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

Bachelor of Arts

Course Name: Fundamentals of Programming Languages

Course Code: BAB32308T

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

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PROGRAMME COORDINATOR

Dr. Pinky Sra (Assistant Professor)
School of Social Sciences and Liberal Arts
JGND PSOU, Patiala

COURSE COORDINATOR AND EDITOR:

Dr. Karan Sukhija (Assistant Professor)
School of Sciences and Emerging Technologies
JGND PSOU, Patiala

COURSE OUTCOMES:

- An ability to design, implement, and evaluate a computer-based system, process, components, or program to meet desired needs.
- An ability to analyse a problem, and identify and define the computing requirements appropriate to its solution.
- Understanding of code organization and functional hierarchical decomposition with using complex data types.
- Understanding a concept of object thinking within the framework of functional model.
- Ability to work with arrays of complex objects.

JAGAT GURU NANAK DEV
PUNJAB STATE OPEN UNIVERSITY PATIALA
(Established by Act No.19 of 2019 of Legislature of the State of Punjab)

PREFACE

Jagat Guru Nanak Dev Punjab State Open University, Patiala was established in December 2019 by Act 19 of the Legislature of State of Punjab. It is the first and only Open University of the State, entrusted with the responsibility of making higher education accessible to all especially to those sections of society who do not have the means, time or opportunity to pursue regular education.

In keeping with the nature of an Open University, this University provides a flexible education system to suit every need. The time given to complete a programme is double the duration of a regular mode programme. Well-designed study material has been prepared in consultation with experts in their respective fields.

The University offers programmes which have been designed to provide relevant, skill-based and employability-enhancing education. The study material provided in this booklet is self-instructional, with self-assessment exercises, and recommendations for further readings. The syllabus has been divided in sections, and provided as units for simplification.

The Learner Support Centres/Study Centres are located in the Government and Government aided colleges of Punjab, to enable students to make use of reading facilities, and for curriculum-based counselling and practicals. We, at the University, welcome you to be a part of this institution of knowledge.

Dean Academic Affairs

Name of Programme: Bachelor of Arts
Name of Course: Fundamentals of Programming Languages
Course Code: BAB32308T

MAX MARKS:100
EXTERNAL:70
INTERNAL:30
PASS:40%
Credits: 4

INSTRUCTIONS FOR THE PAPER SETTER/EXAMINER:

- The syllabus prescribed should be strictly adhered to.
- The question paper will consist of three sections: A, B, and C. Sections A and B will have our questions each from the respective sections of the syllabus and will carry 10 marks each. The candidates will attempt two questions from each section.
- Section C will have fifteen short answer questions covering the entire syllabus. Each question will carry 3 marks. Candidates will attempt any 10 questions from this section.
- The examiner shall give a clear instruction to the candidates to attempt questions only at one place and only once. Second or subsequent attempts, unless the earlier ones have been crossed out, shall not be evaluated.
- The duration of each paper will be three hours.

INSTRUCTIONS FOR THE CANDIDATES:

Candidates are required to attempt any two questions each from the sections A, and B of the question paper, and any ten short answer questions from Section C. They have to attempt questions only at one place and only once. Second or subsequent attempts, unless the earlier ones have been crossed out, shall not be evaluated.

Course Outcomes:

CO 1	**To understand the fundamentals of programming language.**
CO 2	Learner will be able to implement various algorithms and develops the basic concepts and terminology of programming in general.
CO 3	To understand the syntax and semantic of programming language in terms of procedural oriented and object oriented.

CO 4	To understand the concepts of modular programming in general, along with the advanced used of OOPs to more emphasis on data rather than functions.
CO 5	Learner will be able to develop embedded system to be used as real time applications.

Section A

Module 1: Basics of Programming: 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: Arithmetic, Relational, Logical, Comma, Conditional, Assignment, Operator Precedence and Associativity in C, Input and Output Statements, Assignment statements.

Module 2: Control Structure in C: 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.

Section B

Module 3: Basics of oops: Basic concepts (objects, classes, inheritance, polymorphism, encapsulation), Advantages of OOP over procedural programming, Classes and Objects: Declaring classes, creating objects, Access specifiers (public, private, protected), Constructors and destructors, Static members.

Module 4: Inheritance and Polymorphism: Base and derived classes, Types of inheritance (single, multiple, multilevel, hierarchical), Access control in inheritance, Function overloading, Operator overloading, Virtual functions and runtime polymorphism, Abstract classes and pure virtual functions.

Suggested Reading:

- Kanetkar, Yashavant. Let us C. BPB publications, 2018.
- Kamathane, Programming in C, Oxford University Press.
- E. Balagurusamy, Programming in C, Tata McGraw-Hill.
- E. Balagurusamy “Programming with Java”, TMH

SECTION- A

Module 1: Basics of Programming

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: Arithmetic, Relational, Logical, Comma, Conditional, Assignment, Operator Precedence and Associativity in C, Input and Output Statements, Assignment statements.

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

1.2 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

1.3 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.

1.3.1 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

1.3.2 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.

1.3.3 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.

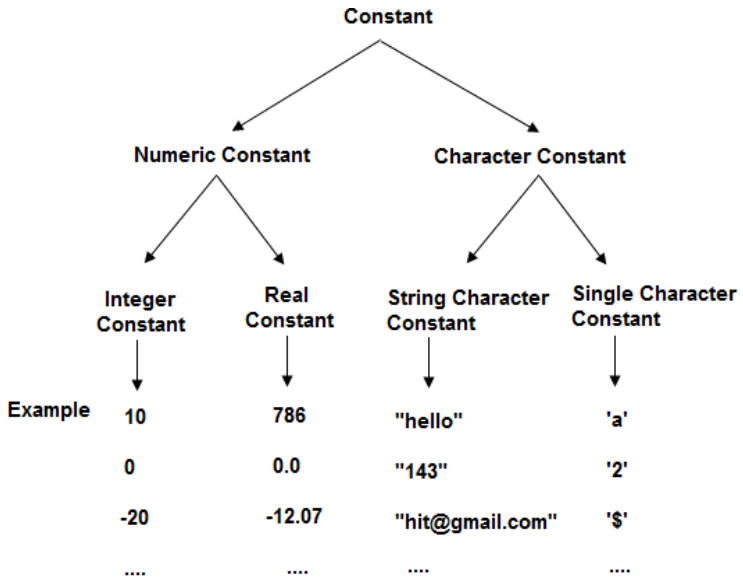
- **Using const keyword:** The 'const' keyword is used to create a constant of any given datatype in a program. For creating a constant, there is need to prefix the declaration of the variable with the 'const' keyword.
  Syntax: const datatype constant_Name = value;
  Example: const float PI=3.14**;**

- **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 identifier_Name 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 Constants 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 Constants (base 10)
  - Octal Constants (base 8)
  - Hexadecimal Constants (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

1.3.4 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

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)

1.4 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

1.4.1 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

1.4.2 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;
}
```

1.4.3 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

1.5 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.

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

1.5.1 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
```

1.5.2 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.

1.5.3 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
```

1.5.4 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;
```

1.5.5 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!

1.5.6 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
```

1.6 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.

Operator Precedence and Associativity

1.6.1 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.

1.6.2 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

1.7 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

1.7.1 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
```

1.7.2 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();
}
```

1.8 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 */
\
```

Section A

Module 2: Control Structure in C

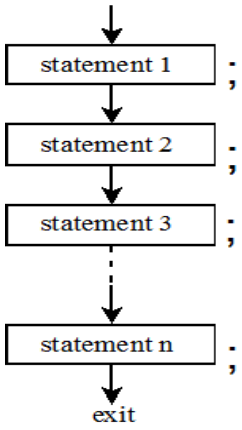
Sequential Flow Statement, Conditional Flow Statement, Decision Control statements: if, if-else, nested-if, else-if ladder. Loop control statements: While, do-while, for loop, Nested of Loops. Case Control Statements: Switch Statement, goto Statement, Break Statement, Continue Statement.

2. 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:

2.1 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.

2.2 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

2.3 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:

2.3.1 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.

2.3.2 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.

2.3.3 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

2.3.4 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.

Syntax:

```
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

2.4 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

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 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

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

2.5 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

2.5.1 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.

2.4.2 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

2.4.3 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

2.4.4 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.

SECTION- B

Module 3: Basics of oops

Basics of oops: Basic concepts (objects, classes, inheritance, polymorphism, encapsulation), Advantages of OOP over procedural programming, Classes and Objects: Declaring classes, creating objects, Access specifiers (public, private, protected), Constructors and destructors, Static members.

3.1 Basics Concepts of OOPs

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:

3.1.1 Object

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 (.)

3.1.2 Classes

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.

3.1.3 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.

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 :

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.

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

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.

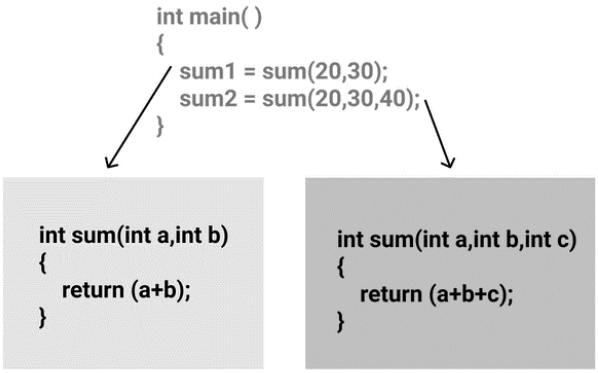
3.1.4 Polymorphism

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.



3.1.5 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

7 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.

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

9 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.

10 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.

3.2 Advantages of OOP over procedural programming

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods. Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time.

3.3 Classes and objects

Classes and objects are the cornerstones of Object-Oriented Programming (OOP) in C++. They enable programmers to create modular, reusable, and maintainable code. In this article, we will dive deep into understanding what classes and objects are, explore their

functionalities, and learn how to implement them in C++ effectively.

Understanding Classes in C++

A class in C++ is a user-defined data type that encapsulates data members (variables) and member functions (methods) within a single unit. Think of a class as a blueprint or template that defines the structure and behavior of objects. Classes are the foundation of OOP and facilitate encapsulation, abstraction, and data hiding.

Analogy of Classes:

Imagine a class of cars. Cars can have different names, models, and brands, yet they all share common properties like having four wheels, a speed limit, and mileage. In this analogy:

- The **Car** represents the class.
- The common properties such as wheels, speed limits, and mileage are the **data members**.
- Actions like accelerating, braking, and honking are the **member functions**.

Defining a Class in C++

A class in C++ is defined using the class keyword followed by the class name. The body of the class contains the data members and member functions, enclosed in curly braces {}.

Syntax:

```
class ClassName {
    access_specifier:
    // Data members and member functions
};
```

- **Access Specifiers**: Control the accessibility of the class members. Common specifiers are public, private, and protected.
- **Data Members**: Variables that hold the state of the class.
- **Member Functions**: Functions that define the behavior of the class.

Example of a Simple Class:

```
class Car {
        public:     // Access specifier
            string brand;  // Data member
            int speed;     // Data member

            // Member function
            void display() {
                cout << "Brand: " << brand << ", Speed: " << speed << " km/h" << endl;
            }
    };
```

Understanding Objects in C++

An object is an instance of a class. When you create an object, you are creating a real-world entity that has the properties and behaviors defined by the class. While a class only defines what an object will be, an object is an actual instance that occupies memory and can be manipulated.

Creating an Object:

To use the data members and functions of a class, you need to create an object. When you create an object, memory is allocated for it, allowing you to access and manipulate the data members and functions defined in the class.

Syntax:

```
ClassName ObjectName;  // Creating an object
```

Example:

```
Car myCar; // Creating an object of the Car class
```

Example: Implementing Classes and Objects in C++

Let's combine everything with a practical example that demonstrates defining a class, creating objects, and interacting with class members.

```
#include <iostream>
        using namespace std;
        // Define a class called 'Car'
        class Car {
```

```
    private: // Private access specifier ensures data security
        string model;
        int year;
        int speed;
    public:
        // Constructor to initialize data members
        Car(string m, int y, int s) : model(m), year(y), speed(s) {}
        // Member function to display car details
        void displayDetails() {
            cout << "Model: " << model << endl;
            cout << "Year: " << year << endl;
            cout << "Speed: " << speed << " km/h" << endl;
        }
        // Member function to accelerate the car
        void accelerate(int increment) {
            speed += increment;
            cout << model << " accelerated to " << speed << " km/h." << endl;
        }

        // Member function to apply brakes
        void brake(int decrement) {
            speed = (speed - decrement >= 0) ? speed - decrement : 0;
            cout << model << " slowed down to " << speed << " km/h." << endl;
        }
};
int main() {
    // Creating an object of the Car class
    Car myCar("Tesla Model S", 2022, 0);

    // Displaying the car details
    myCar.displayDetails();
    // Accelerating the car
    myCar.accelerate(60);
    // Applying brakes
```

```
    myCar.brake(20);
    return 0;
}
```

Output:

```
Model: Tesla Model S
    Year: 2022
    Speed: 0 km/h
    Tesla Model S accelerated to 60 km/h.
    Tesla Model S slowed down to 40 km/h.
```

Key Concepts of Classes and Objects:

1. **Encapsulation**: Classes encapsulate data and methods into a single unit, promoting data security and integrity.
2. **Access Specifiers**:

- **Public**: Members are accessible from outside the class.
- **Private**: Members are only accessible within the class, providing a layer of protection against unintended modifications.
- **Protected**: Members are accessible within the class and derived classes.

1. **Memory Management**: No memory is allocated when a class is defined. Memory is only allocated when an object of the class is instantiated.
2. **Multiple Objects**: You can create multiple objects from a single class, each maintaining its state.

Benefits of Using Classes and Objects in C++:

- **Modularity**: Break down complex problems into smaller, manageable parts.
- **Reusability**: Write code once in the class and reuse it by creating multiple objects.
- **Data Hiding**: Protects data by restricting access to private members.
- **Ease of Maintenance**: Changes in class definitions do not affect other parts of the code.

3.4 Access Specifiers in oops

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:

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().

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

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
```

3.5 Constructors and Destructors

Constructor and Destructor are the special member functions of the class that are created by the C++ compiler or can be defined by the user. The constructor is used to initialize the object of the class while the destructor is called by the compiler when the object is destroyed.

Introduction to Constructor and Destructor 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.

What if we can assign the quantities along with the declaration of the new player automatically? A constructor can help do this in a better and simpler way.

Moreover, when the player deletes his account, we need to deallocate the memory assigned to him. This can be done using a destructor.

What is a Constructor in C++?

A Constructor is a special member function of a class and shares the same name as of class, which means the constructor and class have the same name. Constructor is called by the compiler whenever the object of the class is created, it allocates the memory to the object and initializes class data members by default values or values passed by the user while creating an object. Constructors don’t have any return type because their work is to just create and initialize an object.

Syntax of Constructor:

```
class scaler {
  public:
    // Constructor
    scaler() {
   // Constructor body.
    }
};
```

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. 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. The constructor in C++ cannot be virtual. A virtual table (also called vtable) is made for each class having one or more virtual functions. Virtual functions ensure that the correct function is called for an object regardless of the type of reference used for the function call. Whenever an object is created of such a class, it contains a 'virtual-pointer' that points to the base of the corresponding vtable. Whenever there is a virtual function call, the vtable refers to the function address.

But when a class constructor is executed, there is no virtual table created yet in the memory, meaning no virtual pointer is defined yet. As a result, a constructor cannot be declared virtual.

Types of Constructors in C++

There are four types of constructors in C++.

- Default Constructors
- Parameterized Constructors
- Copy Constructors
- Dynamic Constructors

Default Constructor:

A constructor which does not receive any parameters is called a Default Constructor or a Zero Argument Constructor. Every class object is initialized with the same set of values in the default constructor. Even if a constructor is not defined explicitly, the compiler will provide a default constructor implicitly.

Syntax:

```
//default constructor without any arguments
Employee :: Employee()
```

Example:

```
#include <bits/stdc++.h>
using namespace std;
class Employee {
  public:
    int age;
    // Default constructor.
    Employee() {
      /* Data member is defined with the help of the
      default constructor.*/
      age = 50;
    }
};
int main() {
  // Object of class Employee declared.
  Employee e1;
  // Prints value assigned by default constructor.
  cout << e1.age;
  return 0;
}
```

Output:

50

Example:

Considering the previous example, the default constructor (i.e., Employee() in Employee class definition) defined by the programmer assigns the data member age to a value of 50. But in the given example here, as the data member age is not assigned to any value, the compiler calls the default constructor, and age is initialized to 0 or unpredictable garbage values.

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Employee {
  public:
    int age;
    // Default constructor not defined.
    // Compiler calls default constructor.
};
int main() {
  // Object e1 declared.
  Employee e1;
  cout << e1.age;
  return 0;
}
```

Output:

0

Parameterized Constructor

Using the default constructor, it is impossible to initialize different objects with different initial values. What if we need to pass arguments to constructors which are needed to initialize an object? There is a different type of constructor called Parameterized Constructor to solve this issue. A Parameterized Constructor is a constructor that can accept one or more arguments. This helps programmers assign varied initial values to an object at the creation time.

Example:

```
#include <iostream>
using namespace std;
class Employee {
  public:
    int age;
    // Parameterized constructor
    Employee(int x) {
    /* Age assigned to value passed as an argument
    while object declaration.*/
      age = x;
```

```
    }
};
int main() {
  /* Object c1 declared with argument 40, which
  gets assigned to age.*/
  Employee c1(40);
  Employee c2(30);
  Employee c3(50);
  cout << c1.age << "\n";
  cout << c2.age << "\n";
  cout << c3.age << "\n";
  return 0;
}
Output:          40       30       50
```

Copy Constructor

A Copy constructor in C++ is a type of constructor used to create a copy of an already existing object of a class type. The compiler provides a default Copy Constructor to all the classes. A copy constructor comes into the picture whenever there is a need for an object with the same values for data members as an already existing object. A copy constructor is invoked when an existing object is passed as a parameter.

Syntax:

```
//Copy constructor
Employee :: Employee(Employee &ptr)
```

Example:

```
#include<iostream>
using namespace std;
class Employee {
  private:
    // Data members
    int salary, experience;
  public:
```

```
    // Parameterized constructor
    Employee(int x1, int y1) {
      salary = x1;
      experience = y1;
    }
    // Copy constructor
    Employee(const Employee &new_employee) {
      salary = new_employee.salary;
      experience = new_employee.experience;
    }
    void display() {
      cout << "Salary: " << salary << endl;
      cout << "Years of experience: " << experience << endl;
    }
};
// main function
int main() {
  // Parameterized constructor
  Employee employee1(34000, 2);
  // Copy constructor
  Employee employee2 = employee1;
  cout << "Employee1 using parameterized constructor : \n";
  employee1.display();
  cout << "Employee2 using copy constructor : \n";
  employee2.display();
  return 0;
}
```

Output:

```
Employee1 using the parameterized constructor:
Salary: 34000
Years of experience: 2
Employee2 using copy constructor:
Salary: 34000
Years of experience: 2
```

Explanation:

In this example, object employee1 of class Employee is declared in the first line of the main() function. The data members salary and experience for object employee1 are assigned 34000 and 2, respectively, with the help of a parameterized constructor, which is invoked automatically. When object employee2 is declared in the second line of the main() function, object employee1 is assigned to employee2, which invokes the copy constructor as the argument here is an object itself. As a result, the data member's salary and experience of object employee2 are assigned to values possessed by the salary, and experience data members of object employee1 (i.e., 34000, 2), respectively.

Dynamic Constructor

When the allocation of memory is done dynamically (i.e., Memory is allocated to variables at run-time of the program rather than at compile-time) using a dynamic memory allocator new in a constructor, it is known as a Dynamic constructor. By using this, we can dynamically initialize the objects.

```
#include <iostream>
using namespace std;
class Employee {
  int* due_projects;
  public:
    // Default constructor.
    Employee() {
    // Allocating memory at run time.
      due_projects = new int;
      *due_projects = 0;
    }
    // Parameterized constructor.
    Employee(int x) {
      due_projects = new int;
      *due_projects = x;
    }
```

```
    void display() {
      cout << *due_projects << endl;
    }
};
// Main function
int main() {
  // Default constructor would be called.
  Employee employee1 = Employee();
  cout << "Due projects for employee1:\n";
  employee1.display();
  // Parameterized constructor would be called.
  Employee employee2 = Employee(10);
  cout << "Due projects for employee2:\n";
  employee2.display();
  return 0;
}
```

Output:

Due projects for employee1: 0

Due projects for employee2: 10

Explanation:

Here, integer type pointer variable is declared in class which is assigned memory dynamically when the constructor is called. When we create object employee1 of class Employee in the first line of the main() function, the default constructor(i.e. Employee() in class Employee definition) is called automatically, and memory is assigned dynamically to the pointer type variable(i.e., *due_projects) and initialized with value 0. And similarly, when employee2 is created in the third line of the main() function, the parameterized constructor(i.e., Employee(int x) in the class definition) is called, and memory is assigned dynamically.

Destructor in C++

A Destructor is a member function that is instantaneously called whenever an object is destroyed. The destructor is called automatically by the compiler when the object goes out of scope, i.e., when a function ends, the local objects created within it are also destroyed. The

destructor has the same name as the class name, but the name is preceded by a tilde(~). A destructor has no return type and receives no parameters.

Syntax of Destructor:

```
class scaler {
  public:
   //constructor
    scaler();
   //destructor
    ~scaler();
};
```

Characteristics of a Destructor in C++:

- A destructor deallocates memory occupied by the object when it's deleted.
- A destructor cannot be overloaded. In function overloading, functions are declared with the same name in the same scope, except that each function has a different number of arguments and different definitions. But in a class, there is always a single destructor that does not accept any parameters. Hence, a destructor cannot be overloaded.
- A destructor is always called in the reverse order of the constructor. In C++, variables and objects are allocated on the Stack. The Stack follows the LIFO (Last-In-First-Out) pattern. So, the deallocation of memory and destruction is always carried out in the reverse order of allocation and construction. This can be seen in the code below.
- A destructor can be written anywhere in the class definition. But to bring an amount order to the code, a destructor is always defined at the end of the class definition.

Implementation of Constructors and Destructors in C++

```
#include <iostream>
using namespace std;
class Department {
  public:
    Department() {
      // Constructor is defined.
      cout << "Constructor Invoked for Department class" << endl;
```

```
    }
    ~Department() {
      // Destructor is defined.
      cout << "Destructor Invoked for Department class" << endl;
    }
};
class Employee {
  public:
    Employee() {
      // Constructor is defined.
      cout << "Constructor Invoked for Employee class" << endl;
    }
    ~Employee() {
      // Destructor is defined.
      cout << "Destructor Invoked for Employee class" << endl;
    }
};
int main(void) {
  // Creating an object of Department.
  Department d1;
  // Creating an object of Employee.
  Employee e2;
  return 0;
}
```

Output:

Constructor Invoked for Department class
Constructor Invoked for Employee class
Destructor Invoked for Employee class
Destructor Invoked for Department class

Explanation:

When an object named d1 is created in the first line of main(), i.e. (Department d1), its constructor is automatically invoked during the creation of the object. As a result, the first

line of output, “Constructor Invoked for Department class,” is printed. Similarly, when the e2 object of the Employee class is created in the second line of main(), i.e. (Employee e2), the constructor corresponding to e2 is invoked automatically by the compiler, and “Constructor Invoked for Employee class” is printed.

A destructor is always called in reverse order as that of a constructor. When the scope of the main function ends, the destructor corresponding to object e2 is invoked first. This leads to printing “Destructor Invoked for Employee class”. Lastly, the destructor corresponding to object d1 is called, and “Destructor Invoked for Department class” is printed.

3.6 Static Members in C++

A static data member in C++ is a class member that is shared by all objects of the class. There is only one copy of a static data member, even if multiple objects of the class are created. Static data members are initialized before any object of the class is created.

Syntax

A declaration for a static member is a member declaration whose declaration specifiers contain the keyword static. The keyword static usually appears before other specifiers (which is why the syntax is often informally described as static data-member or static member-function), but may appear anywhere in the specifier sequence. The name of any static data member and static member function must be different from the name of the containing class.

Explanation

Static members of a class are not associated with the objects of the class: they are independent variables with static or thread(since C++11) storage duration or regular functions. The static keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member:

```
class X { static int n; }; // declaration (uses 'static')
int X::n = 1;              // definition (does not use 'static')
```

The declaration inside the class body is not a definition and may declare the member to be of incomplete type (other than void), including the type in which the member is declared:

```
struct Foo;
struct S
```

```
{
   static int a[]; // declaration, incomplete type
   static Foo x;   // declaration, incomplete type
   static S s;     // declaration, incomplete type (inside its own definition)
};
int S::a[10]; // definition, complete type
struct Foo {};
Foo S::x;     // definition, complete type
S S::s;       // definition, complete type
```

However, if the declaration uses constexpr or inline specifier, the member must be declared to have complete type.

To refer to a static member m of class T, two forms may be used: qualified name T::m or member access expression E.m or E->m, where E is an expression that evaluates to T or T* respectively. When in the same class scope, the qualification is unnecessary:

```
struct X
{
   static void f(); // declaration
   static int n;    // declaration
};
X g() { return X(); } // some function returning X
void f()
{
   X::f();  // X::f is a qualified name of static member function
   g().f(); // g().f is member access expression referring to a static member function
}
int X::n = 7; // definition
void X::f() // definition
{
   n = 1; // X::n is accessible as just n in this scope
}
```

Static members obey the class member access rules (private, protected, public).

Static member functions rules

- Static member functions are not associated with any object. When called, they have no this pointer.
- Static member functions cannot be virtual, const, volatile, or ref-qualified.
- The address of a static member function may be stored in a regular pointer to function, but not in a pointer to member function.

Static data members rules

- Static data members are not associated with any object. They exist even if no objects of the class have been defined. There is only one instance of the static data member in the entire program with static storage duration, unless the keyword thread_local is used, in which case there is one such object per thread with thread storage duration.
- Static data members cannot be mutable.
- Static data members of a class in namespace scope have external linkage if the class itself has external linkage (is not a member of unnamed namespace). Local classes (classes defined inside functions) and unnamed classes, including member classes of unnamed classes, cannot have static data members.

A static data member may be declared inline. An inline static data member can be defined in the class definition and may specify an initializer. It does not need an out-of-class definition:

```
struct X
{
  inline static int fully_usable = 1; // No out-of-class definition required, ODR-usable
  inline static const std::string class_name{"X"}; // Likewise
  static const int non_addressable = 1; // C.f. non-inline constants, usable
                                        // for its value, but not ODR-usable
  // static const std::string class_name{"X"}; // Non-integral declaration of this
                                               // form is disallowed entirely
};
```

SECTION- B

Module 4: Inheritance and Polymorphism

Inheritance and Polymorphism: Base and derived classes, Types of inheritance (single, multiple, multilevel, hierarchical), Access control in inheritance, Function overloading, Operator overloading, Virtual functions and runtime polymorphism, Abstract classes and pure virtual functions.

4.1 Inheritance

What is Inheritance: 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.

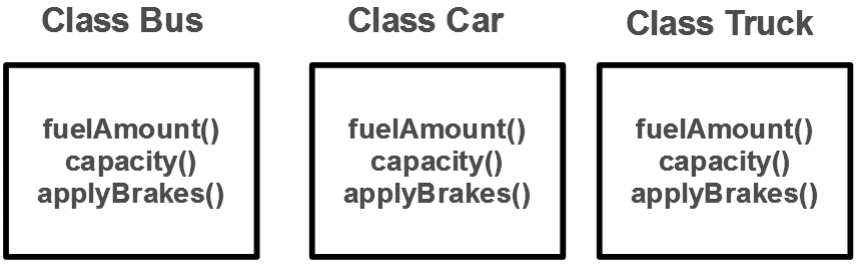
- **Sub Class**: The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class**: The class whose properties are inherited by a subclass is called Base Class or Superclass.

Reusability using Inheritance

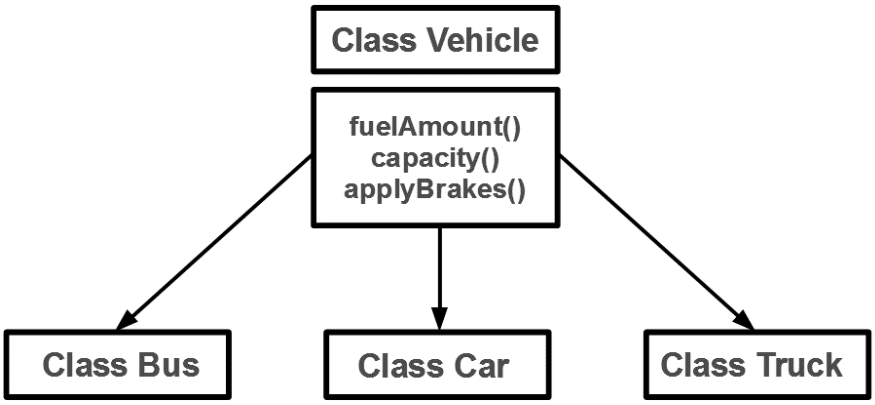
C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The 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
{                          }
```

4.1.1 Types of Inheritance

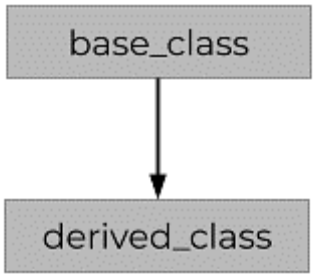
During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. C++ supports five types of inheritance:

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

The detail of each type is given below:

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

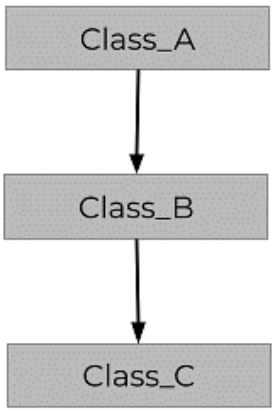
Dog has 1 tail

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.

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

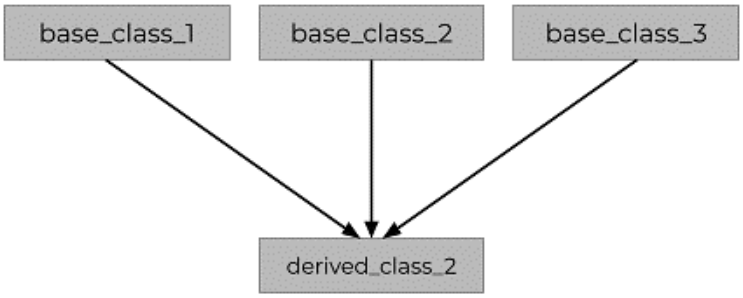
Puppy has 1 tail

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.

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
```

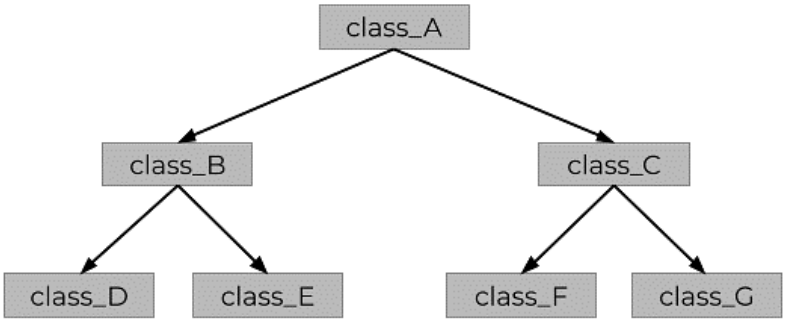
Enter the two highest marks: 40 50
Enter the mark for CoCurricular Activities: 30

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.

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

4.1.2 Access Control in Inheritance

In terms of access control, 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)

4.2 Polymorphism

Polymorphism is derived from two Greek words, “poly” and “morph”, which mean “many” and “forms”, respectively. Hence, polymorphism meaning in oops refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.

Polymorphism in oops Example

As previously explained, polymorphism in oops helps an object take on many different forms. In this section, we will provide different examples of polymorphism to show how it works. The Animal class has a makeSound() method that outputs “Animal making a sound…” while the subclasses Dog, Cat, and Elephant, each provide their own implementation of the same function to produce individual noises.

```
class Animal
{
   void makeSound()
{
      System.out.println("Animal making a sound...");
}
}
class Dog extends Animal
{
   void makeSound() {
      System.out.println("Dog barking...");
   }
}
class Cat extends Animal
{
        void makeSound()
        {
                System.out.println("Cat meowing...");
```

```
        }
}
class Elephant extends Animal
{
        void makeSound()
        {
                System.out.println("Elephant trumpeting...");
        }
}
class TestPolymorphism2
{
   public static void main(String args[])
{
      Animal animal;
      animal = new Dog();
      animal.makeSound();
      animal = new Cat();
      animal.makeSound();
      animal = new Elephant();
      animal.makeSound();
   }
}
```

Output:

Dog barking…

Cat meowing…

Elephant trumpeting…

4.4 Function Overloading in oops

Function Overloading in oops occurs when there are functions having the same name but have different numbers of parameters passed to it, which can be different in data like int, double, float and used to return different values are computed inside the respective overloaded method. Function overloading is used to reduce complexity and increase the efficiency of the program by involving more functions that are segregated and can be used to

distinguish among each other with respect to their individual functionality. Overloaded functions are related to compile-time or static polymorphism. There is also a concept of type conversion, which is basically used in overloaded functions used to calculate the conversion of type in variables.

Overloaded functions have the same name but different types of arguments or parameters assigned to them. They can be used to calculate mathematical or logical operations within the number of assigned variables in the method. The syntax of the overloaded function can be given below, where there are up to N number of variables assigned.

Syntax:

```
public class OverloadedMethod
{
	public int FunctionName(int x, int y) //Two parameters in the function
	{
		return (x + y); //Returns the sum of the two numbers
	}
// This function takes three integer parameters
	public int FunctionName(int x, int y, int z)
	{
		return (x + y + z);
	}
// This function takes two double parameters
	public double FunctionName(double x, double y)
	{
		return (x + y);
	}
//Many more such methods can be done with different number of parameters
// Code used to input the number and
	public static void main(String args[])
	{
		FunctionName s = new FunctionName();
		System.out.println(s.FunctionName(10, 20));
		System.out.println(s. FunctionName(10, 20, 30));
		System.out.println(s. FunctionName(10.5, 20.5));
	}
}
```

Explanation: Function overloading works by calling different functions having the same name, but the different number of arguments passed to it. There are many coding examples that can be shown in order to identify the benefits and disadvantages of function overloading properly.

4.5 Operator overloading in oops

Operator overloading aims to redefine an operator that has been defined and has certain functions to complete more detailed and specific operations and other functions. From an object-oriented perspective, it means an operator can be defined as a method of a class, so the function of the operator can be used to represent a certain behaviour of the object.
There are at least two benefits to being able to perform operator overloading for numeric operations of non-primitive types.

1. The code is simpler to write and less error-prone.
2. The code is easier to read without many parentheses.

How to Implement Operator Overloading in oops

The implementation of operator overloading in oops still uses Manifold. Manifold allows you to overload Java operators in various scenarios, such as arithmetic operators (including +,-, *, /, and %), comparison operators (>, >=, <, <=, ==, and !=), and index operators ([]). Please see Java's Missing Feature: Extension Methods for more information about the integration of Manifold.

Arithmetic Operator

Manifold is a function that maps each overload of an arithmetic operator to a specific name. For example, if you define a plus(B) method in class A, that class can be called using a + b instead of a.plus(b). The following chart describes the mappings:

Operator	Method Call
c = a + b	c = a.plus(b)
c = a – b	c = a.minus(b)
c = a * b	c = a.times(b)
c = a / b	c = a.div(b)
c = a % b	c = a.rem(b)

Those familiar with Kotlin should know that this is an imitation of Kotlin's operator overloading.
Let's define a numeric Num to facilitate illustration.

```
public class Num
{
```

```
        private final int v;
        public Num(int v)
        {
                this.v = v;
        }

        public Num plus(Num that)
        {
                return new Num(this.v + that.v);
        }

        public Num minus(Num that)
        {
                return new Num(this.v - that.v);
        }

        public Num times(Num that)
        {
                return new Num(this.v * that.v);
        }
}
```

For the following code:

```
Num a = new Num(1);
Num b = new Num(2);
Num c = a + b - a;
```

4.6 Virtual functions and runtime polymorphism

A member function that has the keyword virtual used in its declaration in the base class and is redefined (Overridden) in the derived class is referred to as a virtual function. The late binding instruction instructs the compiler to execute the called function during runtime by matching the object with the appropriately called function. Runtime Polymorphism refers to this method.

1. No matter what kind of reference (or pointer) is used to invoke a function, virtual functions make sure the right function is called for an object.
2. Their primary purpose is to implement runtime polymorphism.
3. In base classes, functions are declared using the virtual keyword.
4. Runtime resolution of function calls is carried out.

Polymorphism is a term used to describe the capacity to assume several shapes. If there is a hierarchy of classes connected to one another by inheritance, it happens. Polymorphism, which is defined as "showing diverse traits in different contexts," can be summarised as

"showing different characteristics in a variety of situations" and "polymorphism."

What is the use of virtual functions?

To achieve Runtime Polymorphism, virtual functions are primarily used. Only a base class type pointer (or reference) can enable runtime polymorphism. A base class pointer can also point to both objects from the base class and those from derived classes. Also, without even knowing the type of derived class object, we can use virtual functions to compile a list of base class pointers and call any of the derived classes' methods.

```
#include<iostream>
using namespace std;
class B
{
public:
	virtual void s()
	{
		cout<<" In Base \n";
	}
};
class D: public B
{
	public:
		void s()
		{
			cout<<"In Derived \n";
		}
};

int main(void)
{
	D d; // An object of class D
	B *b= &d; // A pointer of type B* pointing to d
	b->s(); // prints "D::s() called"
	return 0;
}
```

Output: In Derived

What are the rules for virtual functions?

I. Virtual functions are not permitted to be static or friendly to other classes.

II. Pointers or references of base class type are required to access virtual functions.

III. Both the base class and any derived classes should use the same function prototype.

IV. There cannot be a virtual constructor in a class. However, it might have a virtual destroyer.

V. The base class always defines them, and the derived class redefines them.

What is runtime polymorphism?

Runtime polymorphism is the process of binding an object at runtime with a capability. Overriding methods is one way to implement runtime polymorphism. At runtime, not at compilation time, the Java virtual machine decides which method to invoke. Additionally known as dynamic binding or late binding. The parent class's method is overridden in the child class, according to this concept. The term "method overriding" refers to the situation where a child class implements a method specifically that was supplied by one of its parent classes. You can see runtime polymorphism in the example that follows.

Example

```
class Test
{
	public void method()
	{
		System.out.println("Method 1");
	}
}
public class DEMO extends Test
{
	public void method()
	{
		System.out.println("Method 2");
	}
	public static void main(String args[])
	{
		Test test = new DEMO();
		test.method();
	}
}
```

Output: Method 2

4.7 Abstract Class in oops

In oops, abstract class is declared with the abstract keyword. It may have both abstract and non-abstract methods (methods with bodies). An abstract is a Java modifier applicable for classes and methods in oops but not for Variables. In this article, we will learn the use of

abstract classes in oops. Furthermore, Java abstract class is a class that cannot be initiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the “abstract” keyword in its class definition.

Illustration of Abstract class

```
abstract class Shape
{
   int color;
   // An abstract function
   abstract void draw();
}
```

In oops, the following some important observations about abstract classes are as follows:

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. There can be a final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”.
5. We can define static methods in an abstract class.
6. We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract.
7. If a class contains at least one abstract method then compulsory should declare a class as abstract.
8. If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

Example of Java Abstract Class

```
// Abstract class
abstract class Sunstar
{
   abstract void printInfo();
}

// Abstraction performed using extends
```

```
class Employee extends Sunstar
{
   void printInfo()
   {
      String name = "avinash";
      int age = 21;
      float salary = 222.2F;
       System.out.println(name);
      System.out.println(age);
      System.out.println(salary);
   }
}

// Base class
class Base {
   public static void main(String args[])
   {
      Sunstar s = new Employee();
      s.printInfo();
   }
}
```

Output:

```
avinash
21
222.2
```

4.8 Pure Virtual Function

Pure virtual function is a virtual function for which we don't have implementations. An abstract method in oops can be considered as a pure virtual function. Let's take an example to understand this better.

Example of Pure Virtual Function:

```
abstract class Dog
{
        final void bark()
        {
                System.out.println("woof");
        }
```

```
        abstract void jump(); //this is a pure virtual function
}
class MyDog extends Dog
{
    void jump()
    {
        System.out.println("Jumps in the air");
    }
}
public class Runner
{
    public static void main(String args[])
    {
        Dog ob1 = new MyDog();
        ob1.jump();
    }
}
```

Output: Jumps in the air

This is how virtual function can be used with abstract class.

Run-Time Polymorphism

Run-time polymorphism is when a call to an overridden method is resolved at run-time instead of compile-time. The overridden method is called through the reference variable of the base class.

Output: Java Certification Course

```
class Edureka
{
    public void show()
    {
        System.out.println("welcome to edureka");
    }
```

```
}
class Course extends Edureka
{
        public void show()
        {
                System.out.println("Java Certification Program");
        }
public static void main(String args[])
{
        Edureka ob1 = new Course();
        ob1.show();
}
}
```

Points To Remember

- For a virtual function in oops, you do not need an explicit declaration. It is any function that we have in a base class and redefined in the derived class with the same name.
- The base class pointer can be used to refer to the object of the derived class.
- During the execution of the program, the base class pointer is used to call the derived class functions.

End of the Document