ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY, PATIALA

(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)

The Motto of the University
(SEWA)

SKILL ENHANCEMENT **EMPLOYABILITY** **WISDOM**
ACCESSIBILITY



DIPLOMA IN SOFTWARE DEVELOPMENT AND PROGRAMMING
SEMESTER-I
Course: Computer Programming
Course Code: CP-1-02T

ADDRESS: C/28, THE LOWER MALL, PATIALA-147001
WEBSITE: www.psou.ac.in

SELF-INSTRUCTIONAL STUDY MATERIAL FOR JGND PSOU, ALL COPYRIGHTS WITH JGND PSOU, PATIALA

CP-1-02T: Computer Programming

Total Marks: 100
External Marks: 70
Internal Marks: 30
Credits: 6
Pass Percentage: 40%

Course: Computer Programming	
Course Code: CP-1-02T	
Course Outcomes (COs) After the completion of this course, the students will be able to:	
CO1	Develop the ability to analyze problems, design algorithms, and implement solutions using C/C++ programming, showcasing proficiency in algorithmic problem-solving skills.
CO2	Implement and manipulate fundamental data structures such as arrays, linked lists, stacks, queues, trees, and hash tables in C/C++, demonstrating competence in choosing and utilizing appropriate data structures for different scenarios.
CO3	Gain expertise in handling exceptions, debugging C/C++ code, and implementing error-handling strategies to create robust and reliable programs.
CO4	Understand and apply principles of multithreading and concurrency in C/C++, including synchronization mechanisms, thread communication, and concurrent programming, showcasing the ability to develop efficient and responsive applications.
CO5	Familiarize oneself with common C++ frameworks gaining an understanding of how frameworks can streamline development and improve code organization and maintainability.

Detailed Contents:

Module No.	Module Name	Module Contents
Module I	**Problem Solving with Computers**	**Problem Solving with Computers:** Evolution of C Language, Character Set in C, Tokens, Keywords, Identifier, Constants, Variables, Rules for defining Variables, Data Types in C Language: Basic data type, Derived data type and Enum data type, Operators in C: Types of Operator: Arithmetic, Relational, Logical, Comma, Conditional, Assignment, Operator Precedence and Associativity in C, Input and Output Statements, Assignment statements.
Module II	**Control Structure**	**Control Structure:** Sequential Flow Statement, Conditional Flow Statement, Decision Control statements: if, if-else, nested-if, else-if ladder. Loop control statements: While, do-while, for loop, Nested of Loops. Case Control Statements:

		Switch Statement, goto Statement, Break Statement, Continue Statement
Module III	**Arrays and Pointers in C**	**Arrays and Pointers in C**: Arrays, Characteristic of Arrays, Representation, Declaration and Initialization of an Array, Types of Arrays: one dimensional, multi-dimensional arrays. Pointer, Pointers Declaration and Initialization, Types of Pointers, Pointer Expressions and Pointer Arithmetic.
Module IV	**Functions**	**Functions:** Function in C, Function Declaration and Definition, Types of Functions, Library Vs. User-defined Functions, Function Calling Methods, Function Parameters: Actual Parameter, Formal Parameter, Parameter Passing Techniques: Call by Value and Call by Reference, Recursive Function, Pointers and Functions.
Module V	**Strings and User Defined Data Types**	**Strings:** C Strings, Difference between char array and string literal, Traversing String, Accepting string as the input, Pointers with strings, String Functions **User Defined Data types:** Structure, Structure Variables Declaration, Accessing Structure Data Members, Array of Structures, Nested of Structure, Passing structure to function, Structures Limitations, Union, Difference between Structure and Union in C.
Module VI	**Object Oriented Programming**	**Object Oriented Programming:** Need of an Object-Oriented Programming, C++ and its Applications, OOPs Concepts in C++: Class, Objects, Encapsulation, Abstraction, Polymorphism, Inheritance, Dynamic Binding and Message Passing. Access Specifiers in C++: Private, Protected and Public.

Books

1. E. Balagurusamy, "Programming in C", Tata McGraw Hill.
2. Kamthane, "Programming with ANSI and Turbo C", Pearson Education
3. Rajaraman,V, "Fundamentals of Computers", PHI
4. Kanetkar, "Let Us C", BPB Publications.
5. Herbert Schildt, "The Complete Reference C++", Tata McGraw-Hill.
6. Deiteland Deitel, "C++ How to Program", Pearson Education.
7. Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications.
8. Bjarne Strautrup, "The C++ Programming Language", Addition-Wesley Publication Co.
9. Stanley B. Lippman, Josee Lajoie, "C++ Primer", Pearson Education.
10. E. Balagurusamy, "Object Oriented Programming with C++", Tata McGraw-Hill

COURSE: COMPUTER PROGRAMING

UNIT 1: BASIC OF PROBLEM SOLVING IN C LANGUAGES

1.1 Problem Solving with Computers

1.2 Evolution of C Language

1.3 Character Set in C

1.4 Tokens

1.5 Keywords

1.6 Identifiers

1.7 Constants

1.8 Variables

1.9 Rules for defining Variables

1.10 Data Types

1.10.1 Basic data type

1.10.2 Derived data type

1.10.3 Enum data type

1.1 Problem Solving with Computers

Computer based problem solving is a systematic process of designing, implementing and using programming tools during the problem solving stage. The following six steps must be followed to solve a problem using computer.

1. **Problem analysis:** It is the process of defining a problem and decomposing overall system into smaller parts to identify possible inputs, processes and outputs associated with the problem.
2. **Program design:** The second stage in problem solving using computer cycle is program design. This stage consists of preparing algorithms, flowcharts and pseudocodes. Generally, this stage intends to make the program more user friendly, feasible and optimized.
3. **Coding:** In this stage, process of writing actual program takes place. A coded program is most popularly referred to as a source code. The coding process can be done in any language (high level and low level). The actual use of computer takes

place in this stage in which the programmer writes a sequence of instructions ready for execution. Coding is also known as programming.

4. **Compilation and Execution:** In step 3 coding is done in either high level language or low level language (assembly language). For the computer to understand these languages, they must be translated into machine level language. The translation process is carried out by a compiler/interpreter (for high level language) or an assembler (for assembly language program).
5. **Debugging and Testing:** Debugging is the process of finding errors and removing them from a computer program, otherwise they will lead to failure of the program. Debugging is generally done by program developer. Further, testing is performed to verify that whether the completed software package functions or works according to the expectations defined by the requirements. Testing is generally performed by testing team which repetitively executes program with intent to find error.
6. **Program Documentation:** The program documentation is the process of collecting information about the program. The documentation process starts from the problem analysis phase to debugging and testing.

1.1 Evolution of C Language

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A. Dennis Ritchie is known as the founder of the c language. The root of all modern languages is ALGOL (Algorithmic Language). ALGOL was the first computer programming language to use a block structure, and it was introduced in 1960. In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language). BCPL was derived from ALGOL. In 1970, Ken Thompson created a language using BCPL called B. Both BCPL and B programming languages were typeless. After that, C was developed using BCPL and B by Dennis Ritchie at the Bell lab in 1972. So, in terms of history of C language, it was used in mainly academic environments, but at long last with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain extensive support among professionals. The following table highlights the evolution of C Language.

Sr. No.	Development Year	Language Name	Developer Name

1	1960	Algol	International Group
2	1967	BCPL	Martin Richard
3	1970	B	Ken Thompson
4	1972	Traditional C	Dennis Ritchie
5	1978	K & R C	Kernighan & Dennis Ritchie
6	1989	ANSI C	ANSI Committee
7	1990	ANSI/ISO C	ISO Committee

1.3 Character Set in C

The set of characters that are used to represent words, numbers and expression in C language is called C character set. The combination of these characters form words, numbers and expression in C. The characters in C are grouped into the following four categories.

- Letters or Alphabets
- Digits
- Special Characters
- White Spaces

Type of Character	**Characters**
Lowercase Alphabets	a to z
Uppercase Alphabets	A to Z
Digits	0 to 9

Special Characters	` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ – | / \ ; : _ ─ , . ?
White Spaces	Blank Spaces, Carriage Return, Tab, New Line

1.4 Tokens

The individual elements of a program are called Tokens. In a C program, a number of individual units or elements occur and these elements are termed as C Tokens. In C programming language, the following 6 types of tokens are available:

- Keywords
- Identifiers
- Constant
- Operators
- Strings
- Special Characters

Tokens in C are a fundamental part of the programming language. They are the smallest individual units. Tokens provide functionality for users to interact easily with the compiler.

1.5 Keywords

Keywords are predefined words for a C programming language. All keywords have fixed meaning and these meanings cannot be change. They serve as basic building blocks for program statements. A list of 32 keywords in the C language is given below:

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

1.6 Identifiers

C identifier is a name used to identify a variable, function, or any other user-defined item. An

identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- Identifiers should not begin with any numerical digit.
- Identifiers are case sensitive i.e. both uppercase and lowercase letters are distinct.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be meaningful, short, and easy to read.

1.7 Constants

A constant is basically a named memory location in a program that holds a single value throughout the execution of that program. It can be of any data type- character, string, floating-point, double, and integer, etc. A constant is a value or variable that can't be changed in the program.

There are two ways to define constant in C programming.

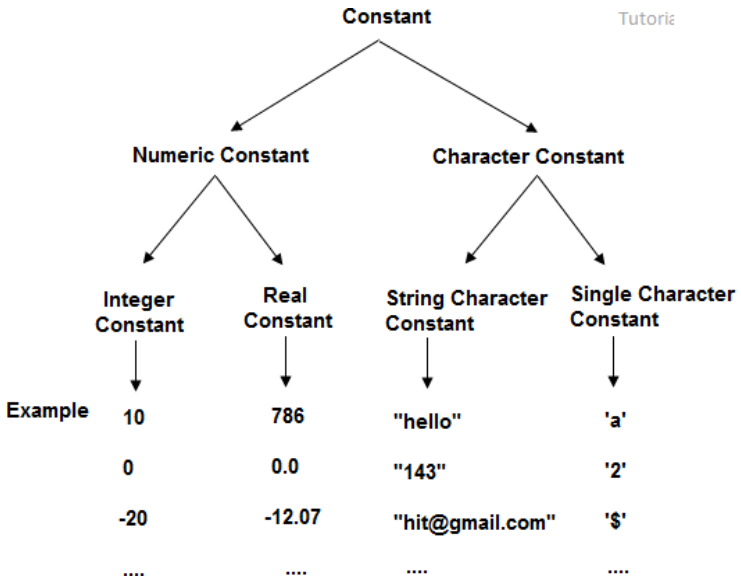
- **Using const keyword:** The _const‘ keyword is used to create a constant of any given datatype in a program. For creating a constant, there is need to prefix the declaration of the variable with the _const‘ keyword.
  Syntax: const datatype constantName = value;
  Example: const float PI=3.14**;**

- **Using #define pre-processor:** To define the constants _#define‘ pre-processor directive can also be used. It must be defined in the very beginning of the program as all the preprocessor directives must be defined before the global declaration.
  Syntax: #define identifierName value
  Example: #define PI 3.14

A constant is very similar to variables in the C programming language, but it can hold only a single variable during the execution of a program.

NOTE: Literals are the constant values assigned to the constant variables.

Types of Constant in C



- **Integer Constants:** An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:
  - Decimal Constant(base 10)
  - Octal Constant(base 8)
  - Hexadecimal Constant(base 16)
- **Floating point/Real constants:** A floating point constant is a numeric constant that has either a fractional form or an exponent form.
  - Examples of Real constants in decimal form are:

    2.2, +8.0, -4.15
  - Examples of Real constants in exponential notation are:

    +1e23, -9e2, +3e-15
- **Character Constants:** The character constants are symbols that are enclosed in one single quotation. The maximum length of a character quotation is of one character only. Example:

  _B‘

  _5‘

  _+‘

- **String Constants:** The string constants are a collection of various special symbols, digits, characters, and escape sequences that get enclosed in double quotations.

  The definition of a string constant occurs in a single line:

  ―Programming in C‖

- **Escape Sequences or Backslash Character Constants:** These are some types of characters that have a special type of meaning in the C language. These types of constants must be preceded by a backslash symbol so that the program can use the special function in them. Below mentioned is a list of all the special characters used in the C language:

Meaning of Character	**Backslash Character**
Backspace	\b
New line	\n
Form feed	\f
Horizontal tab	\t
Carriage return	\r
Single quote	\‘
Double quote	\‖
Vertical tab	\v
Backslash	\\
Question mark	\?
Alert or bell	\a
Hexadecimal constant (Here, N – hex.dcml cnst)	\XN
Octal constant (Here, N is an octal constant)	\N

1.8 Variables

A variable is a name given to the memory location that is used to store data. The value of the variable can be changed, and it can be reused several times. It is a way to represent memory location through symbol so that it can be easily identified.

Declaration of Variables: Variables are the storage areas in a code that the program can easily manipulate. Every variable in C language has some specific type- that determines the layout and the size of the memory of the variable, the range of values that the memory can hold, and the set of operations that one can perform on that variable.

***Syntax*:** type variable_list;

***Example*:** **int** a1;
float b1;
char c1;

Here, a1, b1, c1 are variables of int, float and char data types respectively.

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

***Syntax*:** type variable_name = value;

***Example*:** int a = 31, b = 15; // declaration and initialization of variables a, b

1.9 Rules for defining Variables

- A variable can have alphabets, digits, and underscore.
- No whitespace is allowed within the variable name.
- Variables should not be declared with the same name in the same scope.
- A variable name must not be any reserved word or keyword, e.g. char, float, etc.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- Maximum length of variable is 8 characters depend on compiler and operation system.

For Example

int a1; // valid declaration

int 1a; // invalid declaration – the name of the variable should not start using a number

int roll_no; // valid declaration

int roll$no // invalid declaration – no special characters allowed

char break; // keywords cannot be used as variable name.

int roll no; // invalid declaration – there must be no spaces in the name of the variable

NOTE: C is case-sensitive language. Here, MARKS, Marks and marks are three different variables.

int MARKS; // it is a new variable

int Marks; // it is a new variable

int marks; // it is a new variable

Types of Variables in C: Based on the name and the type of the variable, the variables can be of the following basic types:

- **Global Variable**: A variable that gets declared outside a block or a function is known as a global variable. Any function in a program is capable of changing the value of a global variable. Hence, the global variable will be available to all the functions in the code. Because the global variable in C is available to all the functions, it is declared at the beginning of a block.

  Example:

```
int code=30; //  global variable
void function1()
 {
   int a=20; //  local variable
 }
```

- **Local Variable:** A local variable is a type of variable that is declared inside a block or a function, unlike the global variable. Hence, it is mandatory to declare a local variable in C at the beginning of a given block.

Example:

```
void function1()
 {
int a=10; //  local variable
  }
```

A local variable has to be initialized in a code before we use it in the program.

- **Automatic Variable:** Every variable that is declared inside a block (in the C language) is by default automatic in nature. Automatic variable can be declared explicitly by using the keyword ***auto***.

Example:

```
void main(){
 int a1=810; //  local variable (implicitly automatic variable)
 auto int b1=510; // an automatic variable
 }
```

- **Static Variable:** The static variable in C language is declared using the ***static*** keyword. This variable retains the given value between various function calls.

Example:

```
void function1()
{
 int m=10; //  Local variable
static int n=10; // Static variable
m=m+1;
n=n+1;
printf(—%d,%d‖,m,n);
}
```

Output:

Local variable m will print the value 11 for every function call.
Static variable n will print the value that is incremented in each and every function call i.e. 11, 12, 13 and so on

- **External Variable:** A user can share a variable in multiple numbers of source files in C by using an external variable. The keyword ***extern*** *is used to declare* an external variable.

  Syntax:

  extern int a=10;// external variable (also a global variable)

1.10 Data Types in C Language

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Types of Data Types in C

Data Type	Example of Data Type
Primary/ Basic Data Type	Floating-point, integer, double, character, void
Derived Data Type	Union, structure, array, etc.
Enumerated Data Type	Enums

1.10.1 Basic/Primary Data Types: There are the primitive or primary data types in C programming language:

- **Integer:** Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, 0, -5, 10.

```
// C program to print Integer data types
#include <stdio.h>
int main()
{
// Integer variable with positive data.
int a1 = 19;

// integer variable with negative data.
int b1 = -91;

printf("Integer variable with positive data: %d\n", a1);
```

```
printf("Integer variable with negative data: %d\n", b1);

return 0;
}
```

Output:

Integer variable with positive data: 19

Integer variable with negative data: -91

- **Character:** This data type is used to store only a single character. The storage size of the character is 1. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

```
// C program to print Character data types.
#include <stdio.h>
 int main()
  {
   char ch = 'a';
   char c;
   printf("Value of ch: %c\n", ch);
   ch++;
    printf("Value of ch after increment is: %c\n", ch);

    // c is assigned ASCII values
    // which corresponds to the
    // character 'c'
    // a-->97 b-->98 c-->99
    // here c will be printed
    c = 99;
    printf("Value of c: %c", c);
    return 0;
}
```

Output:

Value of ch: a

Value of ch after increment is: b

Value of c: c

- **Floating-point:** In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

```
// C program to print Float data types
#include <stdio.h>
int main()
{
   float a1 = 5.0f;
   float b1 = 12.5f;
     // 2x10^-4
   float c1 = 2E-4f;
     printf("%f\n",a1);
     printf("%f\n",b1);
     printf("%f",c1);
  return 0;
}
```

Output:

5.000000

12.500000

0.000200

- **Double:** A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C.

```
// C program to print double data types
#include <stdio.h>
 int main()
{
   double a1 = 23125623.00;
```

```
    double b1 = 2.267823;
    double c1 = 2312312312.123123;
    printf("%lf\n", a1);
    printf("%lf\n", b1);
    printf("%lf", c1);
    return 0;
}
```

Output:

```
23125623.00

2.267823

2312312312.123123
```

- **Void:** The void data type in C is used to specify that no value is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

```
// function return type void
void exit(int check);


// Function without any parameter can accept void.
int print(void);
```

Range of Values of Basic Data Types in C

Data Type	**Format Specifier**	**Minimal Range**	**Memory Size (in bits)**
char	%c	-127 to 127	8
int	%d	-32,767 to 32,767	16 or 32
float	%f	1E-37 to 1E+37 along with six	32

		digits of the precisions here	
double	%lf	1E-37 to 1E+37 along with six digits of the precisions here	64

1.10.2 Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc. These are the data type whose variable can hold more than one value of similar type. In C language it can be achieve by array.

```
int m[] = {10,20,30};  // valid
int n[] = {100, 'A', "ABC"};   //  invalid
```

The C language supports a few derived data types. These are:

- **Arrays –** The array basically refers to a sequence (ordered sequence) of a finite number of data items from the same data type sharing one common name.
- **Function –** A Function in C language refers to a self-contained block of single or multiple statements. It has its own specified name.
- **Pointers –** The Pointers in C language refer to some special form of variables that one can use for holding other variables‘ addresses.
- **Unions –** Unions data types are very similar to the structures. It is used to store objects of various different types in the very same location of the memory. It means that in any program, various different types of union members would be able to occupy the very same location at different times.
- **Structures –** A collection of various different types of data type items that get stored in a contagious type of memory allocation is known as structure in C language.

```
struct student
{
int roll_no;
char name[15];
float marks;
}
```

1.10.3 Enum Data Types

Enumeration is a user defined data type in C language. It is mainly used to assign names to

integral constants, the names make a program easy to read and maintain. The keyword _enum‘ is used to declare new enumeration types in C language.

Syntax: enum flag {int_const1, int_const2,. int_constN};

In the above declaration, enum named as flag is defined containing 'N' integer constants. The default value of int_const1 is 0, int_const2 is 1, and so on. The default value of the integer constants can be changed at the time of the declaration.

// Program to demonstrate working of enum data type

```
#include<stdio.h>
 enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
 int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Output:

2

COURSE: COMPUTER PROGRAMING

UNIT 2: OPERATORS AND EXPRESSION IN C

2.1 Operators in C

2.2 Types of Operator

2.2.1 Arithmetic Expressions

2.2.2 Relational Operator

2.2.3 Logical Operator

2.2.4 Comma Operator

2.2.5 Conditional Operator

2.2.6 Assignment Operator

2.3 Operator Precedence and Associativity in C

2.3.1 Operator Precedence

2.3.2 Operator Associativity

2.4 Input and Output Statements

2.5 Assignment Statements

2.1 Operators in C:

The operators are simply a symbol that can be used to perform operations. In simpler words, we can also say that an operator is a type of symbols that inform a compiler to perform specific mathematical, conditional, or logical functions. For example, + and - are the operators to perform addition and subtraction in any C program. C has many operators that almost perform all types of operations. These operators are really useful and can be used to perform every operation. Basically, operators serve as the foundations of the programming languages. Thus, the overall functionalities of the C programming language remain incomplete if we do not use operators.

2.2 Types of Operators:

Various types of operators are available in the C language that performs different types of operations.

I. Arithmetic Operators
II. Relational Operators
III. Logical Operators

IV. Comma Operators

V. Conditional Operators

VI. Assignment Operators

2.2.1 Arithmetic Operators:

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables). We can also say that, it helps a user to perform the mathematical operations as well as the arithmetic operations in a program, such as subtraction (-), addition (+), division (/), multiplication (*), the remainder of division (%), decrement (–), increment (++).

Further, arithmetic operators are divided into two types:

Unary Operators: Operators that operate or work with a single operand are unary operators. For example: Increment(++) and Decrement(–) Operators.

Let‘s look at an example for demonstrating the working of increment and decrement operator:

```
// Examples of increment and decrement operators:
#include <stdio.h>
int main()
{
   int a = 11, b = 90;
   float c = 100.5, d = 10.5;
   printf("++a = %d \n", ++a);
   printf("--b = %d \n", --b);
   printf("++c = %f \n", ++c);
   printf("--d = %f \n", --d);
   return 0;
}
```

Output:

```
++a = 12
--b = 89
++c = 101.500000
--d = 9.500000
```

In the above code example, the increment and decrement operators ++ and -- have been used as prefixes. Note that these two operators can also be used as postfixes like a++ and a-- when required.

Binary Operators: Operators that operate or work with two operands are binary operators. For example: Addition(+), Subtraction(-), multiplication(*), Division(/) operators. Let‘s look at an example of binary Arithmetic operations in C below assuming variable a holds 7 and variable b holds 5.

```
// Examples of arithmetic operators in C
#include <stdio.h>
int main()
{
    int a = 7,b = 5, c;
     c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a is divided by b = %d \n",c);
    return 0;
}
```

Output:

```
a+b = 12
a-b = 2
a*b = 35
a/b = 1
Remainder when a divided by b = 2
```

2.2.2 Relational Operators:

Rational Operators are used for the comparison of the values of two operands. For example, checking if one operand is equal to the other operand or not, whether an operand is greater than the other operand or not, etc. Some of the relational operators are (==, >= , <=)(See this article for more reference). In other words, relational operators are specifically used to compare two quantities or values in a program. It checks the relationship between two operands. If the given relation is true, it will return 1 and if the relation is false, then it will return 0. Relational operators are heavily used in decision-making and performing loop operations.

The table below shows all the relational operators supported by C.

Operator	**What it does**	**Example**
==	Equal to	5==5 will be 1
>	Greater than	5>6 will be 0
<	Less than	6<7 will be 1
>=	Greater than equal to	2 >= 1 will be 1
<=	Less than equal to	1 <= 2 will be 1
!=	Not equal to	5 != 6 will be 1

Below is an example showing the working of the relational operator:

// Example of relational operators

```
#include <stdio.h>
int main()
{
int x = 8, y = 10;
printf("%d == %d is False(%d) \n", x, y, x == y);
printf("%d != %d is True(%d) \n ", x, y, x != y);
printf("%d > %d is False(%d)\n ",  x, y, x > y);
printf("%d < %d is True (%d) \n", x, y, x < y);
printf("%d >= %d is False(%d) \n", x, y, x >= y);
printf("%d <= %d is True(%d) \n", x, y, x <= y);
```

```
return 0;
}
```

Output:

```
8 == 10 is False(0)
8 != 10 is True(1)
8 > 10 is False(0)
8 < 10 is True(1)
8 >= 10 is False(0)
8 <=10 is True(1)
```

All the relational operators work in the same manner as described in the table above.

2.2.3 Logical Operators:

In the C programming language, we have three logical operators when we need to test more than one condition to make decisions. These logical operators are:

&& (meaning logical AND)

|| (meaning logical OR)

! (meaning logical NOT)

An expression containing a logical operator in C language returns either 0 or 1 depending upon the condition whether the expression results in true or false. Logical operators are generally used for decision-making in C programming.

The table below shows all the logical operators supported by the C programming language.

Operator	**What it does**
&& (Logical AND)	True only if all conditions satisfy.
|| (Logical OR)	True only if either one condition satisfies.
! (Logical Not)	True only if the operand is 0.

Following is the example that easily elaborates the working of the logical operator:-

```
#include <stdio.h>
int main()
{
  int i = 5, j = 5, k = 10, final;
  printf("i is equal to j or k greater than j is is %d \n", (i == j) && (k > j));
  printf("i is equal to j or k less than j is %d \n", (i == j) || (k < j));
  printf("i not equal to j or k less than j is %d \n", (i != j) || (k < j));
   return 0;
}
```

Output:

```
i is equal to j or k greater than j is 1
i is equal to j or k less than j is 1
i not equal to j or k less than j is 0
```

2.2.4 Comma Operators:

Comma Operators are used for separating expressions, variable declarations, function calls etc. It works on two operands. It is a binary operator. Comma acts as a separator.

Syntax of comma operator:-

```
int a=1, b=2, c=3, d=4;
```

2.2.5 Conditional Operators:

Conditional or ternary operator is used to construct the conditional expression. A conditional operator pair "?:"

Syntax of Conditional Operators:

```
exp1 ? exp2 : exp3
```

Here exp1, exp2, exp3 are expressions.

The Operator ?: works as follows: exp1 is evaluated first. If it is true, then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, then exp3 is evaluated and its value becomes the value of the expression.

Example of Conditional Operator:

```
#include <stdio.h>
int main()
{
int number=13;
(number>14)? (printf("It is greater than number 14!")) : (printf("It is less than
number 14!"));  // conditional operator
return 0;
}
```

Output:

It is less than number 14!

If we set the number to 15 then it will give the output⇒ It is greater than number 14!

2.2.6 Assignment Operators:

An assignment operator is mainly responsible for assigning a value to a variable in a program. Assignment operators are applied to assign the result of an expression to a variable. This operator plays a crucial role in assigning the values to any variable. The most common assignment operator is =. C language has a collection of shorthand assignment operators that can be used for C programming. The table below lists all the assignment operators supported by the C language:

Operator	**Example**
=	a=b or b=a
+=	a += b or a = a+b
-=	a -=b or a = a-b
*=	a *= b or a = a*b
/=	a /= b or a = a/b
%=	a %= b or a = a%b

The below example explains the working of assignment operator:

```
#include <stdio.h>
int main()
{
  int a = 99, result;
  result = a;
  printf("Welcome to TechVidvan Tutorials...\n");
  printf("Value of result = %d\n", result);
  result += a;    // or result = result + a
  printf("Value of result = %d\n", result); // After Addition
  result -= a;    // or result = result - a
  printf("Value of result = %d\n", result); // After Subtraction
  result *= a;    // or result = result * a
  printf("Value of result = %d\n", result); // After Multiplication
  result /= a;    // or result = result / a
  printf("Value of result = %d\n", result);
  return 0;
}
```

Output:

```
Welcome to TechVidvan Tutorials…
Value of result = 99
Value of result = 198
Value of result = 99
Value of result = 9801
Value of result = 99
```

Arithmetic Expressions:

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression computes a value of type int, float or double. When an expression contains only integral operands, then it is known as pure integer expression when it contains only real operands, it is known as pure real expression, and when it contains both integral and real operands, it is known as mixed mode expression.

Let's understand through an example.

6*2/ (2+1 * 2/3 + 6) + 8 * (8/4)

Evaluation of expression	Description of each operation
6*2/(2+1 * 2/3 +6) +8 * (8/4)	An expression is given.
6*2/(2+2/3 + 6) + 8 * (8/4)	2 is multiplied by 1, giving value 2.
6*2/(2+0+6) + 8 * (8/4)	2 is divided by 3, giving value 0.
6*2/ 8+ 8 * (8/4)	2 is added to 6, giving value 8.
6*2/8 + 8 * 2	8 is divided by 4, giving value 2.
12/8 +8 * 2	6 is multiplied by 2, giving value 12.
1 + 8 * 2	12 is divided by 8, giving value 1.
1 + 16	8 is multiplied by 2, giving value 16.
17	1 is added to 16, giving value 17.

2.3 Operator Precedence and Associativity

2.3.1 Operator Precedence

Operator Precedence in C is used to determine the sequence in which different operators will be evaluated if two or more operators are present in an expression. The associativity of operators is used to determine whether an expression will be evaluated from left-to-right or from right-to-left if there are two or more operators of the same precedence.

Operator precedence controls how terms in an expression are grouped and how an expression is evaluated. Certain operators take precedence over others. The multiplication operator, for example, takes priority over the addition operator.

For example, x = 2 + 3 * 5;

Here, the value of x will be assigned as 17 and not 20. The ―*‖operator has higher precedence than the ―+‖ operator. So the first 3 is multiplied by 5 to get 15, and then 2 is added to 15 to result in 17.

2.3.2 Operator Associativity

The direction in which an expression is evaluated is determined by the associativity of

operators. Associativity is utilized when two operators of the same precedence exist in an expression. Associativity can be either left to right or right to left.

For example, consider x = 5 / 3 * 3;

Here, the value of x will be assigned as 3 and not 5. _*‘ operator and _/‘ operator have the same precedence, but their associativity is from Left to Right. So first 5 is divided by 3 to get 1, and then 1 is multiplied by 3, resulting in 3.

The following table explain the order of Precedence and Associativity in C Language.

Operator	**Order of Precedence**	**Associativity**
.	Direct member selection	Left to right
->	Indirect member selection	Left to right
[]	Array element reference	Left to right
()	Functional call	Left to right
~	Bitwise(1‘s) complement	Right to left
!	Logical negation	Right to left
–	Unary minus	Right to left
+	Unary plus	Right to left
—	Decrement	Right to left
++	Increment	Right to left
*	Pointer reference	Right to left
&	Dereference (Address)	Right to left
(type)	Typecast (conversion)	Right to left
sizeof	Returns the size of an object	Right to left

%	Remainder	Left to right
/	Divide	Left to right
*	Multiply	Left to right
–	Binary minus (subtraction)	Left to right
+	Binary plus (Addition)	Left to right
>>	Right shift	Left to right
<<	Left shift	Left to right
>	Greater than	Left to right
<	Less than	Left to right
>=	Greater than or equal	Left to right
<=	Less than or equal	Left to right
==	Equal to	Left to right
!=	Not equal to	Left to right
^	Bitwise exclusive OR	Left to right
&	Bitwise AND	Left to right
||	Logical OR	Left to right
|	Bitwise OR	Left to right
?:	Conditional Operator	Right to left
&&	Logical AND	Left to right

,	Separator of expressions	Left to right
=	Simple assignment	Right to left
/=	Assign quotient	Right to left
*=	Assign product	Right to left
%=	Assign remainder	Right to left
-=	Assign difference	Right to left
+=	Assign sum	Right to left
|=	Assign bitwise OR	Right to left
^=	Assign bitwise XOR	Right to left
&=	Assign bitwise AND	Right to left
>>=	Assign right shift	Right to left
<<=	Assign left shift	Right to left

2.4 Input and Output Statements:

Input and Output statement are used to read and write the data in C programming. These are embedded in stdio.h (standard Input/Output header file). Input means to provide the program with some data to be used in the program and Output means to display data on screen or write the data to a printer or a file.C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.There are mainly two of Input/Output functions are used for this purpose. These are discussed as:

3.1 Unformatted I/O functions

3.2. Formatted I/O functions

2.4.1. Unformatted I/O functions: There are mainly six unformatted I/O functions discussed

as follows:

a) getchar()
b) putchar()
c) gets()
d) puts()
e) getch()
f) getche()
g) getchar()

This function is an Input function. It is used for reading a single character from the keyboard. It is a buffered function. Buffered functions get the input from the keyboard and store it in the memory buffer temporally until you press the Enter key.

The general syntax is as: v = getchar();

where v is the variable of character type.

For example:

```
char n;
n = getchar();
```

A simple C-program to read a single character from the keyboard is as:

```
/*To read a single character from the keyboard using the getchar() function*/
#include <stdio.h>
main()
{
char n;
n = getchar();
}
```

- **putchar():** This function is an output function. It is used to display a single character on the screen. The general syntax is as: putchar(v);

where v is the variable of character type, For example:

```
char n;
```

putchar(n); A simple program is written as below, which will read a single character using getchar() function and display inputted data using putchar() function:

```
/*Program illustrate the use of getchar() and putchar() functions*/
#include <stdio.h>
main()
{
char n;
n = getchar();
putchar(n);
}
```

- **gets()**: This function is an input function. It is used to read a string from the keyboard. It is also a buffered function. It will read a string when you type the string from the keyboard and press the Enter key from the keyboard. It will mark null character (‗\0‘) in the memory at the end of the string when you press the enter key. The general syntax is as:

  ```
  gets(v);
  ```
  where v is the variable of character type. For example:
  ```
  char n[20];
  gets(n);
  ```
  A simple C program to illustrate the use of gets() function:

  ```
  /*Program to explain the use of gets() function*/
  #include <stdio.h>
  main()
  {
  char n[20];
  gets(n);
  }
  ```
  puts()

This is an output function. It is used to display a string inputted by gets() function. It is also used to display a text (message) on the screen for program simplicity. This function appends a newline (―\n‖) character to the output.

The general syntax is as:

```
puts(v);
or
puts("text line");
```

where v is the variable of character type.

A simple C program to illustrate the use of puts() function:

```
/*Program to illustrate the concept of puts() with gets() functions*/
#include <stdio.h>
main()
{
char name[20];
puts("Enter the Name");
gets(name);
puts("Name is :");
puts(name);
}
```

The Output is as follows:

```
Enter the Name      Geek
Name is:            Geek
```

- **getch():** This is also an input function. This is used to read a single character from the keyboard like getchar() function. But getchar() function is a buffered is function, getchar() function is a non-buffered function. The character data read by this function is directly assigned to a variable rather it goes to the memory buffer, the character data is directly assigned to a variable without the need to press the Enter key.

Another use of this function is to maintain the output on the screen till you have not press the Enter Key. The general syntax is as:

```
v = getch();
```

where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getch();
puts("Char is :");
putchar(n);
getch();
}
```

The output is as follows:

```
Enter the Char
Char is L
getche()
```

All are same as getch(0 function execpt it is an echoed function. It means when you type the character data from the keyboard it will visible on the screen. The general syntax is as:

```
v = getche();
```

where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getche();
puts("Char is :");
putchar(n);
getche();
}
```

The output is as follows:

Enter the Char L

Char is L

Formatted I/O functions

Formatted I/O functions which refers to an Input or Ouput data that has been arranged in a particular format. There are mainly two formatted I/O functions discussed as follows:

scanf()

printf()

scanf()

The scanf() function is an input function. It used to read the mixed type of data from keyboard. You can read integer, float and character data by using its control codes or format codes. The general syntax is as:

scanf("control strings",arg1,arg2,.....argn);

or

scanf("control strings",&v1,&v2,&v3,.....&vn);

Where arg1,arg2,…….argn are the arguments for reading and v1,v2,v3,.....vn all are the variables.

The scanf() format code (spedifier) is as shown in the below table:

Example Program:

```
/*Program to illustrate the use of formatted code by using the formatted scanf() function */
#include <stdio.h>
main()
{
char n,name[20];
int abc;
float xyz;
printf("Enter the single character, name, integer data and real value");
scanf("\n%c%s%d%f", &n,name,&abc,&xyz);
getch();
```

}

printf()

This ia an output function. It is used to display a text message and to display the mixed type (int, float, char) of data on screen. The general syntax is as:

printf("control strings",&v1,&v2,&v3,. ..... &vn);

or

printf("Message line or text line");

Where v1,v2,v3,… ..... vn all are the variables.

The control strings use some printf() format codes or format specifiers or conversion characters.

Example Program:

```
/*Below the program which show the use of printf() function*/
#include <stdio.h>
main()
{
int a;
float b;
char c;
printf("Enter the mixed type of data");
scanf("%d",%f,%c",&a,&b,&c);
getch();
}
```

2.5 Assignment Statements:

An Assignment statement is a statement that is used to set a value to the variable name in a program. C provides an assignment operator for this purpose, assigning the value to a variable using assignment operator is known as an assignment statement in C. The function of this operator is to assign the values or values in variables on right hand side of an expression to variables on the left hand side.

The syntax of the assignment expression

Variable = constant / variable/ expression;

The data type of the variable on left hand side should match the data type of constant/variable/expression on right hand side with a few exceptions where automatic type

conversions are possible.

Examples of assignment statements:

```
b = c ; /* b is assigned the value of c */
a = 9 ; /* a is assigned the value 9*/
b = c+5; /* b is assigned the value of expr c+5 */
\
```

COURSE: COMPUTER PROGRAMING

UNIT 3: CONTROL STRUCTURE IN C

3.1 Control Structure

3.1.1 Sequential Flow Statement

3.1.2 Conditional Flow Statement

3.2 Decision Control Statements

3.2.1 If Statement

3.2.2 If-else statement

3.2.3 Nested-if Statement

3.3 Loop Control Statements

3.3.1 While Statement

3.3.2 do-while Statement

3.3.3 for loop Statement

3.3.4 Nested Loop Statement

3.4 Case Control Statements

3.4.1 Switch Statement

3.4.2 goto statement

3.4.3 Break Statement

3.4.4 Continue Statement

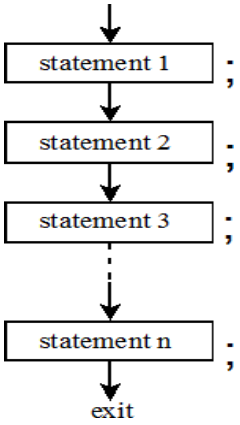
3.1 Control Structure:

Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be clearer and understood if they use self-contained modules called as logic or control structures. It basically analyses and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of flow of control known as Sequential flow, Conditional flow, Iteration flow. The sequential flow is un-conditional flow of control, but the next two are types of conditional statements. The description about each are given below:

3.1.1 Sequential Flow Statement

Sequential flow as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are

given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern. The following figure highlights the flow of sequential flow statements.



The Process start with statement 1 follow with statement 2, next follow with statement 3 and same follow till statement n in sequential manner. Therefore when the program which flows only from top to bottom without changing the flow of control are known as sequential control statements.

3.1.2 Conditional Flow Statement

Sometimes, it is desirable to alter the sequence of the statements in the program depending upon certain circumstances. Repeat a group of statements until certain specified conditions are met. This involves a kind of decision making to see whether a particular condition has occurred or not and direct the computer to execute certain statements accordingly. Based on application and the specific requirement, it is necessary to:

(i) To alter the flow of a program

(ii) Test the logical conditions

(iii) Control the flow of execution as per the selection these conditions can be placed in the program using decision-making statements.

In simple words, Control statements in C help the system to execute a certain logical statement and decide whether to enable the control of the flow through a certain set of statements or not. Based on the conditions and flow of execution, control statement is classified into three categories named as:

- Decision Control Statements

- Loop Control Statements
- Case Control Statements

3.2 Decision Control statements:

There come situations when we need to make some decisions, on which we can decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. In conditional control, the execution of statements depends upon the condition-test. If the condition evaluates to true, then a set of statements is executed otherwise another set of statements is followed. This control is also called Decision Control because it helps in making decision about which set of statements is to be executed. In C language, if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions. The Decision Control Statements are used to evaluate the one or more conditions and make the decision whether to execute set of statement or not. Based on the hierarchy of conditions, the Decision Control Statements has five types of control statements:

- if Statement
- if-else Statement
- Nested if-else statement
- else-if Ladder

The detail about each statement is given below:

3.2.1 **if statement**:

Simple if statements are carried out to perform some operation when the condition is only true. If the condition of the if statement is true then the statements under the block is executed else the control is transferred to the statements outside the block directly and none of the statements will be executed. It is also called a one-way selection statement.

Syntax:

```
If (expression)
{
        //code to be executed
```

}

Following program illustrates the use of if construct in C-Language.

```
#include<stdio.h>
int main()
{
    int num1=1;
    int num2=2;
    if(num1<num2)                    //test-condition
    {
        printf("num1 is smaller than num2");
    }
    return 0;
}
```

Output: ―num1 is smaller than num2‖

In the above program, we have initialized two variables with num1, num2 with value as 1, 2 respectively. Then, we have used if with a test-expression to check which number is the smallest and which number is the largest. We have used a relational expression in if construct. Since the value of num1 is smaller than num2, the condition will evaluate to true. Thus it will print the statement inside the block of If. After that, the control will go outside of the block and program will be terminated with a successful result.

3.2.2 If-else statement:

The single if statement may work pretty well, but in some situations, user may have to execute statements based on true or false under certain conditions and user may want to work with multiple variables or the extended conditional parameters, then the if-else statement is the optimum choice, by using the if statement, only one block of the code executes after the condition is true but by using the if-else statement, there are two possible blocks of code where the first block is used for handling the success part and the other one for the failure condition. It is also called two way selection statement.

Syntax:

```
if(expression)
```

```
{
 //Statements
}
else
{
 //Statements
}
```

As discussed above, the if statement is applicable to make one comparison, and if the user want to make comparison between two variables, then only if-else statement is applicable. The following program illustrates the use of if-else construct in C-Language.

```
#include<stdio.h>
int main()
{
    int num1=1;
    int num2=2;
    if(num1>num2)                    //test-condition
    {
            printf("num1 is Larger");
    }
    else
    {
            printf("num2 is Larger");
    }
    return 0;
}
```

Output: ―num2 is Larger‖

The above program is start with initialisation of two variable num1 and num2 with value 1 & 2. The if statement evaluate the condition part of both the values and verify that either num1 is greater than num2 or not? The num1 have value 1 and num2 is 2, then num1>num2 condition evaluates to false therefore, the else block is executed and the output of program is num2 is Larger.

3.2.3 Nested If statement:

We already saw how useful if and else statements are, but what if we need to check for more conditions even when one condition is satisfied? Then C Language provides an extended feature as Nested-if statement. Nested if statement in C is the nesting of if statement within another if statement and nesting of if statement with an else statement. Once an else statement gets failed there are times when the next execution of statement wants to return a true statement, there we need nesting of if statement to make the entire flow of code in a semantic order. Nested if statement in C plays a very pivotal role in making a check on the inner nested statement of if statement with an else very carefully. The following syntax highlights the functioning of nested if statement.

Syntax:

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```

How nested-if statement works?

- The if statement evaluates the test expression inside the parenthesis ().
- If the test expression is evaluated to true, statements inside the body of if are executed.
- If the test expression is evaluated to false, statements inside the body of if are not executed.

Following program illustrates the use of Nested-if construct in C-Language.

```
#include<stdio.h>
void main()
{
    int a, b, c;

    printf("Enter three numbers\n");
```

```
scanf("%d %d %d", &a, &b, &c);

if(a > b)
{
        if(a > c)
                printf("a: %d is largest\n", a);
        else
                printf("c: %d is largest\n", c);
}
else
{
        if(b > c)
                printf("b: %d is largest\n", b);
        else
                printf("c: %d is largest\n", c);

}
}
```

Output:

Enter three numbers 89 12 65

Output: a:89 is largest

3.2.4 Else-if Ladder: The nested if statement provides the feature to evaluate the expression of three variables, but when the user wants to evaluate among more than three, then C language also provide the feature in form of else-if ladder. The else-if ladder helps user decide from among multiple options. The C/C++ if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The following syntax highlights the condition evaluation sections of else-if ladder statements.

Suntax:

```
if (Condition1)
{
```

```
    Statement1;
}
else if(Condition2)
{
    Statement2;
}
.
.
.
else if(ConditionN)
{
    StatementN;
}
else
{
    Default_Statement;
}
```

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. The following programs demonstrate the c-programme to evaluate the expression using else-if ladder statement.

```
#include<stdio.h>
void main ()
{
  int a,b,c,d;
  printf("Enter the values of a,b,c,d: ");
   scanf("%d%d%d%d",&a,&b,&c,&d);
        if(a>b && a>c && a>d)
        {
              printf("%d is the largest",a);
```

```
}
else if(b>c && b>a && b>d)
{
        printf("%d is the largest",b);
}
else if(c>d && c>a && c>b)
{
        printf("%d is the largest",c);
}
else
{
        printf("%d is the largest",d);
}
}
}
```

Output:

Enter the values of a,b,c,d: 15 20 25 30

30 is the largest

3.3 Loop control statements

In computer programming, sometimes programmer have to perform same task again and again on the same data with a few changes. In this situation programmer can either write same code again and again which consumes lots of time and space as well over can use loop to iterate same code to save time and space. Looping Statements in C execute the sequence of statements many times until the stated condition becomes false. A loop in C consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the C loop is to repeat the same code a number of times. C Language provides the following advantages of with the use of Looping?

- It provides code reusability.
- Using loops, we do not need to write the same code again and again.
- Using loops, we can traverse over the elements of data structures structures (array or linked lists).

Depending upon the position of a control statement in a program, looping statement in C is classified into two categories:

1. Entry controlled loop: In an entry control loop in C, a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.
2. Exit controlled loop: In an exit controlled loop, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an infinite loop. An infinite loop is also called as an ―Endless loop.‖ Following are some characteristics of an infinite loop:

1. No termination condition is specified.
2. The specified conditions never meet.

On the basis of these two categories, in C programming there are four types of loops are discussed below:

- While Loop
- Do-while loop
- For Loop
- Nested Loop

3.3.1 While Loop: While loop is entry controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed.

The syntax of the while loop is:

```
while (testExpression)
{
  // the body of the loop
}
```

The execution flow of while loop is explained below:

- The while loop evaluates the testExpression inside the parentheses ().
- If testExpression is true, statements inside the body of while loop are executed. Then, testExpression is evaluated again.
- The process goes on until testExpression is evaluated to false.
- If testExpression is false, the loop terminates (ends).

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance. The following program explain the functioning of while loop in c language.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
    int num=1;      //initializing the variable
    while(num<=10)        //while loop with condition
    {
        printf(" %d ",num);
        num++;                  //incrementing operation
    }
    return 0;
    }
```

Output: 1 2 3 4 5 6 7 8 9 10

The above program illustrates the use of while loop to print series of numbers from 1 to 10.

- We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
- In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
- In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

3.3.2 Do-while Loop Statements: Do-while is exit controlled loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is similar to the while loop with one important difference. The body of do-while loop is executed at least

once. Only then, the test expression is evaluated. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user. Therefore, it is also called post tested loop.

The execution flow of do-while loop is explained below:

- The body of do...while loop is executed once. Only then, the testExpression is evaluated.
- If testExpression is true, the body of the loop is executed again and testExpression is evaluated once more.
- This process goes on until testExpression becomes false.
- If testExpression is false, the loop ends.

The syntax of the C language do-while loop is given below:

```
do
{
//code to be executed
}while(testExpression);
```

As described above the do-while runs at least once even if the condition is false because the condition is evaluated, after the execution of the body of loop. The following program explain the functioning of do-while loop in c language.

// C Program to demonstrate the do-while loop behaviour

```
// when the condition is false from the start
#include <stdbool.h>
#include <stdio.h>
int main()
    {
   // declaring a false variable
   bool condition = false;
        do
        {
                printf("This is loop body.");
        } while (condition); // false condition

   return 0;
         }
```

Output:

This is loop body.

In the above programme, even when the condition is false at the start, the loop body is executed once. This is because in the do-while loop, the condition is checked after going through the body so when the control is at the start.

- It goes through the loop body.
- Executes all the statements in the body.
- Checks the condition which turns out to be false.
- Then exits the loop.

3.3.3 For Loop Statement: It is also called entry controlled loop. It is also provides a functionality/feature to recall a set of conditions for a defined number of times, moreover, this methodology of calling checked conditions automatically is known as for loop. It is mainly used to traverse arrays, vectors, and other data structures.

```
for(initialization; check/test expression; updation)
{
        // body consisting of multiple statements
}
```

Characteristics of for loop in C:

For loop follows a very structured approach where it begins with initializing a condition then checks the condition and in the end executes conditional statements following with updation of values.

- **Initialization**: This is the first parameter of a fundamental for loop that accepts a conditional variable that iterates the value or helps in checking the condition.
- **Conditional Statement**: It accepts 3 parameters (Initialization, Condition, and Updation) that indicate what condition needs to be followed and checked.
- **Check/Test Condition**: The Second parameter of a fundamental for loop defines the condition that needs to be followed to run the following code statements. In simple terms, if the check expression is true then the iteration of the loop continues otherwise the loop is terminated and further checks (if possible/there) are left unchecked.
- **Updation**: The Third parameter of a fundamental for loop defines the increment or decrement of the conditional variable that will iterate the code according to the condition.

To learn more about when the test expression is evaluated to true and false, the following program demonstrate the flow of for loop.

// Print numbers from 1 to 10

```
#include <stdio.h>
int main()
{
int i;
for (i = 1; i < 11; ++i)
{
printf("%d ", i);
}
return 0;
}
```

Output: 1 2 3 4 5 6 7 8 9 10

Explanation:

1. i is initialized to 1.
2. The test expression i < 11 is evaluated. Since 1 less than 11 is true, the body of for loop is executed. This will print the 1 (value of i) on the screen.
3. The update statement ++i is executed. Now, the value of i will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print 2 (value of i) on the screen.
4. Again, the update statement ++i is executed and the test expression i < 11 is evaluated. This process goes on until i becomes 11.
5. When i becomes 11, i < 11 will be false, and the for loop terminates.

3.3.4 Nested-Loop Statements: C supports nesting of loops in C. Nesting of loops is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C. Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. User can define any type of loop inside another loop; for example, you can define 'while' loop inside a 'for' loop.

Syntax of Nested loop:

```
Outer_loop
{
 Inner_loop
 {
  // inner loop statements.
 }
   // outer loop statements.
 }
```

Outer_loop and Inner_loop are the valid loops that can be any type of aforesaid loops.

Working of Nested Loop:

- Execution of statement within the loop flows in a way that the inner loop of the nested loop gets declared, initialized and then incremented.
- Once all the condition within the inner loop gets satisfied and becomes true it moves for the search of the outer loop. It is often called a loop within a loop.

Suppose we want to loop through each day of a week for 3 weeks. To achieve this, we can create a loop to iterate three times (3 weeks). And inside the loop, we can create another loop to iterate 7 times (7 days). This is how we can use nested loops. The following program exemplify the used of nested loop to print pattern using for loop statement.

```
// C program to display a triangular pattern
// Number is entered by the user
#include <stdio.h>
int main()
{
  int i, j, n;
  printf("Enter Number :  ");
  scanf ("%d", &n);
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= i; j++) {
      printf("*  ");
    }
```

```
    printf("\n");
    }
  return 0;
   }
```

Output:

Enter Number: 4

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

3.4 Case control statements:

The statements which are used to execute only specific block of statements in a series of blocks are called case control statements. There are 4 types of case control statements in C language named as:

- Switch Statement
- goto Statement
- break Statement
- continue Statement

3.4.1 Switch statement: Sometimes, there may be requirement to evaluate the multiple expression against different cases. Then switch statement in C tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed. Each case in a block of a switch has a different name/number which is referred to as an identifier. The value provided by the user is compared with all the cases inside the switch block until the match is found. The following syntax explain the method to use switch statement.

Syntax:

```
switch(expression)
{
  case constant-expression :
    statement(s);
    break; /* optional */
```

```
case constant-expression  :
   statement(s);
   break; /* optional */
/* you can have any number of case statements */
default : /* Optional */
statement(s);
}
```

Some major rules must be follow while using a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Let's see a simple example of c language using switch statement.

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
```

```
scanf("%d",&number);
switch(number){
case 10:
        printf("number is equals to 10");
        break;
case 50:
        printf("number is equal to 50");
        break;
case 100:
        printf("number is equal to 100");
        break;
default:
        printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output:

Case 1:

enter a number:4

number is not equal to 10, 50 or 100

Case 2:

enter a number:50

number is equal to 50

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

3.4.2 goto Statement: The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break

statement. However, using goto is avoided these days since it makes the program less readable and complex in large cases.

Syntax of goto Statement:

goto label;

... .

... .

label:

statement;

The label is an identifier. When the goto statement is encountered, the control of the program jumps to label: and starts executing the code. The following program exemplify the use of goto to print table.

```
//Print a number table
#include <stdio.h>
int main()
{
  int num,i=1;
  printf("Enter the number whose table you want to print?");
  scanf("%d",&num);
  table:
  printf("%d x %d = %d\n",num,i,num*i);
  i++;
  if(i<=10)
  goto table;
}
```

Output:

Enter the number whose table you want to print? 10

10 x 1 = 10

10 x 2 = 20

10 x 3 = 30

10 x 4 = 40

10 x 5 = 50

10 x 6 = 60

10 x 7 = 70

10 x 8 = 80

10 x 9 = 90

10 10 = 100

3.4.3 Break Statement: The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.

Syntax

The syntax for a break statement in C is as follows –

```
{
    break;
}
```

The break statement in C programming has the following two usages:

- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the switch statement.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block. The following two programs demonstrate the use of break statement in both the cases.

```
#include<stdio.h>
int main()
{
int i=1,j=1;//initializing a local variable
for(i=1;i<=3;i++)
{
    for(j=1;j<=3;j++)
    {
        printf("%d &d\n",i,j);
        if(i==2 && j==2)
```

```
        {
            break;//will break loop of j only
        }
        }//end of for loop
return 0;
}
```

Output:

1 1

1 2

1 3

2 1

2 2

3 1

3 2

3 3

The following program exemplify the use of break in switch statement.

```
#include<stdio.h>
int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:
    printf("number is equals to 10");
    break;
case 50:
    printf("number is equal to 50");
    break;
case 100:
    printf("number is equal to 100");
    break;
default:
    printf("number is not equal to 10, 50 or 100");
}
```

return 0;

}

Output:

Case 1:

enter a number:4

number is not equal to 10, 50 or 100

Case 2:

enter a number:50

number is equal to 50

3.4.4 Continue Statement: The continue statement in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition. The continue statement in C programming works somewhat like the break statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control to pass to the conditional tests. The syntax and example of continue statement is given below:

Syntax:

```
//loop statements
continue;
//some lines of the code which is to be skipped
```

Continue statement example 1

```
#include<stdio.h>
void main ()
{
   int i = 0;
   while(i!=10)
   {
      printf("%d", i);
      continue;
```

```
        i++;
         }
        }
```

Output:

```
        infinite loop
```

Continue statement example 2

```
#include<stdio.h>
int main()
        {
        int i=1;//initializing a local variable
        //starting a loop from 1 to 10
for(i=1;i<=10;i++)
        {
        if(i==5)
        {//if value of i is equal to 5, it will continue the loop
                continue;
        }
        printf("%d \n",i);
        }//end of for loop
return 0;
        }
```

Output:

```
        1
        2
        3
        4
        6
        7
        8
        9
        10
```

As you can see, 5 is not printed on the console because loop is continued at i==5.

COURSE: COMPUTER PROGRAMING

UNIT 4: ARRAYS AND POINTERS IN C

4.1 Arrays

4.1.1 Characteristic of Arrays

4.1.2 Representation of Arrays

4.1.3 Declaration and Initialization of an Array

4.2 Types of Arrays

4.2.1 One-Dimensional Arrays

4.2.2 Multi-Dimensional Arrays:

4.3 Pointer

4.3.1 Pointer Declaration and Initialization

4.3.2 Types of Pointers

4.3.3 Pointer Expressions and Pointer Arithmetic

4.1 Arrays

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

For example, to store the marks of a student in 4 different subjects, there is no need to define different variables for the marks in the different subject. Instead of that, array can be used which can store the marks in each subject at the contiguous memory locations.

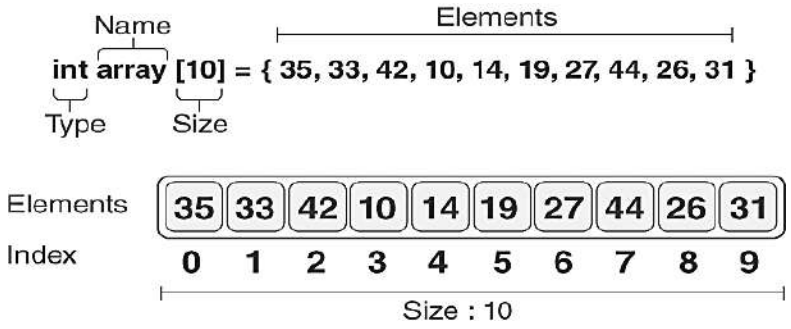
4.1.1 Characteristic of Arrays:

1. Array is a group of variables of similar data types referred to by a single element.
2. Array elements are stored in a contiguous memory location.
3. The size of the array should be mentioned while declaring it.
4. Array elements are always counted from zero (0) onward.
5. Array elements can be accessed using the position of the element in the array.
6. The array can have one or more dimensions.

4.1.2 Representation of an Array:

Arrays always store the same type of values. In the below mentioned example:

- **int** is a data type of Array having size 10.
- Data items stored in an array are known as elements.
- The location or placing of each element has an index value.



4.1.3 Declaration and Initialization of an Array:

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

Syntax: data_type array_name[array_size];

Example: int marks[4];

In this example, int is the *data_type*, marks are the *array_name*, and 4 is the *array_size*. Hence, marks is the one-dimensional array.

NOTE: The name of the array is also a pointer to the first element of the array.

Initialization of Array:

A) The simplest way to initialize an array is by using the index of each element as mentioned in the following example.

```
//Program to demonstrate the Initialization of Array
#include<stdio.h>
int main()
{
int i=0;
```

```
int marks[4];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
//traversal of array
for(i=0;i<4;i++){
printf("%d \n",marks[i]);
}//end of for loop
return 0;
}
```

Output:

```
80
60
70
8585
```

B) Initialize elements of an array at the time of declaration

Syntax: datatype Array_Name[size] = { value1, value2, value3, ….. valueN };

Example: int marks[4]={30,40,50,60};

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

int marks[]={30,40,50,60};

```
//Program to demonstrate the Initialization of Array
#include<stdio.h>
int main(){
int i=0;
int marks[4]={30,40,50,60};
//declaration and initialization of array
//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
```

```
}
return 0;
}
```

Output:

```
30
40
50
60
```

Advantages of Array:

- It is a better version of storing the data of the same size and same type.
- It enables us to collect the number of elements in it.
- Arrays have a safer cache positioning that improves performance.
- Arrays can represent multiple data items of the same type using a single name.

Disadvantages of Array:

- In an array, it is essential to identify the number of elements to be stored.
- It is a static structure. It means that in an array, the memory size is fixed.
- When it comes to insertion and deletion, it is a bit difficult because the elements are stored sequentially and the shifting operation is expensive.

NOTE: Array elements are accessed by using an integer index. Array index starts with 0 and goes till the size of the array minus 1.

4.2 Types of Arrays

There are two types of arrays:

- One-Dimensional Arrays
- Multi-Dimensional Arrays

4.2.1 One-Dimensional Arrays: A one-dimensional array is a kind of linear array. It involves single sub-scripting. The [] (brackets) is used for the subscript of the array and to declare and access the elements from the array.

Syntax: DataType ArrayName [size];

Example: int marks [6];

4.2.2 Multi-Dimensional Arrays: A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order. In multi-dimensional arrays, there are two categories:

- Two-Dimensional Arrays
- Three-Dimensional Arrays

Two-Dimensional Arrays

An array involving two subscripts [] [] is known as a two dimensional array. They are also known as the array of the array. Two-dimensional arrays are divided into rows and columns and are able to handle the data of the table.

Syntax: DataType ArrayName[row_size][column_size];

Example: int a [6][6];

- **Initializing Two-Dimensional Arrays**

1. **First Method**:
   int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
   The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.
2. **Second Method**:
   int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

```
//Program to demonstrate two-dimensional Array
#include<stdio.h>
int main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
for(i=0;i<4;i++){
for(j=0;j<3;j++){
```

```
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
  }//end of j
 }//end of i
 return 0;
 }
```

Output:

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

3. **Third Method:**

```
int x[3][4];

for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 4; j++)
{
        Scanf(—%d‖,  x[i][j]);

    }

  }
```

- **Accessing Two-Dimensional Arrays:** Elements of Two-Dimensional arrays can be accessed using the row indexes and column indexes.

  Example: int x[2][1];

The above example represents the element present in the third row and second column.

Three-Dimensional Arrays

When there is need to create two or more tables of the elements to declare the array elements, then in such a situation three-dimensional arrays is used.

Syntax: DataType ArrayName[size1][size2][size3];

For Example: int a [6][5][5];

- **Initializing Three-Dimensional Array**: Initialization in a Three-Dimensional array is the same as that of Two-dimensional arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

Method 1:

```
int x[2][3][4] =    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                     11, 12, 13, 14, 15, 16, 17, 18, 19,
                     20, 21, 22, 23};
```

Method 2:

```
int x[2][3][4] =    {
                     { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
                     { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
                     };
```

- **Accessing Three-Dimensional Arrays**: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

// C program to print elements of Three-Dimensional Array

```
#include <stdio.h>

int main(void)
```

```
{
   // initializing the 3-dimensional array
   int x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },
                      { { 6, 7 }, { 8, 9 }, { 10, 11 } } };

   // output each element's value
   for (int i = 0; i < 2; ++i) {
      for (int j = 0; j < 3; ++j) {
         for (int k = 0; k < 2; ++k) {
            printf("Element at x[%i][%i][%i] = %d\n", i, j, k, x[i][j][k]);
         }
         }
         }
   return (0);
        }
```

Output:

```
Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7
Element at x[1][1][0] = 8
Element at x[1][1][1] = 9
Element at x[1][2][0] = 10
Element at x[1][2][1] = 11
```

Difference between One-Dimensional and Two-Dimensional Array

Parameters	One-Dimensional Array	Two-Dimensional Array
Concept	A one-dimensional array stores a single list of various elements having a similar data type.	A two-dimensional array stores an array of various arrays, or a list of various lists, or an array of various one-dimensional arrays.
Representation	It represents multiple data items in the form of a list.	It represents multiple data items in the form of a table that contains columns and rows.
Dimensions	It has only one dimension.	It has a total of two dimensions.
Total Size (in Bytes)	Total number of Bytes = The size of array * the size of array variable or datatype.	Total number of Bytes = The size of array visible or datatype * size of second index * size of the first index.

4.3 Pointer

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The pointer sizes depend on their architecture.

NOTE: The pointer of type void is called **Void pointer** or **Generic pointer**. Void pointer can hold address of any type of variable.

The following operators are used for the pointers in C language:

Operator	Name	Uses and Meaning of Operator
*	Asterisk	Declares a pointer in a program. Returns the referenced variable‘s value.
&	Ampersand	Returns a variable‘s address

4.3.1 Pointer Declaration and Initialization

Just like the variables, there is need to declare the pointers in C before we use them in any program.

Syntax:

datatype *Pointer_var_name;

- The data_type refers to this pointer‘s base type in the variable of C. It indicates which type of variable is the pointer pointing to in the code.
- The asterisk (*) is used to declare a pointer which is also called the indirection pointer.

Example:

int *q; // a pointer q for int data type

char *x; // a pointer x for char data type

Pointer Initialization

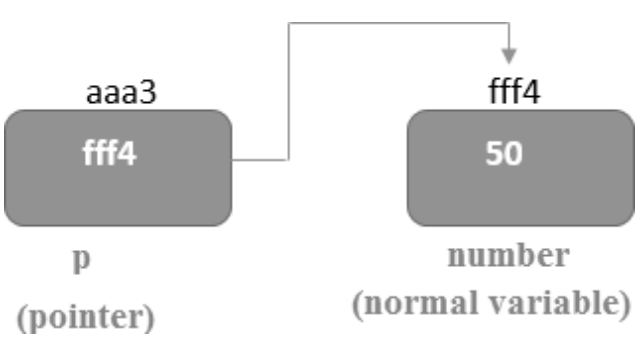
After the pointer declaration, the next step is to initialise the pointers just like the standard variables using an address of the variable. The ampersand (&) operator is used to get the variable‘s address in the program. If we don‘t initialise the pointers in any C program and start using it directly, the results will be ambiguous.

Syntax:

pointer = &variable;

Example:

```
int *p;
p=&number;
```



// C program to illustrate Pointers

```
#include<stdio.h>
 int main(){
 int number=50;
 int *p;
 p=&number; //stores the address of number variable
 printf("Address of number variable is %x \n",p);
 // p prints the address of the number
 printf("Value of p variable is %d \n",*p);
 // prints the value stored in pointer p.
 return 0;
 }
```

OUTPUT:

Address of number variable is fff4

Value of p variable is 50

4.3.2 Types of Pointers

There are various types of pointers that can be used in the C language.

- **The Null Pointer:** To create a null pointer in C language, assign the pointer with a null value during its declaration in the program. This type of method is especially useful when the pointer has no address assigned to it. The null pointer has a value of 0.

//Use of NULL Pointer

```
#include <stdio.h>
int main()
{
int *a = NULL; // the null pointer declaration
printf(—Value of the variable a in the program is equal to :\n%x‖,a);
return 0;
}
```

Output:

Value of the variable a in the program is equal to: 0

- **The Void Pointer:** This pointer has no standard data type, and declares it with the use of the keyword ***void***. The void pointer is generally used for the storage of any variable‘s address. The void pointer is also known as the generic pointer in the C language.

//Use of void Pointer

```
#include <stdio.h>
int main()
{
void *q = NULL; // the void pointer of the program
printf(—Size of the void pointer in the program is equal to : %d\n‖,sizeof(q));
return 0;
}
```

Output:

Size of the void pointer in the program is equal to: 4

Advantages of using Pointers

- Pointers make it easy for us to access locations of memory.
- Pointers are used for dynamic allocation and deallocation of memory.
- Pointers are also used to create complex data structures, like the linked list, tree, graph, etc.
- Pointers provide an efficient way in which we can access the elements present in the structure of an array.
- Pointers reduce the code and thus improve the overall performance. We can use the pointers for returning various values from any given function.

Disadvantages of using Pointers

- The concept of pointers is a bit tricky to understand.
- Pointers in a program can lead to leakage of memory.
- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are comparatively slower than variables in C.
- These can lead to some errors like segmentation faults, accessing of unrequired memory locations, and many more.

Applications of Pointers

There are various uses of the pointers in C language.

- Dynamic allocation of memory: We can allocate the memory dynamically in C language by using the calloc() and malloc() functions along with the pointers.
- Structures, Functions, and Arrays use the pointers in the C. It help in reducing the code and improving the program‘s performance.

4.3.3 Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers. The Pointer Arithmetic is slightly different from the ones that we generally use for mathematical calculations. The operations are:

- Increment/Decrement of a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers to the same type of pointers.

NOTE: Pointer arithmetic is meaningless unless performed on an array.

// C program to illustrate Pointer Arithmetic

```
#include <stdio.h>

int main()
{

    // Declare an array
    int v[3] = { 10, 100, 200 };

    // Declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
```

```
ptr = v;

for (int i = 0; i < 3; i++) {

    // print value at address which is stored in ptr
    printf("Value of *ptr = %d\n", *ptr);

    // print value of ptr
    printf("Value of ptr = %p\n\n", ptr);

    // Increment pointer ptr by 1
    ptr++;
    }
return 0;
    }
```

OUTPUT:

```
Value of *ptr = 10
Value of ptr = 0x7ffe8ba7ec50

Value of *ptr = 100
Value of ptr = 0x7ffe8ba7ec54

Value of *ptr = 200
Value of ptr = 0x7ffe8ba7ec58
```

Pointers and Arrays

- **Array of Pointers:** Pointers are variables which stores the address of another variable. Pointer to an array points the address of memory block of an array variable.

Syntax:

```
datatype *variable_name[size];
```

- **datatype** defines the datatype of variable like int, char, float etc.

- **variable_name** is the name of variable given by user.
- **size** defines the size of array variable.

Example: int (*ptr) [10];

//Program to demonstrate Arrays of Pointers

```
#include <stdio.h>
int main ()
{
  int *arr[3];
  int *a;
  printf( "Value of array pointer variable : %d", arr);
  printf( "Value of pointer variable : %d", &a);
  return 0;
}
```

Output:

Value of array pointer variable: 1481173888

Value of pointer variable: 1481173880

- **Pointer to an Array:** In a pointer to an array, store the base address of the array in the pointer variable.

  Example: *ptr=arr;
  *ptr=&arr;
  *ptr=&arr[0];

  Here, ptr will store the base address of the array.

//Program to demonstrate Pointer to Arrays

```
#include <stdio.h>
int main()
{
// array declaration and initialization
        int arr[5] = {3, 5, 7, 9, 11}, i;

// both `arr` and `&arr` return the address of the first element of the array.
```

```
        int *ptr = arr;

// printing the elements of array using addition arithmetic on pointer
        for(i = 0; i < 5; i++)
        {
                printf("%d ", *(ptr + i));
        }
        return 0;
        }
```

Output:

3 5 7 9 11

COURSE: COMPUTER PROGRAMING

UNIT 5: FUNCTIONS AND POINTERS IN C

5.1 Functions in C

5.1.1 Function Declaration and Definition

5.1.2 Functions Definition

5.1.3 Library Vs. User-defined Functions

5.1.4 Function calling Methods

5.1.5 Function Parameters

5.1.5.1 Actual Parameter

5.1.5.2 Formal Parameter

5.1.6 Parameter Passing Techniques

5.1.7 Recursion in C

5.1.8 Pointers and Functions

5.2 Strings in C

5.2.1 Difference between char array and string literal

5.2.2 Traversing String

5.2.3 Accepting string as the input

5.2.4 Pointers with strings

5.2.5 String Functions

5.1 Functions in C

Function in C programming is a reusable block of code that makes a program easier to understand, test and can be easily modified without changing the calling program. Functions divide the code and modularize the program for better and effective results. In short, a larger program is divided into various subprograms which are called as functions

There are three components of a C function.

5.1.1 Function Declaration: A function must be declared globally in a C program to tell the compiler about the function name, function parameters, and return type. Function declaration means writing a name of a program. It is a compulsory part for using functions in code. In a function declaration, we just specify the name of

a function that we are going to use in our program like a variable declaration. The function declarations (called prototype) are usually done above the main () function and take the general form:

Syntax: return_data_type function_name (data_type arguments);

- The **return_data_type**: is the data type of the value function returned back to the calling statement.
- The **function_name**: is followed by parentheses
- **Arguments:** names with their data type declarations optionally are placed inside the parentheses.

Example: consider the following program that shows how to declare a cube function to calculate the cube value of an integer variable

```
#include <stdio.h>
/*Function declaration*/
int add(int a,b);
/*End of Function declaration*/
```

5.1.2 Function Definition: Function definition means just writing the body of a function. A body of a function consists of statements which are going to perform a specific task. A function body consists of a single or a block of statements. It is also a mandatory part of a function. It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

```
int add(int a,int b)  //function body
        {
        int c;
        c=a+b;
        return c;
        }
```

1. **Function Call:** A function call means calling a function whenever it is required in a program. Whenever we call a function, it performs an operation for which it was designed. A function call is an optional part of a program. Function can be called from anywhere in the program. The parameter list must not differ in function calling and function

declaration. We must pass the same number of functions as it is declared in the function declaration.

```
                        result = add(4,5);
#include <stdio.h>
int add(int a, int b);              //function declaration
int main()
{
        int a=10,b=20;
        int c=add(10,20);   //function call
        printf("Addition:%d\n",c);
        getch();
}
int add(int a,int b)  //function body
{
        int c;
        c=a+b;
        return c;
}
```

Output:

Addition:30

5.1.3 Library Vs. User-defined Functions

Every C program has at least one function which is the main function, but a program can have any number of functions. The main () function in C is a starting point of a program. In _C‘ programming, functions are divided into two types:

1. Library functions
2. User-defined functions

The difference between the library and user-defined functions in C is that we do not need to write a code for a library function. It is already present inside the header file which we always include at the beginning of a program.

- **Library Functions:** Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions

are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, Tt use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output. The list of mostly used header files is given in the following table.

S.No	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.

5.1.4 Function calling Methods

A function may or may not accept any argument. It may or may not return any value. Based on these facts, there are four different aspects of function calls. The details of each with suitable example is given below:

Function without arguments and without return value

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

```
    printf("Javatpoint");
}
```

OUTPUT:

Hello Javatpoint

Function with arguments and without return value

```
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

OUTPUT:

Going to calculate the sum of two numbers:

Enter two numbers 10 24

The sum is 34

Function without arguments and with return value

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();
    printf("%d",result);
}
```

```
int sum()
{
   int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   return a+b;
}
```

OUTPUT:

Going to calculate the sum of two numbers:

Enter two numbers 10 24

The sum is 34

Function with arguments and with return value

```
#include<stdio.h>
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\nGoing to calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
   return a+b;
}
```

OUTPUT:

Going to calculate the sum of two numbers:

Enter two numbers: 10 20

The sum is: 30

5.1.5 Function Parameters

A parameter is a way to convey information to a function. Sometimes we also refer to parameters as arguments. Parameters are nothing but the input provided to the function. Function work on these parameters to produce a result or to perform a certain action. The data necessary for the function to perform the task is sent as parameters. Parameters can be actual parameters or Formal Parameters. The key difference between Actual Parameters and Formal Parameters is that Actual Parameters are the values that are passed to the function when it is invoked while Formal Parameters are the variables defined by the function that receives values when the function is called. There is basic two terminology used when using arguments in functions**:**

5.1.5.1 Actual Parameter: Actual parameters are the values, variables or expressions that we pass to a calling function. While defining the actual parameters in the calling function we do not associate data type with the parameters. In a calling function, the actual parameters can be provided either by the programmer at the time of programming or by the user in real-time when the program is being executed.

5.1.5.2 Formal Parameter: Formal parameters are the variable that we pass to function when it is declared or defined in the program. The formal parameters defined during function definition accept the real values in the order they are passed by the calling function. The statements in the function definition access these real values by referring to them with the variable name defined by the formal parameter.

The following program demonstrates the concept of actual and formal arguments:

In this example, the value of variable x is 10 before the function func_1() is called, after func_1() is called, the value of x inside main() is still 10. The changes made inside the function func_1() doesn't affect the value of x. This happens because when we pass values to the functions, a copy of the value is made and that copy is passed to the formal arguments. Hence Formal arguments work on a copy of the original value, not the original value itself, that's why changes made inside func_1() is not reflected inside main().

```
#include<stdio.h>
void func_1(int);
int main()
{
int x = 10;
printf("Before function call\n");
```

```
printf("x = %d\n", x);
func_1(x);
printf("After function call\n");
printf("x = %d\n", x);
// signal to operating system program ran fine
return 0;
}
void func_1(int a)
{
 a += 1;
 a++;
 printf("\na = %d\n\n", a);
}
```

5.1.6 Parameter Passing Techniques

In C, a function specifies the modes of parameter passing to it. There are two ways to specify function calls: call by value and call by reference in C. In call by value, the function parameters gets the copy of actual parameters which means changes made in function parameters did not reflect in actual parameters. In call by reference, the function parameter gets reference of actual parameter which means they point to similar storage space and changes made in function parameters will reflect in actual parameters. During the calling of a function, there will be two ways in which we can perform the passing of these arguments to a given function:

Type of Call	**Description**
Call By Reference	The Call by Reference method creates a copy of the address of the given argument into the parameter that is formal in nature. Inside this function, the use of address helps in accessing the actual argument that comes in use in this call. It means that the changes that appear on the parameter are bound to affect the given argument.
Call By Value	The Call by Value method creates a copy of the actual value of the given argument in the parameter of the function that is formal in nature. Here, the changes that appear on the parameter (that exists inside the function) create no effect whatsoever on the available argument.

The C programming, by default, is the Call by Value for passing the arguments. Generally, it means that we cannot make use of the code that exists within a function for altering the arguments that help in calling the function.

// C program to show use of call by value

```
#include <stdio.h>
 void swap(int var1, int var2)
{
  int temp = var1;
  var1 = var2;
  var2 = temp;
}
 int main()
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
         var1, var2);
  swap(var1, var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
         var1, var2);
  return 0;
}
```

OUTPUT:

Before swap Value of var1 and var2 is: 3, 2

After swap Value of var1 and var2 is: 3, 2

// C program to show use of call by Reference

```
#include <stdio.h>
 void swap(int *var1, int *var2)
{
  int temp = *var1;
  *var1 = *var2;
  *var2 = temp;
}
 int main()
```

```
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
        var1, var2);
  swap(&var1, &var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
        var1, var2);
  return 0;
}
```

OUTPUT:

Before swap Value of var1 and var2 is: 3, 2

After swap Value of var1 and var2 is: 2, 3

5.1.7 Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to the entire problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems. Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc. In the following example, recursion is used to calculate the factorial of a number.

```
 #include <stdio.h>
 int fact (int);
 int main()
 {
int n,f;
 printf("Enter the number whose factorial you want to calculate?"); scanf("%d",&n);
   f = fact(n);
```

```
   printf("factorial = %d",f);
}
 int fact(int n)
 {
    if (n==0)
    {
       return 0;
    }
    else if ( n == 1)
    {
       return 1;
    }
    else
    {
       return n*fact(n-1);
  }  }
```

Output:

Enter the number whose factorial you want to calculate? 5

factorial = 120

Memory allocation of Recursive method: Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Advantages of Using Functions

The functions in the C programming language offer the following advantages:

- One of the primary achievements of the C functions is reusability.
- When we make use of the functions, then we can easily avoid the rewriting of the same code/ logic time and again multiple times in any program.

- The calling of C functions may appear as many numbers of times as we want in any program. And we can do so from any place in the program that is given to us.
- We can perform the tracking of a large C program pretty easily if we divide it into various functions.
- However, remember that the function calling always acts as an overhead in the case of a C program.

5.1.8 Pointers and Functions

As discussed earlier we can create a pointer of any data type such as int, char, float. Similarly, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer. C programming allows to create a pointer pointing to the function, which can be further passed as an argument to the function. We can create a function pointer as follows:

Syntax: (type) (*pointer_name)(parameter);

In the above syntax, **type** is the variable type which is returned by the function, ***pointer_name** is the function pointer, and the **parameter** is the list of the argument passed to the function.

Example: **int** (*ip) (**int**);

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

float (*fp) (**float**);

In the above declaration, ***fp** is a pointer that points to a function that returns a float value and accepts a float value as an argument.

NOTE: The declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a **'*'**.

```
//Program to demonstrate the Function to Pointers
#include <stdio.h>
int add(int,int);
int main()
{
  int a,b;
  int (*ip)(int,int);
  int result;
```

```
    printf("Enter the values of a and b : ");
    scanf("%d %d",&a,&b);
    ip=add;
    result=(*ip)(a,b);
    printf("Value after addition is : %d",result);
     return 0;
}
int add(int a,int b)
{
    int c=a+b;
    return c;
}
```

Output:

Enter the values of a and b: 4 55

Value after addition is: 59

5.2 Strings in C

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

```
char ch[10]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

As we know, array index starts from 0, so it will be represented as in the figure given below:



While declaring string, size is not mandatory. So we can write the above code as given

below:

char ch[]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

We can also define the **string by the string literal** in C language. For example:

char ch[]="javatpoint";

In such case, '\0' will be appended at the end of the string by the compiler.

5.2.1 Difference between char array and string literal

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array whereas; it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas; we can reassign the characters of the array.

String Example in C

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4. char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n',          't', '\0'};
5.  char ch2[11]="javatpoint";
6.  printf("Char Array Value is: %s\n", ch);
7.  printf("String Literal Value is: %s\n", ch2);
8.  return 0;
9. }
```

Output

Char Array Value is: javatpoint

String Literal Value is: javatpoint

5.2.2 Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character

- **Using the length of string**

Let's see an example of counting the number of vowels in a string.

```
#include<stdio.h>
void main ()
{
   char s[11] = "javatpoint";
   int i = 0;
   int count = 0;
   while(i<11)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels %d",count);
}
```

Output

The number of vowels 4

- **Using the null character**

  Let's see the same example of counting the number of vowels by using the null character.

```
#include<stdio.h>
void main ()
    {
   char s[11] = "javatpoint";
   int i = 0;
   int count = 0;
   while(s[i] != NULL)
    {
 if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
     {
        count ++;  }
     i++;
     }
   printf("The number of vowels %d",count);
    }
```

Output

The number of vowels 4

5.2.3 Accepting string as the input

```
#include<stdio.h>
void main ()
{
   char s[20];
   printf("Enter the string?");
   scanf("%s",s);
   printf("You entered %s",s);
}
```

Output

Enter the string? C Programming

You entered C Programming

NOTE:

- The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.
- Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

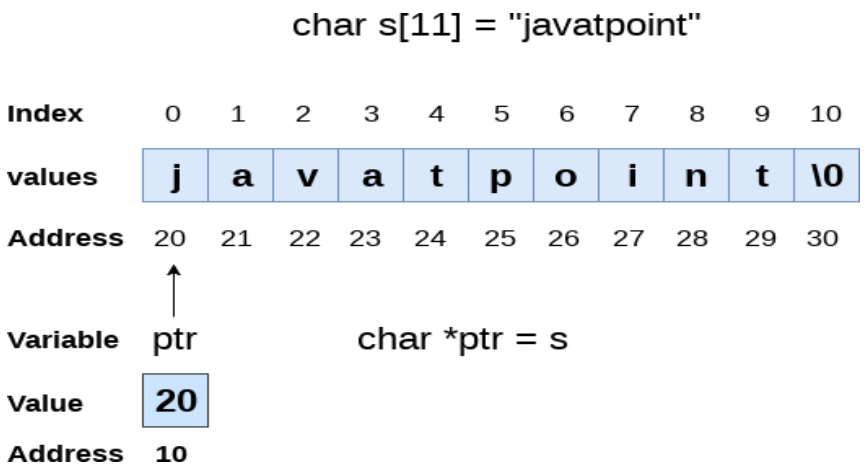
5.2.4 Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

```
#include<stdio.h>
void main ()
{
   char s[11] = "javatpoint";
   char *p = s; // pointer p is pointing to string s.
   printf("%s",p);
}
```

Output

Javatpoint



As we know that string is an array of characters, the pointers can be used in the same way

they were used with arrays. In the above example, p is declared as a pointer to the array of characters s. P affects similar to s since s is the base address of the string and treated as a pointer internally. However, we cannot change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>
void main ()
   {
   char *p = " C Programming ";
   printf("String p: %s\n",p);
   char *q;
   printf("copying the content of p into q...\n");
   q = p;
   printf("String q: %s\n",q);
   }
```

Output

String p: C Programming

copying the content of p into q...

String q: C Programming

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
 #include<stdio.h>
 void main ()
 {
    char *p = " C Programming ";
    printf("Before assigning: %s\n",p);
    p = "hello";
    printf("After assigning: %s\n",p);
 }
```

Output

Before assigning: C Programming

After assigning: hello

5.2.5 String Functions

gets() function: enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration: char[] gets(char[]);

Reading string using gets()

```
#include<stdio.h>
void main ()
{
   char s[30];
   printf("Enter the string? ");
   gets(s);
   printf("You entered %s",s);
    }
```

Output

```
Enter the string?
javatpoint is the best
You entered javatpoint is the best
```

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered

puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

Declaration: **int** puts(**char**[])

Let's see an example to read a string using gets() and print it on the console using puts().

```
#include<stdio.h>
#include <string.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

Output:

```
Enter your name: ABC
Your name is: ABC
```

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

COURSE: COMPUTER PROGRAMING

UNIT 6: USER DEFINED DATA TYPES IN C

6.1 Structure

6.1.1 Structure Variables Declaration

6.1.2 Accessing Structure Data Members

6.1.3 Array of Structures

6.1.4 Nested Structure

6.1.5 Passing structure to function

6.1.6 Structures Limitations

6.2 Union

6.3 Difference between Structure and Union in C

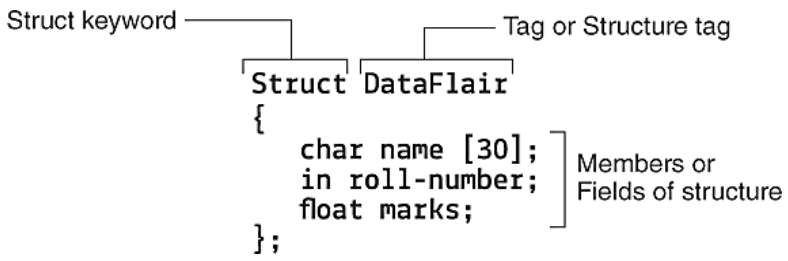
The data types that are defined by the user depending upon the use case of the programmer are termed user-defined data types. C offers a set of constructs that allow users to define their data types. These user-defined data types are constructed using a combination of fundamental data types and derived data types. Fundamental data types are basic built-in types of C programming language. These are integer data type (int), floating data type (float), and character data type (char). Derived data types are derived from fundamental data types, like functions, arrays, and pointers in the C programming language. For example, an array is a derived data type as it contains elements of a similar data type and acts like a new data type in the C programming language. User-defined data types are created by the user using a combination of fundamental and derived data types in the C programming language. To create a user-defined data type, the C programming language provides following five constructs to define new data type named as:

- Structures (struct)
- Union
- Type definitions (Typedef)
- Enumerations (enum)
- Empty data type (void)

6.1 Structure

Structures are the user-defined data type, which allow us to collect the group of different data

types. Here, all the individual components or elements of structures are known as a member. _struct‘ keyword is used to create a structure.



Structure data type is used to store dates of different attributes of different data types.

Syntax to Define the Structure in C:

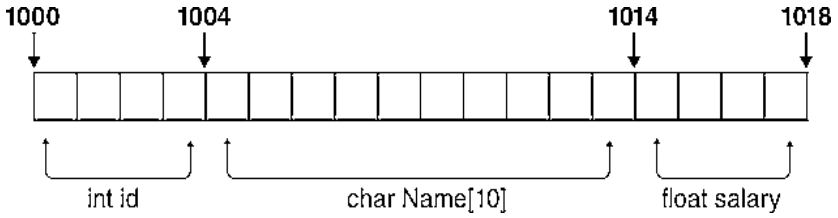
```
Struct structure_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memberN;
};
```

Example of Structure in C:

```
struct student
{
int roll_number;
char name[20];
float percentage;
};
```

In this example structure student is keeping the information of a student which consists of three data fields, roll number, name, and percentage. These fields are known as structure elements or members. These elements are of different data types.

Memory Allocation of Structure:



6.1.1 Structure Variables Declaration

A structure variable can either be declared with structure declaration or as a separate declaration like basic types. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

Syntax:

```
struct struct_name var_name;
or
struct struct_name
 {
DataType member1_name;
DataType member2_name;
DataType member3_name;
…
} var_name;
```

// A variable declaration with structure declaration.

```
struct Point
{
  int x, y;
} p1;  // The variable p1 is declared with 'Point'

// A variable declaration like basic data types
struct Point
{
  int x, y;
};

int main()
{
  struct Point p1;
// The variable p1 is declared like a normal variable
}
```

NOTE: Structure members cannot be initialized with declaration.

6.1.2 Accessing Structure Data Members

There are two ways to access structure members:

1. By **.** (member or dot operator)
2. By **-->** (structure pointer operator)

- Structure members are accessed using dot (.) operator.

Syntax: var_name.member1_name;
var_name.member2_name;

```
#include <stdio.h>
 struct Point {
    int x, y;
};
int main()
{
    struct Point p1 = { 0, 1 };

    // Accessing members of point p1
    p1.x = 20;
    printf("x = %d, y = %d", p1.x, p1.y);

    return 0;
}
```

Output:

x = 20, y = 1

6.1.3 Array of Structures

An array of structures in C language can be defined as the collection of multiple structure variables where each variable contains information about different entities. The array of structures in C language is used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

//**Program to illustrate an array of structures** #include<stdio.h>

```
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
 for(i=0;i<5;i++){
 printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
 }
  return 0;
 }
```

Output:

Enter Rollno:1

Enter Name:Sunny

Enter Rollno:2

Enter Name:Ranjan

Enter Rollno:3

Enter Name:Vishal

Enter Rollno:4

Enter Name:Jhanvi

Enter Rollno:5

Enter Name:Saroj

Student Information List:
Rollno:1, Name:Sunny
Rollno:2, Name:Ranjan
Rollno:3, Name:Vishal
Rollno:4, Name:Jhanvi
Rollno:5, Name:Saroj

6.1.4 Nested Structure

C provides the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

```
#include<stdio.h>
struct address
  {
   char city[20];
   int pin;
   char phone[14];
   };
struct employee
   {
   char name[20];
   struct address add;
    };
void main ()
   {
 struct employee emp;
 printf("Enter employee information\n");
 scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
 printf("Printing the employee information. ..\n");
 printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.p
```

```
in,emp.add.phone);
}
```

Output:

```
Enter employee information
Aman
Chandigarh
160014
8427526788
Printing the employee information....
name: Aman
City: Chandigarh
Pincode: 160014
Phone: 8427526788
```

6.1.5 **Passing structure to function**

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

```
#include<stdio.h>
struct address
   {
   char city[20];
   int pin;
   char phone[14];
    };
struct employee
    {
   char name[20];
   struct address add;
    };
void display(struct employee);
void main ()
```

```
{
 struct employee emp;
 printf("Enter employee information?\n");
 scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
 display(emp);
 }
void display(struct employee emp)
 {
printf("Printing the details. ..\n");
printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
 }
```

6.1.6 Structures Limitations

In C language, Structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures have some limitations.

- The C structure does not allow the struct data type to be treated like built-in data types:
- C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure
- C structures do not permit functions inside Structure
- C Structures cannot have static members inside their body
- C Programming language does not support access modifiers. So they cannot be used in C Structures.
- Structures in C cannot have a constructor inside Structures.

We cannot use operators like +,- etc. on Structure variables. For example, consider the following code:

```
struct number
{
   float x;
};
int main()
{
```

```
    struct number n1, n2, n3;
    n1.x = 4;
    n2.x = 3;
    n3 = n1 + n2;
    return 0;
   }

  /*Output:
  prog.c: In function 'main':
  prog.c:10:7: error:
  invalid operands to binary + (have 'struct number' and
  'struct number') n3=n1+n2;
  */
```

But we can use arithmetic operation on structure variables as mentioned in example given below:

```
// Use of arithmetic operator in structure
#include <stdio.h>
struct number {
   float x;
  };
int main()
 {
   struct number n1, n2, n3;
   n1.x = 4;
   n2.x = 3;
   n3.x = (n1.x) + (n2.x);
   printf("\n%f", n3.x);
   return 0;
   }
```

Output:

6.000000

6.2 Union

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time. Union is a user-defined data type, but unlike structures, they share the same memory location.

```
struct abc
{
  int a;
  char b;
}
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type **int** and 'b' of type **character**. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

```
union abc
{
  int a;
  char b;
}
```

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

NOTE: The size of the union is based on the size of the largest member of the union.

Example:

```
union abc{
int a;
char b;
float c;
double d;
};
```

```
int main()
{
  printf("Size of union abc is %d", sizeof(union abc));
  return 0;
}
```

Output:

Size of union abc is 8 bytes

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. As the double data type occupies the largest memory among all the four data types, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

6.3 Difference between Structure and Union in C

Parameter	**Structure**	**Union**
Keyword	**struct** keyword is used to define a Structure.	**union** keyword is used to define a Union.
Internal Implementation	The implementation of Structure in C occurs internally- because it contains separate memory locations allotted to every input member.	In the case of a Union, the memory allocation occurs for only one member with the largest size among all the input variables. It shares the same location among all these members/objects.
Accessing Members	A user can access individual members at a given time.	A user can access only one member at a given time.
Syntax	The Syntax of declaring a Structure in C is: struct [structure name] { type element_1;	The Syntax of declaring a Union in C is: union [union name] { type element_1; type element_2;

	type element_2; . . } variable_1, variable_2, …;	. . } variable_1, variable_2, …;
Size	A Structure does not have a shared location for all of its members. It makes the size of a Structure to be greater than or equal to the sum of the size of its data members.	A Union does not have a separate location for every member in it. It makes its size equal to the size of the largest member among all the data members.
Value Altering	Altering the values of a single member does not affect the other members of a Structure.	When you alter the values of a single member, it affects the values of other members.
Storage of Value	In the case of a Structure, there is a specific memory location for every input data member. Thus, it can store multiple values of the various members.	In the case of a Union, there is an allocation of only one shared memory for all the input data members. Thus, it stores one value at a time for all of its members.
Initialization	In the case of a Structure, a user can initialize multiple members at the same time.	In the case of a Union, a user can only initiate the first member at a time.

COURSE: COMPUTER PROGRAMING

UNIT 7: OBJECT ORIENTED PROGRAMMING CONCEPT

7.1 Need of an Object-Oriented Programming

7.2 C++ and its Applications

7.3 OOPs Concepts in C++

7.3.1 Class

7.3.2 Object

7.3.3 Encapsulation

7.3.4 Abstraction

7.3.5 Polymorphism

7.3.6 Inheritance

7.3.7 Dynamic Binding and Message Passing

7.3.8 Access Specifiers in C++

7.3.8.1 Public

7.3.8.2 Private

7.3.8.3 Protected

7.1 Need of an Object-Oriented Programming

The earlier approaches to programming were not that good, and there were several limitations as well. Like in procedural-oriented programming, you cannot reuse the code again in the program, and there was the problem of global data access, and the approach couldn‘t solve the real-world problems very well.

In object-oriented programming, it is easy to maintain the code with the help of classes and objects. Using inheritance, there is code reusability, i.e., you don‘t have to write the same code again and again, which increases the simplicity of the program. Concepts like encapsulation and abstraction provide data hiding as well.

Problems solved by object-oriented approach

- **Code reusability**: No need to write the same code again and again, which increases the program‘s simplicity. As the same code can be executed many times in the program.
- **Data hiding**: Data can be hidden from the outside world.

7.2 C++ and its Application:

C++ is a special-purpose programming language developed by **Bjarne Stroustrup** at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

Applications of C++ Programming

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX‘s C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like softwares for MRI machines, high-end CAD/CAM systems etc.

7.3 OOPs Concepts in C++

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language. Object-oriented programming, as the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can

access this data except that function. There are some basic concepts that act as the building blocks of OOPs in C++ named as Class, Objects, Encapsulation, Abstraction, Polymorphism, Inheritance, Dynamic Binding and Message Passing, the description of each is given below:

7.3.1 Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

We can say that a Class in C++ is a blueprint representing a group of objects which shares some common properties and behaviors.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows :

```
class Box
  {
  public:
    double length;  // Length of a box
    double breadth;  // Breadth of a box
    double height;  // Height of a box
    };
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object.

7.3.2 Objects

An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members: The public data members of objects of a class can be accessed using the direct member access operator (.).

Example:

```
#include <iostream.h>
class Box
    {
    public:
     double length;  // Length of a box
     double breadth;  // Breadth of a box
     double height;  // Height of a box
     };
int main() {
  Box Box1;       // Declare Box1 of type Box
  Box Box2;       // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
  // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;
// box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;
  // volume of box 1
```

```
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
 }
```

Output:

Volume of Box1:210

Volume of Box2: 1560

NOTE: Private and Protected members cannot be accessed directly using direct member access operator (.)

7.3.3 Encapsulation

In normal terms, Encapsulation is defined as wrapping up data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name ―sales section‖.

Features of Encapsulation

8 We cannot access any function from the class directly. We need an object to access that function that is using the member variables of that class.

9 The function which we are making inside the class must use only member variables, only then it is called encapsulation.

10 If we don‘t make a function inside the class which is using the member variable of the class then we don‘t call it encapsulation.

11 Increase in the security of data, to restrict and control the modification of our data members.

In C++, encapsulation can be implemented using classes and access modifiers. The following example explain the concept of Encapsulation:

```
// Encapsulation
#include <iostream>
using namespace std;

class Encapsulation {
private:
    // Data hidden from outside world
    int x;

public:
    // Function to set value of
    // variable x
    void set(int a) { x = a; }

    // Function to return value of
    // variable x
    int get() { return x; }
};

// Driver code
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

Output: 5

In the above program, the variable x is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

7.3.4 Abstraction:

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.

- **Abstraction using Classes**: We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files**: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Features of Data Abstraction:

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.

- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

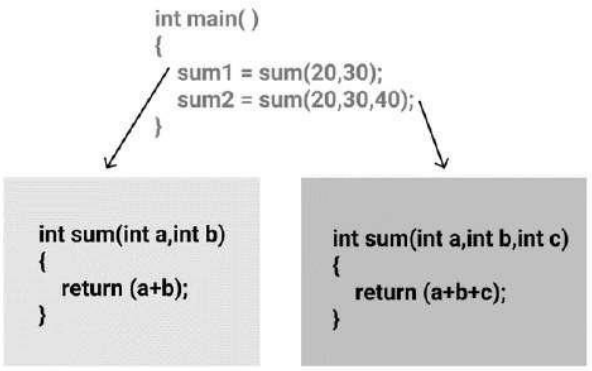
In simple words, Abstraction avoids unnecessary information or irrelevant details and shows only that specific part which the user wants to see.

7.3.5 Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

- Operator Overloading: The process of making an operator exhibit different behaviors in different instances is known as operator overloading.
- Function Overloading: Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, sometimes there are 2 integers, and sometimes there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



7.3.6 **Inheritance**

Inheritance supports the concept of ―reusability‖, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class. Inheritance allows to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

8 **Base and Derived Classes:** A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Access Control in Inheritance: A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class. We can summarize the different access types according to - who can access them in the following way:

Access	**Public**	**protected**	**private**
Same class	Yes	yes	yes
Derived classes	Yes	yes	no
Outside classes	Yes	no	no

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above. While using different type of inheritance, following rules are applied :

9 **Public Inheritance**: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

10 **Protected Inheritance**: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

11 **Private Inheritance**: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Types Of Inheritance: C++ supports five types of inheritance.

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

The details about each section discussed in later sections.

7.3.7 Dynamic Binding and Message Passing

In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has virtual functions to support this. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

Example:

```
// C++ Program to Demonstrate the Concept of Dynamic binding
// with the help of virtual function
#include <iostream>
using namespace std;
class GFG
{
public:
```

```
    void call_Function() // function that call print
    {
        print();
    }
    void print() // the display function
    {
        cout << "Printing the Base class Content" << endl;
    }
};
class GFG2 : public GFG // GFG2 inherit a publicly
{
public:
    void print() // GFG2's display
    {
        cout << "Printing the Derived class Content"
             << endl;
    }
  };
int main()
  {
    GFG geeksforgeeks; // Creating GFG's pbject
    geeksforgeeks.call_Function(); // Calling call_Function
    GFG2 geeksforgeeks2; // creating GFG2 object
    geeksforgeeks2.call_Function(); // calling call_Function
                                    // for GFG2 object
    return 0;
    }
```

Output:

```
Printing the Base class Content
Printing the Base class Content
```

As we can see, the print() function of the parent class is called even from the derived class object. To resolve this we use virtual functions.

- **Message Passing**

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function, and the information to be sent. A message cannot go automatically it creates an interface, which means it creates an interface for an object. The interface provides the abstraction over the message means hide the implementation. So we get to know, An interface is a set of operations that a given object can perform.

All communication between objects is done via message is called message passing, as people exchange information similarly the sending and receiving of information by the object is said to be message passing, to make possible message passing the following things have to be followed to be done :

- User have to create classes that define objects and its behavior.
- Then Creating the objects from class definitions.
- Calling and connecting the communication among objects.

7.3.8 Access Specifiers in C++

Data hiding is an important concept of Object-Oriented Programming, implemented with these Access modifiers' help. It is also known as Access Specifier. Access Specifiers in a class decide the accessibility of the class members, like variables or methods in other classes. That is, it will decide whether the members or methods will get directly accessed by the blocks present outside the class or not, depending on the type of Access Specifier. In a program, we need to create methods or variables that can be accessed by the object of the same class or accessible in the entire program. And Access Modifiers help us to specify that. There are three types of access modifiers in C++:

- Public
- Private
- Protected

To manipulate and fetch the data, a public specifier is used, and to protect the data from outside members, a private specifier is used so that the crucial or sensitive data cannot be tampered with or leaked outside of its block.

Syntax of Declaring Access Specifiers in C++ is given below:

```
class ClassName
{
private:
// Declare private members/methods here.
public:
// Declare public members/methods here.
protected:
// Declare protected members/methods here.
};
```

The following section describes about each access specifier in detail:

7.3.8.1 Public: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// C++ program to demonstrate public
// access modifier
#include<iostream>
using namespace std;
// class definition
class Circle
{
   public:
  double radius;
 double  compute_area()
  {
  return 3.14*radius*radius;
  }
};
// main function
int main()
{
```

```
    Circle obj;
    // accessing public datamember outside class
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

Radius is: 5.5

Area is: 94.985

In the above example, the data member radius is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

7.3.8.2 Private: The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
        double  compute_area()
        {   // member function can access private
            // data member radius
            return 3.14*radius*radius;
```

```
    }
};
// main function
int main()
{
   // creating object of the class
   Circle obj;
   // trying to access private data member
   // directly outside the class
   obj.radius = 1.5;
   cout << "Area is:" << obj.compute_area();
   return 0;
}
```

Output:

```
In function 'int main()':
error: 'double Circle::radius' is private
double radius;
          ^
error: within this context
obj.radius = 1.5;
          ^
```

The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius is attempted, but radius being a private data member, we obtained the above compilation error. However, we can access the private data members of a class indirectly using the public member functions of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
   // private data member
```

```
    private:
        double radius;
    // public member function
    public:
        void compute_area(double r)
        {   // member function can access private
            // data member radius
            radius = r;
            double area = 3.14*radius*radius;
            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }
};
// main function
int main()
{
    // creating object of the class
    Circle obj;
    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);
    return 0;
}
```

Output:

```
Radius is: 1.5
Area is: 7.065
```

7.3.8.3 Protected: The protected access modifier is similar to the private access modifier in the sense that it can‘t be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well. This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

Example:

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;
// base class
class Parent
{
   // protected data members
   protected:
   int id_protected;
};
// sub class or derived class from public base class
class Child : public Parent
{
   public:
   void setId(int id)
   {
      // Child class is able to access the inherited
      // protected data members of base class
      id_protected = id;
   }
   void displayId()
   {
      cout << "id_protected is: " << id_protected << endl;
   }
};
// main function
int main()
{
   Child obj1;
   // member function of the derived class can
   // access the protected data members of the base class
   obj1.setId(81);
```

```
    obj1.displayId();
    return 0;
}
```

Output:

```
id_protected is: 81
```

\

COURSE: COMPUTER PROGRAMING

UNIT 8: CONSTRUCTOR AND INHERITANCE IN C++

8.1 Constructor in C++

8.2 Characteristics of Constructors

8.3 Difference between constructor and member function

8.4 Types of Constructor

8.4.1 Default Constructor

8.4.2 Parameterized Constructors

8.4.3 Copy Constructors

8.4.4 Dynamic Constructor

8.5 Destructor in C++

8.6 Difference between Constructor and Destructor in C++:

8.7 Inheritance in C++

8.8 Modes of Inheritance

8.9 Types of Inheritance

8.9.1 Single Inheritance

8.9.2 Multilevel Inheritance

8.9.3 Multiple Inheritance

8.9.4 Hierarchical Inheritance

8.9.5 Hybrid Inheritance

8.1 Constructor in C++

While programming, sometimes there might be the need to initialize data members and member functions of the objects before performing any operations. Data members are the variables declared in any class by using fundamental data types (like int, char, float, etc.) or derived data types (like class, structure, pointer, etc.). The functions defined inside the class definition are known as member functions. Suppose you are developing a game. In that game, each time a new player registers, we need to assign their initial location, health, acceleration, and certain other quantities to some default value. This can be done by defining separate functions for each quantity and assigning the quantities to the required default values. For this, we need to call a list of functions every time a new player registers. Now, this process

can become lengthy and complicated.

Therefore, C++ provides feature of constructor to do the same, a constructor is a member function of a class that has the same name as the class name. It helps to initialize the object of a class. It can either accept the arguments or not. It is used to allocate the memory to an object of the class. It is called whenever an instance of the class is created. It can be defined manually with arguments or without arguments. There can be many constructors in a class. It can be overloaded but it cannot be inherited or virtual. There is a concept of copy constructor which is used to initialize an object from another object.

Syntax of Constructor:

```
class class_name
{
  private:
  // private members
  public:
  // declaring constructor
  class_name({parameters})
  {
    // constructor body
  }
};
```

In the above syntax, we can see the class has the name class_name and the constructor have also the same name. A constructor can have any number of parameters as per requirements. Also, there is no return type or return value of the constructor.

8.2 Characteristics of Constructors in C++:

- A constructor can be made public, private, or protected per our program's design. Constructors are mostly made public, as public methods are accessible from everywhere, thus allowing us to create the object of the class anywhere in the code. When a constructor is made private, other classes cannot create instances of the class. This is used when there is no need for object creation. Such a case arises when the class only contains static member functions (i.e., the functions which are independent of any class object and can be accessed using a class name with scope resolution operator).

- A constructor in C++ cannot be inherited. However, a derived class can call the base class constructor. A derived class (i.e., child class) contains all members and member functions (including constructors) of the base class.
- Constructor functions are not inherited, and their addresses cannot be referenced.

8.3 Difference between constructor and member function:

- Constructor name must be the same as class name but functions cannot have the same name as the class name.
- Constructors do not have a return type whereas functions must have a return type.
- Constructors are automatically called when an object is created.
- A member function can be virtual, but there is no concept of virtual constructors.
- Constructors are invoked at the time of object creation automatically and cannot be called explicitly using class objects.

8.4 Types of Constructor in C++

There are four types of constructors in C++, Default Constructors, Parameterized Constructors, Copy Constructors, Dynamic Constructors. The detail of each is given below:

8.4.1 Default Constructor: Default constructor is also known as a zero-argument constructor, as it doesn‘t take any parameter. It can be defined by the user if not then the compiler creates it on his own. Default constructor always initializes data members of the class with the same value they were defined.

Syntax of Default Constructor:

```
class class_name
{
  private:
  // private members

  public:

  // declaring default constructor
  class_name()
  {
    // constructor body
```

```
  }

};
```

Example of default constructor:

```
#include <iostream>
using namespace std;
class Person
{
  // declaring private class data members
private:
  string name;
  int age;

public:

  // declaring constructor
  Person()
  {
      cout<<"Default constructor is called"<<endl;
      name = "student";
      age = 12;
  }
  // display function to print the class data members value
  void display()
  {
      cout<<"Name of current object: "<<name<<endl;
      cout<<"Age of current object: "<<age<<endl;
  }
};
int main()
{
    // creating object of class using default constructor
    Person obj;
    // printing class data members
```

```
    obj.display();
    return 0;
}
```

Output:

Default constructor is called
Name of current object: student
Age of current object: 12

In the above code, we have created a class with two data members. Declared a default constructor which always initializes objects of a class with the same name and age. In the main function, we have created an object of the class and printed its data member values by using the display function.

8.4.2 Parameterized Constructor:

Parameterized constructor is used to initialize data members with the values provided by the user. This constructor is basically the upgraded version of the default constructor. We can define more than one parameterized constructor according to the need of the user, but we have to follow the rules of the function overloading, like a different set of arguments must be there for each constructor.

Syntax

```
class class_name{
  private:
  // private members

  public:

  // declaring parameterized constructor
  class_name(parameter1, parameter2,...)
  {
    // constructor body
  }
  };
```

Code to understand the working of the parameterized constructor

```
#include  <iostream>
using namespace std;
class Person
{
  // declaring private class data members
private:
  string name;
int age;
public:

  // declaring parameterized constructor of three different types
  Person(string person_name)
  {
     cout<<"Constructor to set name is called"<<endl;
     name = person_name;
     age = 12;
  }
  Person(int person_age)
  {
     cout<<"Constructor to set age is called"<<endl;
     name = "Student";
     age = person_age;
  }
  Person(string person_name, int person_age)
  {
     cout<<"Constructor for both name and age is called"<<endl;
     name = person_name;
     age = person_age;
  }
  // display function to print the class data members value
  void display()
  {
     cout<<"Name of current object: "<<name<<endl;
```

```
    cout<<"Age of current object: "<<age<<endl;
    cout<<endl;
   }
  };
int main()
   {
   // creating objects of class using parameterized constructor
   Person obj1("First person");
   // printing class data members for first object
   obj1.display();
   Person obj2(25);
   // printing class data members for second object
   obj2.display();
   Person obj3("Second person",15);
   // printing class data members for third object
   obj3.display();
   return 0;
   }
```

Output:

```
Constructor to set name is called
Name of current object: First person
Age of current object: 12

Constructor to set age is called
Name of current object: Student
Age of current object: 25

Constructor for both name and age is called
Name of current object: Second person
Age of current object: 15
```

In the above code, we have created three types of the parametric constructor, one for initialization of name only, second to initialization of age only, and third to initialize both name and age. In the main function, we have created three different types of objects and initialized them in different ways, and printed values for each of them.

8.4.3 Copy Constructor

If we have an object of a class and we want to create its copy in a new declared object of the same class, then a copy constructor is used. The compiler provides each class a default copy constructor and users can define it also. It takes a single argument which is an object of the same class.

Syntax of Copy Constructor:

```
class class_name
{
        private:
        // private members
        public:
        // declaring copy constructor
class_name(const class_name& obj)
{
        // constructor body
}
};
```

In the above syntax, we created a copy constructor which takes an object of the same class as a parameter but it is declared constant and passed as a reference because when an argument is passed as a function parameter it creates a copy for it, to create that copy compiler will again call the copy constructor, means it will call the same function and for that call again there will be a call to create copy which will take this process in never ending recursion of creating copies. To prevent such conditions we pass it as a reference.

Example of copy constructor:

```
#include <iostream>
using namespace std;
class Person
{
  // declaring private class data members
private:
  string name;
int age;
public:
  Person(string person_name, int person_age)
```

```
    {
      cout<<"Constructor for both name and age is called"<<endl;
      name = person_name;
      age = person_age;
    }
  Person(const Person& obj)
    {
      cout<<"Copy constructor is called"<<endl;
      name = obj.name;
      age = obj.age;
    }
  // display function to print the class data members value
  void display()
    {
      cout<<"Name of current object: "<<name<<endl;
      cout<<"Age of current object: "<<age<<endl;
      cout<<endl;
  }
  };
int main()
  {
    // creating objects of class using parameterized constructor
    Person obj1("First person",25);
    // printing class data members for first object
    obj1.display();
    // creating copy of the obj1
    Person obj2(obj1);
    // printing class data members for second object
    obj2.display();
    return 0;
   }
```

Output:

```
Constructor for both name and age is called
Name of current object: First person
```

Age of current object: 25

Copy constructor is called
Name of current object: First person
Age of current object: 25

In the above code, we have created a class and defined two types of constructors in it, the first is a parameterized constructor and another is a copy constructor. Parameterized constructor is used to create an object then by using the copy constructor we create a copy of it and stored it in another object.

8.4.4 Dynamic Constructor

When memory is allocated dynamically to the data members at the runtime using a new operator, the constructor is known as the dynamic constructor. This constructor is similar to the default or parameterized constructor; the only difference is it uses a new operator to allocate the memory.

Syntax of Dynamic Constructor:

```
class class_name
{
  private:
  // private members
  public:
  // declaring dynamic constructor
  class_name({parameters})
  {
    // constructor body where data members are initialized using new operator
  }
  };
```

Example of dynamic constructor:

```
#include <iostream>
using namespace std;
class Person
{
  // declaring private class data members
private:
  int* age;
```

```
public:
  Person(int* person_age)
  {
     cout<<"Constructor for age is called"<<endl;
     // allocating memory
     age = new int;
     age = person_age;
  }
  // display function to print the class data members value
  void display()
  {
     cout<<"Age of current object: "<<*age<<endl;
     cout<<endl;
  }
  };
int main()
  {
    // creating objects of class using parameterized constructor
    int age = 25;
    Person obj1(&age);
    // printing class data members for first object
    obj1.display();
    return 0;
   }
```

Output:

```
Constructor for age is called
Age of current object: 25
```

In the above code, we have created a class with a dynamic constructor. In the main function, we have created an object and initialized it using a dynamic constructor, where we have given memory dynamically using a new operator.

8.8 Destructor in C++

Destructor is just the opposite function of the constructor. A destructor is called by the compiler when the object is destroyed and its main function is to deallocate the memory of the object. The object may be destroyed when the program ends, or local objects of the

function get out of scope when the function ends or in any other case. Destructor has the same as of the class with prefix tilde(~) operator and it cannot be overloaded as the constructor. Destructors take no argument and have no return type and return value.

Syntax of the Destructor:

```
class class_name
 {
 private:
 // private members
 public:
 // declaring destructor
 ~class_name()
 {
  // destructor body
  }
  };
```

In the above syntax, we can see the class has the name class_name and the destructor also has the same name, in addition there is a tilde(~). Also, there is no return type and return value of the destructor.

Important Points about the Destructor

- Destructor are the last member function called for an object and they are called by the compiler itself.
- If the destructor is not created by the user then compile creates or declares it by itself.
- A Destructor can be declared in any section of the class, as it is called by the compiler so nothing to worry about.
- As Destructor is the last function to be called, it should be better to declare it at the end of the class to increase the readability of the code.
- Destructor is just the opposite of the constructor as the constructor is called at the time of the creation of the object and allocates the memory to the object, on the other side the destructor is called at the time of the destruction of the object and deallocates the memory.

8.6 Difference between Constructor and Destructor in C++:

A constructor allows an object to initialize some of its value before it is used. A destructor allows an object to execute some code at the time of its destruction. The following example

demonstrate the concept of Constructor and Destructor in C++.

Example/Implementation of Constructor and Destructor

```
#include <iostream>
using namespace std;
class Z
{
public:
   // constructor
   Z()
   {
      cout<<"Constructor called"<<endl;
   }
   // destructor
   ~Z()
   {
      cout<<"Destructor called"<<endl;
   }
   };
int main()
   {
   Z z1;  // Constructor Called
   int a = 1;
   if(a==1)
   {
      Z z2;  // Constructor Called
   }  // Destructor Called for z2
    } //  Destructor called for z1
```

Output:

```
Constructor called
Constructor called
Destructor called
Destructor called
```

The following table also differentiate the concept of Constructor and Destructor.

Sr. No.	Constructor	Destructor
1.	Constructor helps to initialize the object of a class.	Whereas destructor is used to destroy the instances.
2.	It is declared as **className(arguments if any){Constructor's Body }.**	Whereas it is declared as **~ className(no arguments){ }.**
3.	Constructor can either accept arguments or not.	While it can't have any arguments.
4.	A constructor is called when an instance or object of a class is created.	It is called while object of the class is freed or deleted.
5.	Constructor is used to allocate the memory to an instance or object.	While it is used to deallocate the memory of an object of a class.
6.	Constructor can be overloaded.	While it can't be overloaded.
7.	The constructor's name is same as the class name.	Here, its name is also same as the class name preceded by the tiled (~) operator.

8.7 Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming. Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called ―derived class‖ or ―child class‖ and the existing class is known as the ―base class‖ or ―parent class‖. The derived class now is said to be inherited from the base class. When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

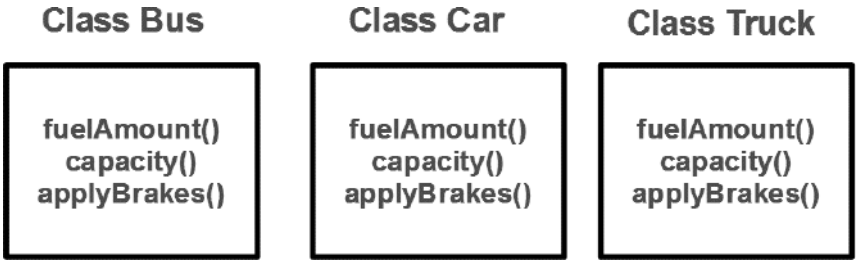
- **Sub Class**: The class that inherits properties from another class is called Subclass or Derived Class.

- **Super Class**: The class whose properties are inherited by a subclass is called Base Class or Superclass.

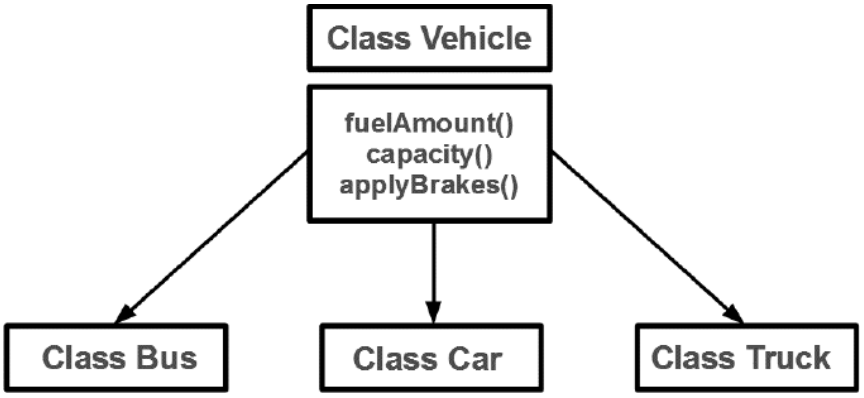
Reusability using Inheritance

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Example: Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



The above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Syntax:

```
class  <derived_class_name> : <access-specifier> <base_class_name>
    {
    //body
     }
```

Note: A derived class doesn‘t inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Example of Inheritance:

```
class ABC : private XYZ          //private derivation
{                          }
class ABC : public XYZ           //public derivation
{                          }
class ABC : protected XYZ        //protected derivation
{                          }
class ABC: XYZ                    //private derivation by default
{                          }
```

8.8 Modes of Inheritance

There are three modes of inheritance that is publicly, privately, and protected. If we are not writing any access specifiers then by default it becomes private.

- **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
- **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
- **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class,

while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
// C++ Implementation to show that a derived class
// doesn‘t inherit access to private data members.
// However, it does inherit a full parent object.
class A
{
public:
   int x;
protected:
   int y;
private:
    int z;
};
class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};
class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};
class D : private A // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the

members of the base class in the subclass when derived in public, protected and private modes:

Base Class Member Access Specifier	Mode of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

8.9 Types of Inheritance

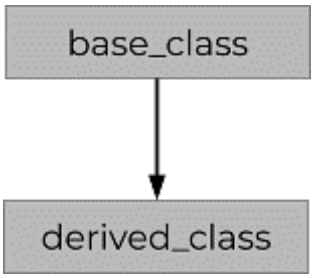
During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

The detail of each type is given below:

8.9.1 Single Inheritance

Single Inheritance is the most primitive among all the types of inheritance in C++. In this inheritance, a single class inherits the properties of a base class. All the data members of the base class are accessed by the derived class according to the visibility mode (i.e., private, protected, and public) that is specified during the inheritance.



Syntax of Single Inheritance is given below:

```
class base_class_1
{
```

```
    // class definition
};
class derived_class: visibility_mode base_class_1
{
    // class definition
};
```

Description: A single derived_class inherits a single base_class. The visibility_mode is specified while declaring the derived class to specify the control of base class members within the derived class.

Given below is a complete Example of Single Inheritance.

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
  string name="";
  public:
  int tail=1;
  int legs=4;
};
class Dog : public Animal
{
  public:
  void voiceAction()
  {
    cout<<"Barks!!!";
  }
  };
int main()
  {
  Dog dog;
  cout<<"Dog has "<<dog.legs<<" legs"<<endl;
  cout<<"Dog has "<<dog.tail<<" tail"<<endl;
```

```
  cout<<"Dog ";
  dog.voiceAction();
}
```

Output:

Dog has 4 legs

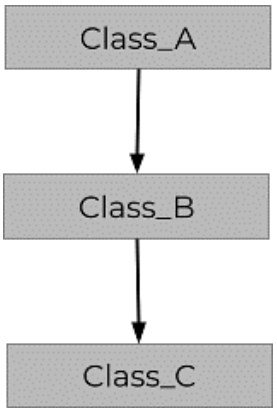
Dog has 1 tail

Dog Barks!!!

Explanation: We have a class Animal as a base class from which we have derived a subclass dog. Class dog inherits all the members of the Animal class and can be extended to include its own properties, as seen from the output.

8.9.2 Multilevel Inheritance

The inheritance in which a class can be derived from another derived class is known as Multilevel Inheritance. Suppose there are three classes A, B, and C. A is the base class that derives from class B. So, B is the derived class of A. Now, C is the class that is derived from class B. This makes class B, the base class for class C but is the derived class of class A. This scenario is known as the Multilevel Inheritance. The data members of each respective base class are accessed by their respective derived classes according to the specified visibility modes.



Syntax of Multilevel Inheritance is given below:

```
class class_A
{
   // class definition
};
```

```
class class_B: visibility_mode class_A
{
   // class definition
};
class class_C: visibility_mode class_B
{
   // class definition
};
```

Description: The class_A is inherited by the sub-class class_B. The class_B is inherited by the subclass class_C. A subclass inherits a single class in each succeeding level.

Example of Multilevel Inheritance is given below:

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
  string name="";
  public:
  int tail=1;
  int legs=4;

};
class Dog : public Animal
{
  public:
  void voiceAction()
  {
    cout<<"Barks!!!";
  }
};
class Puppy:public Dog{
  public:
  void weeping()
  {
```

```
    cout<<"Weeps!!";
  }
};
int main()
{
  Puppy puppy;
cout<<"Puppy has "<<puppy.legs<<" legs"<<endl;
cout<<"Puppy has "<<puppy.tail<<" tail"<<endl;
cout<<"Puppy ";
puppy.voiceAction();
cout<<" Puppy ";
puppy.weeping();
}
```

Output:

Puppy has 4 legs

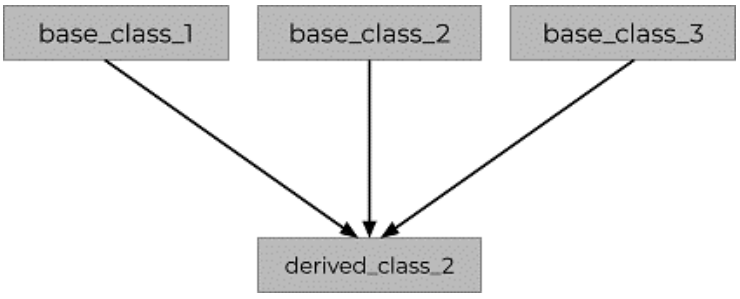
Puppy has 1 tail

Puppy Barks!!! Puppy Weeps!!

Here we modified the example for Single inheritance such that there is a new class Puppy which inherits from the class Dog that in turn inherits from class Animal. We see that the class Puppy acquires and uses the properties and methods of both the classes above it.

8.9.3 Multiple Inheritance

The inheritance in which a class can inherit or derive the characteristics of multiple classes, or a derived class can have over one base class, is known as Multiple Inheritance. It specifies access specifiers separately for all the base classes at the time of inheritance. The derived class can derive the joint features of all these classes and the data members of all the base classes are accessed by the derived or child class according to the access specifiers.



Syntax of Multiple Inheritance is given below:

```
class base_class_1
{
   // class definition
};
class base_class_2
{
   // class definition
};
class derived_class: visibility_mode_1 base_class_1, visibility_mode_2 base_class_2
{
   // class definition
};
```

Description: The derived_class inherits the characteristics of two base classes, base_class_1 and base_class_2. The visibility_mode is specified for each base class while declaring a derived class. These modes can be different for every base class.

Example of Multiple Inheritance is given below:

```
#include <iostream>
using namespace std;
//multiple inheritance example
class student_marks {
protected:
int rollNo, marks1, marks2;
public:
void get()
{
cout << "Enter the Roll No.: "; cin >> rollNo;
cout << "Enter the two highest marks: "; cin >> marks1 >> marks2;
   }
};
class cocurricular_marks
{
protected:
int comarks;
```

```
public:
void getsm() {
cout << "Enter the mark for CoCurricular Activities: "; cin >> comarks;
}
};
//Result is a combination of subject_marks and cocurricular activities marks
class Result : public student_marks, public cocurricular_marks
{
  int total_marks, avg_marks;
  public:
  void display()
{
    total_marks = (marks1 + marks2 + comarks);
    avg_marks = total_marks / 3;
    cout << "\nRoll No: " << rollNo << "\nTotal marks: " << total_marks;
    cout << "\nAverage marks: " << avg_marks;
  }
};
int main()
{
  Result res;
res.get(); //read subject marks
res.getsm(); //read cocurricular activities marks
res.display(); //display the total marks and average marks
}
```

Output:

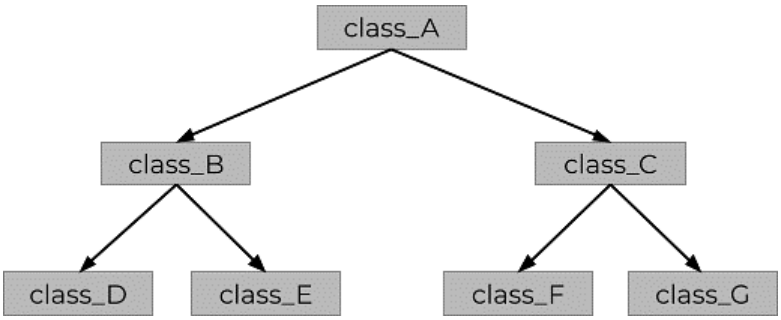
```
Enter the Roll No.: 25
Enter the two highest marks: 40 50
Enter the mark for CoCurricular Activities: 30

Roll No: 25
Total marks: 120
Average marks: 40
```

Explanation: In the above example, we have three classes i.e. student_marks, cocurricular_marks, and Result. The class student_marks reads the subject mark for the student. The class cocurricular_marks reads the student‘s marks in co-curricular activities.

8.9.4 Hierarchical Inheritance

The inheritance in which a single base class inherits multiple derived classes is known as the Hierarchical Inheritance. This inheritance has a tree-like structure since every class acts as a base class for one or more child classes. The visibility mode for each derived class is specified separately during the inheritance and it accesses the data members accordingly.



Syntax of Hierarchical Inheritance is given below:

```
class class_A
{
   // class definition
};
class class_B: visibility_mode class_A
{
   // class definition
};
class class_C : visibility_mode class_A
{
   // class definition
};
class class_D: visibility_mode class_B
{
   // class definition
};
```

```
class class_E: visibility_mode class_C
{
   // class definition
};
```

Description: The subclasses class_B and class_C inherit the attributes of the base class class_A. Further, these two subclasses are inherited by other subclasses class_D and class_E respectively.

Example of Hierarchical Inheritance is given below:

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// first sub class
class Car : public Vehicle {
};
// second sub class
class Bus : public Vehicle {
};
// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Car obj1;
   Bus obj2;
   return 0;
}
```

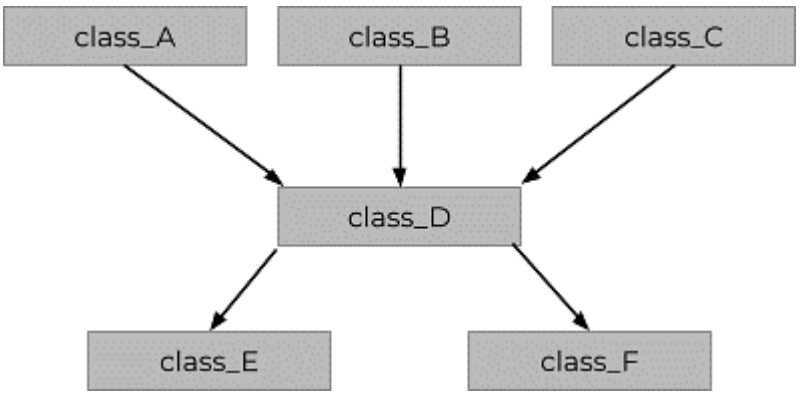
Output:

```
This is a Vehicle
```

This is a Vehicle

8.9.5 Hybrid Inheritance

Hybrid Inheritance, as the name suggests, is the combination of two or over two types of inheritances. For example, the classes in a program are in such an arrangement that they show both single inheritance and hierarchical inheritance at the same time. Such an arrangement is known as the Hybrid Inheritance. This is arguably the most complex inheritance among all the types of inheritance in C++. The data members of the base class will be accessed according to the specified visibility mode.



Syntax of Hybrid Inheritance is given Below:

```
class class_A
{
   // class definition
};
class class_B
{
   // class definition
};
class class_C: visibility_mode class_A, visibility_mode class_B
{
   // class definition
};
class class_D: visibility_mode class_C
{
   // class definition
```

```
};
class class_E: visibility_mode class_C
{
   // class definition
};
```

Description: The derived class class_C inherits two base classes that are, class_A and class_B. This is the structure of Multiple Inheritance. And two subclasses class_D and class_E, further inherit class_C. This is the structure of Hierarchical Inheritance. The overall structure of Hybrid Inheritance includes more than one type of inheritance.

Example of Hybrid Inheritance is given below:

```
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};
// first sub class
class Car : public Vehicle {
};
// second sub class
class Bus : public Vehicle, public Fare {
};
// main function
int main()
{
   // Creating object of sub class will
```

```
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```