facilitated by functions like `fprintf()` and `fputs()`.
- Additionally, C programming provides several other functions suitable for writing data to a file.

Function	Description
<code>fprintf()</code>	This function uses a formatted string and variable arguments list to print output to the file.
<code>fputs()</code>	Prints an entire line in the file along with a newline character at the end.
<code>fputc()</code>	Writes a single character into the file.
<code>fputw()</code>	Writes a number to the file.
<code>fwrite()</code>	Writes the specified number of bytes to the binary file.

Example:

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



```

int main()
{
    FILE *fptr;
    int num = 42;

    // Opening the file in write mode
    fptr = fopen("numbers.txt", "w");
    // Writing data to the file using different functions
    fprintf(fptr, "The number is: %d\n", num);
    fputs("This is a test line.\n", fptr);
    fputc('A', fptr);
    fputw(100, fptr);
    // Closing the file
    fclose(fptr);
    return 0;
}

```

OUTPUT:

The number is: 42

This is a test line.

A

Closing a File

- The `fclose()` function is utilized to close a file in C programming.
- It is essential to close a file after performing file operations in C to release system resources and ensure proper memory management.

Syntax of `fclose()`

```
fclose(file_pointer);
```

Here, `file_pointer` is a pointer to the opened file that you want to close.

Example:

```

FILE *fptr;

fptr = fopen("fileName.txt", "w");

```

// Perform file operations

```
fclose(fp);
```

After completing the necessary file operations in C, the `fclose()` function is called to close the file pointed to by `fp`.

Example: (File operations)**// C program to Create a file, write in it, and Close the file**

```
#include <stdio.h>
#include <string.h>
int main()
{
    // Declare a pointer for the file
    FILE *diaryFile;
    // Content to be written into the file
    char diaryEntry[150] = "Diary of a Programmer - "
        "Today, I learned about file handling in C, "
        "which feels like unlocking a new programming superpower.";

    // Opening the file "LearningDiary.txt" in write mode ("w")
    diaryFile = fopen("LearningDiary.txt", "w");
    // Verifying if the file was successfully opened
    if (diaryFile == NULL)
    {
        printf("LearningDiary.txt file could not be opened.\n");
    }
    else
    {
        printf("File opened successfully.\n");
        // Checking if there's content to write
        if (strlen(diaryEntry) > 0)
        {
            // Writing the diary entry to the file
```

```

    fputs(diaryEntry, diaryFile);
    fputs("\n", diaryFile); // New line at the end of the entry
}
// Closing the file to save changes
fclose(diaryFile);
printf("Diary entry successfully recorded in LearningDiary.txt\n");
printf("File closed. Diary saved.\n");
}
return 0;
}

```

OUTPUT:

File opened successfully.

Diary entry successfully recorded in LearningDiary.txt

File closed. Diary saved.

5.11.3 Text and Binary File

Files in C can be handled either as text files or binary files:

Text Files: Text files contain human-readable characters and are typically created and edited using text editors. Functions like `fscanf()`, `fprintf()`, `fgets()`, and `fputs()` are used to read from and write to text files.

Binary Files: Binary files contain data in a format that is not human-readable. They are used for storing and retrieving structured data efficiently. Functions like `fread()` and `fwrite()` are used for binary file I/O operations

Text file

- The user can create these files easily while handling files in C.
- It stores information in the form of ASCII characters internally.
- When the file is opened, the content is readable by humans.

- It can be created by any text editor with a .txt or .rtf (rich text) extension.
- Since text files are simple, they can be edited by any text editor like Microsoft Word, Notepad, Apple Text Edit, etc.

Binary file

- It stores information in the form of 0's or 1's instead of ASCII characters.
- It is saved with the .bin extension, taking less space.
- Since it is stored in a binary number system format, it is not readable by humans.
- Binary file is more secure than a text file.

Read and Write in a Binary File

Opening a Binary File

- When intending to operate on a file in binary mode, utilize the access modes "rb", "rb+", "ab", "ab+", "wb", or "wb+" with the fopen() function.
- Additionally, employ the ".bin" file extension for binary files.

Example:

```
FILE* filePointer = fopen("example.bin", "rb");
```

In this example, the file "example.bin" is opened in binary mode for reading using the "rb" access mode.

Write to a Binary File

- When writing data to a binary file, the fwrite() function proves invaluable.
- This function allows us to store information in binary form, comprising sequences of bits (0s and 1s).

Syntax of fwrite()

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *file_pointer);
```

Parameters

ptr: A reference to the memory block holding the data intended for writing.

size: The byte size of each element to be written.

nmemb: The count of elements to be written.

file_pointer: The FILE pointer associated with the output file stream.