PROGRAMMING IN C++

CORE COURSE II

PROGRAMMING IN C++

Objective:

To impart basic knowledge of Programming Skills in C++ language.

Unit I

Basic Concepts of Object- Oriented Programming - Benefits of OOP - Object Oriented Languages - Applications of OOP – Structure of C++ Program - Tokens, Expressions and Control Structures – Functions in C++

Unit II

Classes and Objects – Constructors and Destructors –Operator Overloading and Type Conversions

Unit III

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

Unit IV

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

1. Balagursamy E, Object Oriented Programming with C++, Tata McGraw Hill Publications, Sixth Edition, 2013

Reference Books

1. Ashok Kamthane, Programming in C++, Pearson Education,2013.

*****

VALLUVAR COLLEGE OF SCIENCE AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE

2 Mark Questions

Unit – I:

Basic Concepts of Object- Oriented Programming - Benefits of OOP – Object Oriented Languages - Applications of OOP – Structure of C++ Program - Tokens, Expressions and Control Structures – Functions in C++

1. What are the characteristics of procedure oriented programming?

Some characteristics exhibited by procedure-oriented programming are:

• Emphasis is on doing things (algorithms).
• Large programs are divided into smaller programs known as functions.
• Most of the functions share global data.
• Data move openly around the system from function to function.
• Functions transform data from one form to another.
• Employs top-down approach in program design.

2. Write the principal advantages of Object Oriented Programming?

The principal advantages are:

• Through inheritance, we can eliminate redundant code and extend the use of existing classes.
• We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
• The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
• It is possible to have multiple instances of an object to co-exist without any interference.
• It is possible to map objects in the problem domain to those in the program.
• It is easy to partition the work in a project based on objects.
• The data-centered design approach enables us to capture more details of a model inimplementable form.
• Object-oriented systems can be easily upgraded from small to large systems.
• Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
• Software complexity can be easily managed.

3. Define Object.

Objects are the basic run·time entities in an object·oriented system. They may represent a person, a place, a bank aooount, a table of data or any item that the program has to handle. They may also represent user·defined data such as vectors, time and lists.

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.

4. What is Object Oriented Programming?

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

5. What is Class?

A class in C++ is a user-defined type or data structure declared with keyword ***class*** that has data and functions (also called member variables and member functions) as its members whose access is governed by the three access specifiers private, protected or public. By default access to members of a C++ class is private. The private members are not accessible outside the class; they can be accessed only through methods of the class. The public members form an interface to the class and are accessible outside the class.

6. Write few words about Object Oriented Languages.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages, and
2. Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based progrramming are:

• Data encapsulation
• Data hiding and access mechanisms
• Automatic initialization and clear-up of objects
• Operator overloading

Languages that support programming v.itb objects a.re said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object·oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

Object-based features + Inheritance + dynamic binding

Languages that support these features include C++, Smalltalk, Object Pascal and Java.

7. List the applications of object oriented programming?

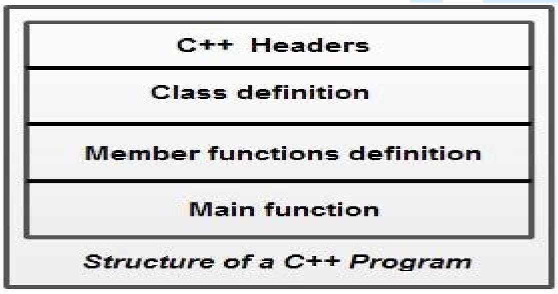
   The promising areas for application of OOP include:
   - Real-time systems
   - Simulation and modeling
   - Object-oriented databases
   - Hypertext, hypermedia and expertext
   - AI and expert systems
   - Neural networks and parallel programming
   - Decision support and office automation syste1ns
   - CIM/CAM/CAD systems

8. What is C++?

   C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirerofSimula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. rhereforo, C++ is an extension ofC with a major addition of the class oonstruct feature of Simula67. Since the class was a major addition to tho original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea ofC++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

9. Give the structure of C++ program.

   Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely, headers, class definition, member functions definitions and main function.



Structure of a C++ Program

Note that C++ provides the flexibility of writing a program with or without a class and its member functions definitions.

10.Write note on iostream File.

We have used the following #include directive in the program:

#include <iostream>

This directive causes the preprocesSor to add the contents of the iostream file to the program. It contains declarations for the identifier **cout** and the operator **<<**. Some old versions of C++ use a header file called iootream.h. This is one of the changes introduced by ANSI C++. The header file iostream should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use iostream.hpp; yet others iostream.hxx. We must include appropriate header files depending on the contents of the program and implementation.

11.What is Namespace?

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directive, like

using namespace std;

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current global scope. **using** and **namespace** are the new keywords of C++.

12.What are tokens in C++?

The smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language.

13.What is a variable in C++? How is it declared?

A variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In C++, all the variables must be declared before use.

How to declare variables?

A typical variable declaration is of the form:

```
// Declaring a single variable
type variable_name;

// Declaring multiple variables:
type variable1_name, variable2_name, variable3_name;
```

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore '_' character. However, the name must not start with a number.

14.Enumerate the rules of naming variables in C++.

Following are the rules for naming variables:

1. Variable names in C++ can range from 1 to 255 characters.
2. All variable names must begin with a letter of the alphabet or an underscore(_).
3. After the first initial letter, variable names can also contain letters and numbers. Variable names are case sensitive.
4. No spaces or special characters are allowed.
5. You cannot use a C++ keyword (a reserved word) as a variable name.

Here are some examples of acceptable variable names:

mohd	Piyush	abc	move_name	a_123
myname50	_temp	j	a23b9	retVal

You can declare variables using the syntax:

```
datatype variable_name;
```

For example,

```
int my_var;
float my_float;
```

15.Write about C++ data types?

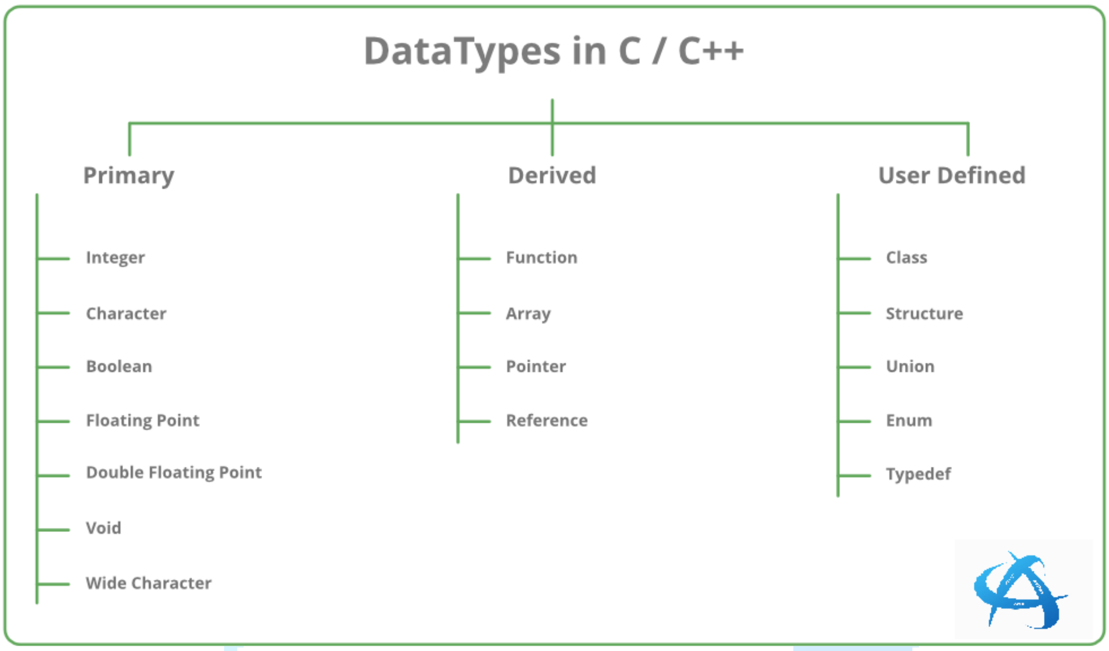
Data types in C++ is mainly divided into three types:

1. **Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:
   - Integer
   - Character

- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

2. **Derived Data Types:** The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
   - Function
   - Array
   - Pointer
   - Reference

3. **Abstract or User-Defined Data Types:** These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
   - Class
   - Structure
   - Union
   - Enumeration
   - Typedef defined DataType



16.What are the new operators in C++?

All C operators are valid in C++ also. In addition, C++ introduces some new operators. The new operators are:

<<	Insertion operator
>>	Extraction operator
::	Scope resolut ion operator
::*	Pointer-to-member declaratory

=>*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
new	Memory allocation operator
setw	Field width operator

17.What is scope resolution operator?

In C++, scope resolution operator is ::. It is used for following purposes.

1) To access a global variable when there is a local variable with same name:
2) To define a function outside a class.
3) To access a class's static variables.
4) In case of multiple Inheritance:

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

5) For namespace

If a class having the same name exists inside two namespace we can use the namespace name with the scope resolution operator to refer that class without any conflicts

6) Refer to a class inside another class:

If a class exists inside another class we can use the nesting class to refer the nested class using the scope resolution operator

18.What are manipulators?

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as uaing the ne\vline character "\n". For example, the statement

```
.....
.....
cout << 1111 = " << m << end l
<< "n = " << n << end l
<< "p = " << p << end 1:
.....
.....
```

would cause three lines of output, one for each variable.

If the numbers are have to right-justified in the form of output, is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(S) << stMn << endl;
```

The manipulator setw(5) specifies a field width 5 for printing the value of the variable sum.

19.Write about Type Cast Operator.

C++ _permits explicit type conversion or variables or expressions using the type cast operator.
Traditional C casts arc augmented in C++ by a function..call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression // C notation
type-name (expression) // C++ notation
```

Examples:
```
average • sum/(float)i; // C notation
average •sum/float(!); // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier.
For example,
```
p = int * (q);
```

20.What is Function Overloading?

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
Function overloading can be considered as an example of polymorphism feature in C++.
Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

21.Define function prototyping?

A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of arguments.

Function prototyping is one very useful feature of C++ function. A function prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

Syntax:
type function-name (argument- list);

The argument·list contains the types and names of argument that must be passed to the function.

Example:
float volume(int x, float y, float z) ;

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

float volume(int x, float y, z);

is illegal.

22.What is inline function?

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code
}
```

23.What is data abstraction?

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Abstraction using Classes: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Abstraction using access specifiers: Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

Members declared as public in a class, can be accessed from anywhere in the program.
Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers.

24.Explain GoTo Statement.

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.
Syntax:

```
Syntax1      |  Syntax2
----------------------------
goto label;  |   label:
.            |   .
.            |   .
.            |   .
label:       |   goto label;
```

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier which indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.

<u>Unit – II:</u>

Classes and Objects – Constructors and Destructors –Operator Overloading and Type Conversions

1. Define Classes and Objects.

A class is a way to bind the data and its associated functions together. It allows the data (and function&) to be hidden, if necessary, from external use. when defining a class, we are creating a new abstract data type that can be treated like any other built-in data type.
Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implement.
The general form of a class declaration ia:

```
class class name
(
};
private:
            variable declarations;
            function declarations;
public:
            variable declarations;
            function declaration;
```

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.

2. Distinguish between Object and Classes.

Class	Object
A class is a blueprint from which you	An object is the instance of the class,

can create the instance, i.e., objects.	which helps programmers to use variables and methods from inside the class.
A class is used to bind data as well as methods together as a single unit.	object acts as a variable of the class.
Classes have logical existence.	Objects have a physical existence.
A class doesn't take any memory spaces when a programmer creates one.	An object takes memory when a programmer creates one.
The class has to be declared only once.	Objects can be declared several times depending on the requirement.

3. What is Static member function?

Just like static member variables we have static member functions that are used for a specific purpose. To create a static member function we need to use the static keyword while declaring the function. Since static member variables are class properties and not object properties, to access them we need to use the class name instead of the object name.

Properties of static member functions:

A static function can only access other static variables or functions present in the same class

Static member functions are called using the class name.
Syntax- class_name::function_name( )

4. What is Friend function in C++?

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows –

```
class ABC
{
    .....
    .....
public:
    .....
    .....
friend void xyz(void); // declaration
} ;
```

5. List the special characteristics of friend function.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has boon declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

6. What is meant by Constructors and list various types of Constructors?

A constructor is a 'special' member function whose task is to initialize the object of its class. It is special because its name is the same as the Class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
A constructor is dec1nred and defined as follows:

```
// class with a constructor
class integer
{
        int m, n;
    public:
        integer(void) ;     //constructor declared
        .....
        .....
integer :: integer(void)    //constructor defined
{
    m = 0; n = 0;
}
```

Types of Constructors

I. **Default Constructors:** Default constructor is the constructor which doesn’t take any argument. It has no parameters.
II. **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.
III. **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class.

7. What is a Parameterized Constructor?

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

The constructor integer() may be modified to take arguments aa shown below:

```
class integer
{

        int m, n;
   public:
        integer (int x, int y);        //parameterized constructor
        .....
        .....
 };
 integer :: integer(int x, int y)
 {
        m=x;  n = y;
 }
```

When a constructor has boon parameterized, the object declaration statement such as

```
integer int1;
```

may not work, We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

• By calling the constructor explicitly.
• By calling the constructor implicitly.

8. What is copy constructor?

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
    ClassName (const ClassName &old_obj);
```

Following is a simple example of copy constructor.

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()          {  return x; }
    int getY()          {  return y; }
};

int main()
{
    Point p1(10, 15);    // Normal constructor is called here
    Point p2 = p1;       // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

9. What is meant by Destructors? Give example.

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function u1hosc name is the same as the class name but is preceded by a tilde(~). For example, the destructor for the class integer can be defined as shown below:

~Integer() { }

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};
```

```
String::String(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

String::~String()
{
   delete []s;
}
```

10.Describe the importance of destructor.

- Automatic invocation and no explicit call from user code
- Overloading or inheritance not allowed
- Access modifiers or parameters not to be specified
- Order of call to destructor in a derived class is from the most derived to the least derived
- Called not only during the object destruction, but also when the object instance is no longer eligible for access
- Used in classes but not structs
- Used only to release expensive unmanaged resources (like windows, network connection, etc.) that the object holds, rather than for releasing managed references
- It is called automatically whenever object is destroyed or removed from the memory.
- It is used to perform any operation at the time of destruction of any object.
- It is called as per LIFO(Stack) method.(IMP)

11.How is a member function of a class defined outside?

A public member function can also be defined outside of the class with a special type of operator known as Scope Resolution Operator (SRO); SRO represents by :: (double colon)

Let's consider the syntax

```
return_type class_name::function_name(parameters)
{
       function_body;
}
```

Syntax for declaring function outside of class

```
class class_name
 {
   ........
   ........
   public:
     return_type function_name (args); //function declaration
 };
//function definition outside class
return_type class_name :: function_name (args)
 {
   ..........; // function definition
 }
```

12.Write a program to overload unary minus operator.

```
/*C++ program for unary minus (-) operator overloading.*/
#include<iostream>
using namespace std;

class NUM
{
    private:
        int n;

    public:
        //function to get number
        void getNum(int x)
        {
            n=x;
        }
        //function to display number
        void dispNum(void)
        {
            cout << "value of n is: " << n;
        }
        //unary - operator overloading
        void operator - (void)
        {
            n=-n;
        }
};

int main()
{
    NUM num;
    num.getNum(10);
    -num;
    num.dispNum();
    cout << endl;
    return 0;
}
```

Output

```
    value of n is: -10
```

13.Give any two rules of Operator Overloading.

There are certain restrictions and limitations in overloading Operators. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use **friend** functions to overload certain operators. However member functions can be used to overload them.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argunl.cnt (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

14.What do you mean by type conversion?

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1) Implicit Type Conversion Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

```
// An example of implicit conversion

#include <iostream>
using namespace std;

int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;

    return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2) **Explicit Type Conversion:** This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

  Syntax:

  ```
  (type) expression
  ```

  where type indicates the data type to which the final result is converted.

Example:

```
// C++ program to demonstrate
// explicit type casting
#include <iostream>
```

```
using namespace std;
int main()
{
    double x = 1.2;
    // Explicit conversion from double to int
    int sum = (int)x + 1;
    cout << "Sum = " << sum;
    return 0;
}

Output:
Sum = 2
```

- **Conversion using Cast operator:** A Cast operator is an unary operator which forces one data type to be converted into another data type.

C++ supports four types of casting:

1) Static Cast
2) Dynamic Cast
3) Const Cast
4) Reinterpret Cast

Example:

```
#include <iostream>
using namespace std;
int main()
{
     float f = 3.5;
    // using cast operator
     int b = static_cast<int>(f);
     cout << b;
}
Output:
3
```

Unit – III:

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

1. What is derived class?

A derived class is a class created or derived from another existing class. The existing class from which the derived class is created through the process of inheritance is known as a base class or superclass.

Derived classes are used for augmenting the functionality of base class by adding or modifying the properties and methods to suit the requirements of the specialization necessary for derived class. This allows for defining virtual methods that form the means to implement polymorphism, which allows a group of objects to work in uniform manner. Thus, the inherent advantages of inheritance and polymorphism like code reuse, faster development, easy maintenance, etc., are realized.

A derived class is also known as subclass or child class.

The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
        ..... //
        ..... //     members of derived class
        ..... //
} ;
```

2. What is Inheritance mean in C++?

The capability of a class to derive properties and characteristics from another class is called Inheritance.
Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

3. What are the different types/forms of Inheritance?

1) Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

Syntax:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

2) Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

Syntax:

```
class subclass_name : access_mode base_class1, access_mode
base_class2, ....
{
  //body of subclass
};
```

3) Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

4) **Hierarchical Inheritance:** In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

5) **Hybrid (Virtual) Inheritance:** Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

4. What is a virtual base class in C++?

The virtual base class is used when a derived class has multiple copies of the base class.

```
#include <iostream>
using namespace std;
class B {
  public: int b;
};

class D1 : public B {
  public: int d1;
};

class D2 : public B {
  public: int d2;
};

class D3 : public D1, public D2 {
  public: int d3;
};

int main() {
  D3 obj;

  obj.b = 40; //Statement 1, error will occur
  obj.b = 30; //statement 2, error will occur
  obj.d1 = 60;
  obj.d2 = 70;
  obj.d3 = 80;

  cout<< "\n B : "<< obj.b
  cout<< "\n D1 : "<< obj.d1;
  cout<< "\n D2: "<< obj.d2;
  cout<< "\n D3: "<< obj.d3;
}
```

In the above example, both D1 & D2 inherit B, they both have a single copy of B. However, D3 inherit both D1 & D2, therefore D3 have two copies of B, one from D1 and another from D2.

5. What is meant by Pointers? Give example.

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C/C++ has the format:

Syntax:

```
datatype *var_name;
int *ptr;   //ptr can point to an address which holds int data
```

6. Explain This Pointer?

The this pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class.

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

Example:

```
#include <iostream>
using namespace std;
class Demo {
private:
  int num;
  char ch;
public:
  void setMyValues(int num, char ch){
    this->num =num;
    this->ch=ch;
  }
  void displayMyValues(){
    cout<<num<<endl;
    cout<<ch;
  }
};
int main(){
  Demo obj;
  obj.setMyValues(100, 'A');
  obj.displayMyValues();
  return 0;
}
```

Output:

```
100
A
```

7. List any two uses of pointers.

    - To pass arguments by reference
    - For accessing array elements
    - To return multiple values
    - Dynamic memory allocation
    - To implement data structures
    - To do system level programming where memory addresses are useful

8. Define virtual function.

    A virtual function a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

    - Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
    - They are mainly used to achieve Runtime polymorphism
    - Functions are declared with a virtual keyword in base class.
    - The resolving of function call is done at Run-time.

    Rules for Virtual Functions

    1) Virtual functions cannot be static and also cannot be a friend function of another class.
    2) Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
    3) The prototype of virtual functions should be same in base as well as derived class.
    4) They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
    5) A class may have virtual destructor but it cannot have a virtual constructor.

9. What is a pure function?

    A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. The only result of calling a pure function is the return value. Examples of pure functions are strlen(), pow(), sqrt() etc. Examples of impure functions are printf(), rand(), time(), etc.

10.What is Pure Virtual Functions?

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

11.What is an abstract class in C++?

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

12.Define 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.

Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

13.What are the different types/forms of Polymorphism?

1) **Compile time polymorphism:** This type of polymorphism is achieved by function overloading or operator overloading.

   - ***Function Overloading:*** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

   - ***Operator Overloading:*** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them

2) **Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.

- ***Function overriding*** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Unit – IV:

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

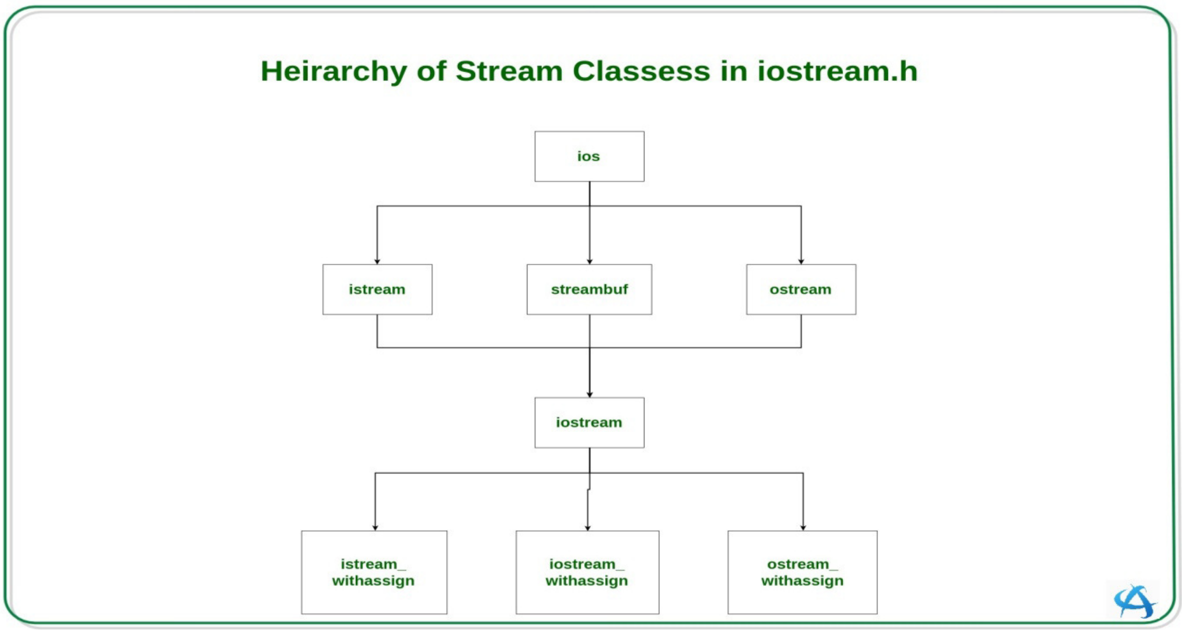
1. What is stream?

The I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as *stream*.

A stream is a sequence of bytes. It acts either as a *source* from which the input data can be obtained or as a *destination* to which the output data can be sent. The source stream that provides data to the program is called the *input stream* and the destination stream that receives output from the program is called the *output stream.*

2. Explain the classes for file stream operation.

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file iostream.h. Figure given below shows the hierarchy of these classes.



1) **ios class** is topmost class in the stream classes hierarchy. It is the base class for **istream**, **ostream**, and **streambuf** class.
2) **istream** and **ostream** serves the base classes for **iostream** class. The class **istream** is used for input and **ostream** for the output.
3) Class **ios** is indirectly inherited to **iostream** class using **istream** and **ostream**. To avoid the duplicity of data and member functions of **ios** class, it is declared as virtual base class when inheriting in **istream** and **ostream** as

```
class istream: virtual public ios
{
};
class ostream: virtual public ios
{
};
```

4) The **_withassign classes** are provided with extra functionality for the assignment operations that's why **_withassign** classes.

3. What is an iostream class?

This class is responsible for handling both input and output stream as both istream class and istream class is inherited into it. It provides function of both istream class and istream class for handling chars, strings and objects such as get, getline, read, ignore, putback, put, write etc..

Example:

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

int main()
{

   // this function display
   // ncount character from array
   cout.write("SachinAathav", 6);
}
```

Output:

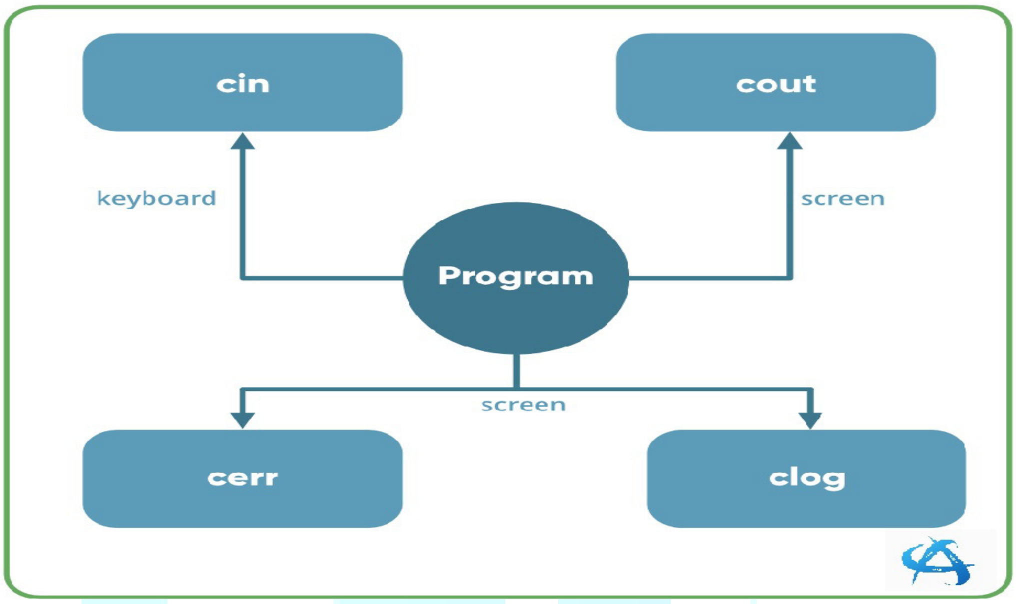
```
Sachin
```

4. How will be the I/O structure in C++?

C++ comes with libraries which provides us with many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.

Input Stream: If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.

Output Stream: If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



I) **Standard output stream (cout):** Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

II) **standard input stream (cin):** Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class istream and is used to read input from the standard input device which is usually a keyboard.
The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object cin which is entered using the keboard.

III) **Un-buffered standard error stream (cerr):** The C++ cerr is the standard error stream which is used to output the errors. This is also an instance of the ostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display later.

IV) **Buffered standard error stream (clog):** This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. The error message will be displayed on the screen too.

5. Give any two features of I/O system supported by C++.

   - C++ *IO is type safe*. IO operations are defined for each of the type. If IO operations are not defined for a particular type, compiler will generate an error.
   - C++ IO *operations are based on streams of bytes and are device independent*. The same set of operations can be applied to different types of IO devices.

6. What is a file?

   A file represents a sequence of bytes, regardless of it being a text file or a binary file. C++ programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

7. What is meant by file input and output streams?

   C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bi-directional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.

   To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

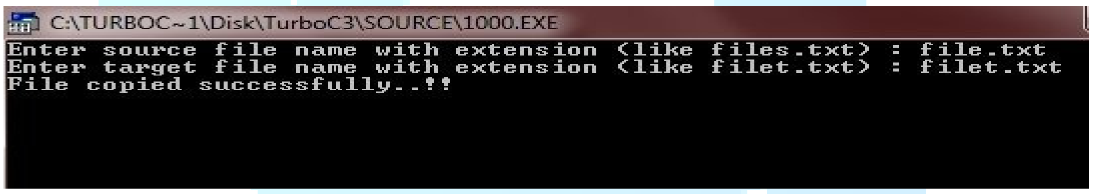
   File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).

8. Write a program to copy a content of file to another file.

```
/* C++ Program - Copy Files */

#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include<stdio.h>
#include<stdlib.h>
void main()
{
        clrscr();
        ifstream fs;
        ofstream ft;
        char ch, fname1[20], fname2[20];
        cout<<"Enter source file name with extension (like files.txt) : ";
        gets(fname1);
        fs.open(fname1);
        if(!fs)
```

```
	{
		cout<<"Error in opening source file..!!";
		getch();
		exit(1);
	}
	cout<<"Enter target file name with extension (like filet.txt) : ";
	gets(fname2);
	ft.open(fname2);
	if(!ft)
	{
		cout<<"Error in opening target file..!!";
		fs.close();
		getch();
		exit(2);
	}
	while(fs.eof()==0)
	{
		fs>>ch;
		ft<<ch;
	}
	cout<<"File copied successfully..!!";
	fs.close();
	ft.close();
	getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:



```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1000.EXE
Enter source file name with extension (like files.txt) : file.txt
Enter target file name with extension (like filet.txt) : filet.txt
File copied successfully..!!
```

9. List out any two manipulators and their meanings.

Stream Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects, for example:

std::cout << std::setw(10);

They are still regular functions and can also be called as any other function using a stream object as an argument, for example:

boolalpha (cout);

Manipulators are used to changing formatting parameters on streams and to insert or extract certain special characters.

Following are some of the most widely used C++ manipulators:

endl

This manipulator has the same functionality as '\n'(newline character). But this also flushes the output stream.

setw

This manipulator changes the width of the next input/output field. When used in an expression out << setw(n) or in >> setw(n), sets the width parameter of the stream out or in to exactly n.

showpoint/noshowpoint

This manipulator controls whether decimal point is always included in the floating-point representation.

setprecision

This manipulator changes floating-point precision. When used in an expression out << setprecision(n) or in >> setprecision(n), sets the precision parameter of the stream out or into exactly n.

10.What is an exception?

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **try** − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

11.Give the rules of Exception Handling.

- Use purpose-designed user-defined types as exceptions (not built-in types)
- Catch exceptions from a hierarchy by reference
- Destructors, deallocation, and swap must never fail

- Don't try to catch every exception in every function
- Minimize the use of explicit try/catch

12. What are the advantages of using exception handling mechanism in C++?

Following are main advantages of exception handling over traditional error handling.

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

13. What is Generic Programming?

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.

<u>Unit – V:</u>

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

1. What is STL? What are its components?

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components –

1) Containers

Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.

2) Algorithms

Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

3) Iterators

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

2. Explain iterators.

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

Operations of iterators :-

1. **begin()** :- This function is used to return the beginning position of the container.

2. **end()** :- This function is used to return the after end position of the container.

3. **advance()** :- This function is used to increment the iterator position till the specified number mentioned in its arguments.

4. **next()** :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

5. **prev()** :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

6. **inserter()** :- This function is used to insert the elements at any position in the container. It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.

3. List the types of containers.

C++ contains three types of containers:

- ✓ Sequential Containers
- ✓ Associative Containers
- ✓ Unordered Containers

1) Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

- **array:** Static contiguous array (class template)
- **vector:** Dynamic contiguous array (class template)
- **deque:** Double-ended queue (class template)
- **forward_list:** Singly-linked list (class template)
- **list :** Doubly-linked list (class template)

2) Associative containers

Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

- **Set:** Collection of unique keys, sorted by keys (class template)
- **Map:** Collection of key-value pairs, sorted by keys, keys are unique (class template).
- **multiset:** Collection of keys, sorted by keys (class template)
- **multimap:** Collection of key-value pairs, sorted by keys (class template)

3) Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

- **unordered_set:** Collection of unique keys, hashed by keys. (class template)
- **unordered_map:** Collection of key-value pairs, hashed by keys, keys are unique. (class template)
- **unordered_multiset:** Collection of keys, hashed by keys (class template)
- **unordered_multimap:** Collection of key-value pairs, hashed by keys (class template)

4. List the types of Iterators.

   ✓ Input Iterators
   ✓ Output Iterators
   ✓ Forward Iterator
   ✓ Bidirectional Iterators
   ✓ Random-Access Iterators

5. Draw the relationship between the three STL components.

1) Containers

Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.

2) Algorithms

Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

3) Iterators

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

6. Write three most popular container classes and its usage.

Container class is one of the classes that were put into class libraries. To handle objects that contain other objects, container classes are used. A GUI class library contains a group of container classes.

The following are the standardized container classes :
1. **std::map :**
Used for handle sparse array or a sparse matrix.
2. **std::vector :**
Like an array, this standard container class offers additional features such as bunds checking through the at () member function, inserting or removing elements, automatic memory management and throwing exceptions.
3. **std::string :**
A better supplement for arrays of chars.

7. What is String?

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type string and assign it a value:
string greeting = "Hello";

8. What is string class?

C++ has in its definition a way to represent sequence of characters as an object of class. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing access to single byte character.

9. Give any two examples of manipulation of strings.

- **strcpy**(str1, str2): Copies string str2 into string str1.
- **strcat**(str1, str2): Concatenates string str2 onto the end of string str1.
- **strlen**(str1): Returns the length of string str1.
- **strcmp**(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- **strchr**(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- **strstr**(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

10. Write commonly used string constructors?

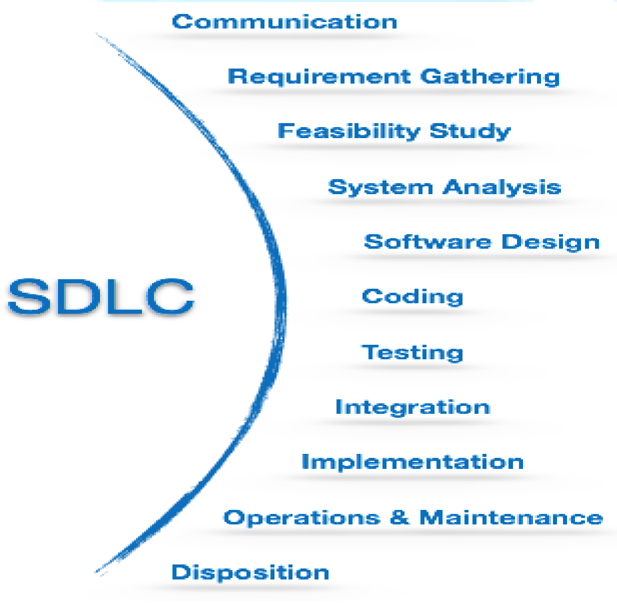
- **String():** This constructor is used for creating an empty string
- **String(const char *str):** This constructor is used for creating string objects from a null-terminated string
- **String(const string *str):** This constructor is used for creating a string object from another string object

11. What is data flow diagram?

Also known as DFD, Data flow diagrams are used to graphically represent the flow of data in a business information system. DFD describes the processes that are involved in a system to transfer data from the input to the file storage and reports generation.

Data flow diagrams can be divided into logical and physical. The logical data flow diagram describes flow of data through a system to perform certain functionality of a business. The physical data flow diagram describes the implementation of the logical data flow.

12. Draw the classic software development life cycle.



13.What is prototyping?

Prototyping refers to an initial stage of a software release in which developmental evolution and product fixes may occur before a bigger release is initiated. These kinds of activities can also sometimes be called a beta phase or beta testing, where an initial project gets evaluated by a smaller class of users before full development.

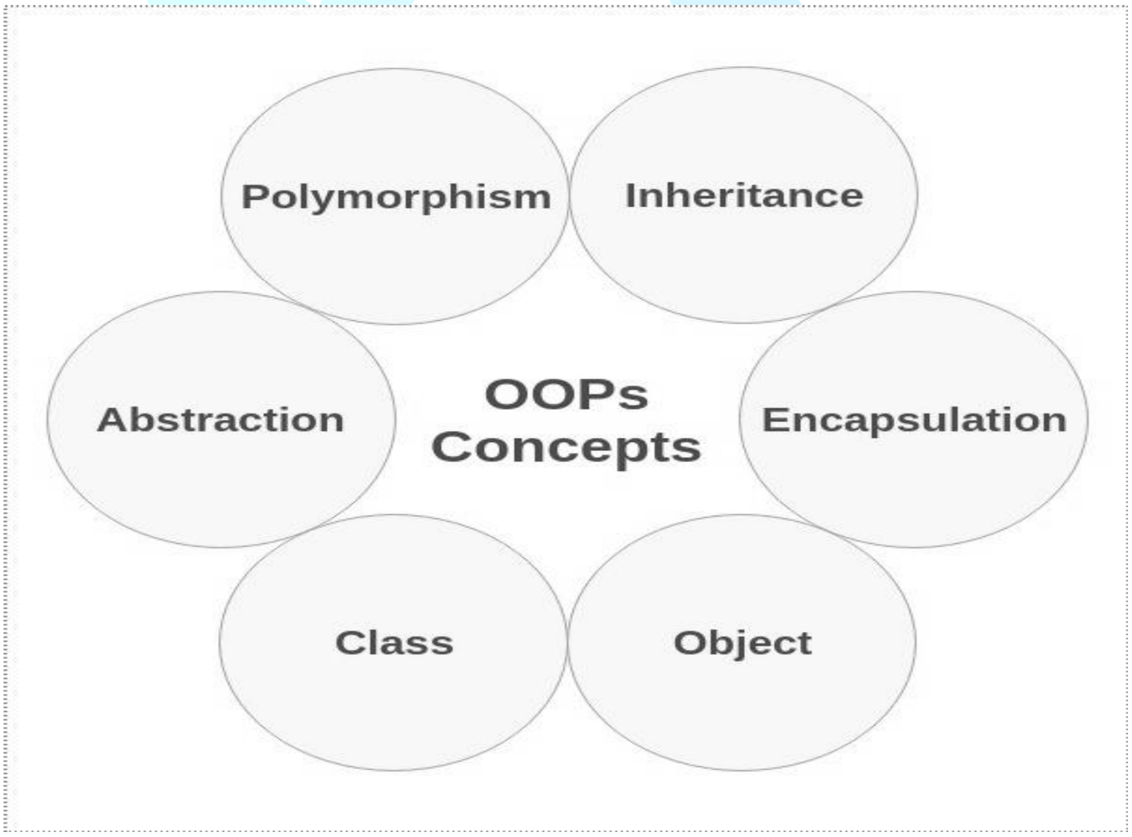
Prototyping is defined as the process of developing a working replication of a product or system that has to be engineered. It offers a small scale facsimile of the end product and is used for obtaining customer feedback

Object Oriented Programming in C++

TABLE OF CONTENT:

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.

Characteristics of an Object Oriented Programming language



Class: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour 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 blue-print representing a group of objects which shares some common properties and behaviours.

For example, lets say we have a class Car which has data members (variables) such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc. Now lets say I create a object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly we can create as many objects as we want using the blueprint(class).

```
//Class name is Car
class Car
{
   //Data members
   char name[20];
   int speed;
   int weight;

public:
   //Functions
   void brake(){
   }
   void slowDown(){
   }
};

int main()
{
  //ford is an object
  Car ford;
}
```

Object: An Object is an identifiable entity with some characteristics and behaviour. 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.

```
class person
{
   char name[20];
   int id;
public:
   void getdetails(){}
};

int main()
{
  person p1; // p1 is a object
}
```

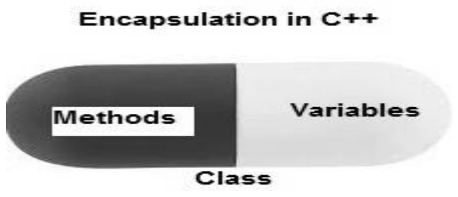
Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation: In normal terms, Encapsulation is defined as wrapping up of 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".



Encapsulation also leads to data abstraction or hiding. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

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

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 characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

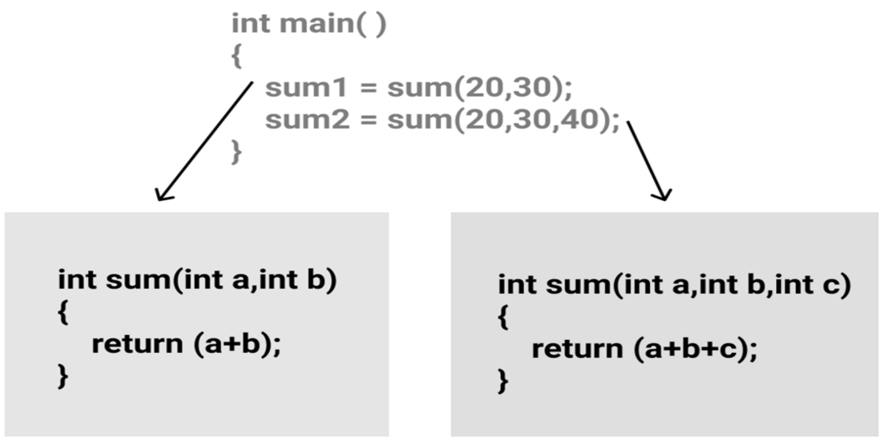
An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading:* The process of making an operator to exhibit different behaviours 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, some times there are 2 integers, some times 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.



```
int main( )
{
   sum1 = sum(20,30);
   sum2 = sum(20,30,40);
}
```

```
int sum(int a,int b)
{
   return (a+b);
}
```

```
int sum(int a,int b,int c)
{
   return (a+b+c);
}
```

```
#include <iostream>
using namespace std;
class Sum {
  public:
    int add(int num1,int num2){
      return num1 + num2;
    }
    int add(int num1, int num2, int num3){
      return num1 + num2 + num3;
    }
};
int main(void) {
   //Object of class Sum
   Sum obj;

   //This will call the second add function
   cout<<obj.add(10, 20, 30)<<endl;

   //This will call the first add function
   cout<<obj.add(11, 22);
   return 0;
}
```

Output:

```
60
33
```

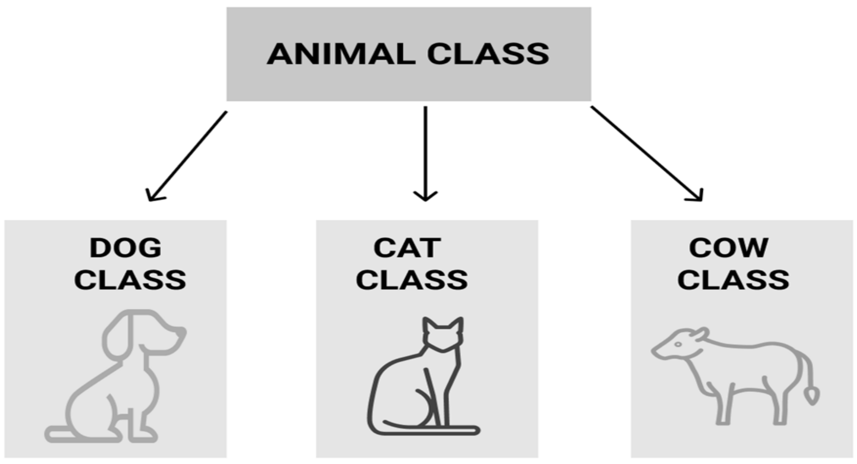
Inheritance: The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

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

Super Class:The class whose properties are inherited by sub class is called Base Class or Super class.

Reusability: 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.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



```
#include <iostream>
using namespace std;
class ParentClass {
  //data member
  public:
    int var1 =100;
};
class ChildClass: public ParentClass {
  public:
  int var2 = 500;
};
int main(void) {
  ChildClass obj;
}
```

Now this object obj can use the properties (such as variable var1) of ParentClass.

Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

C++ Benefits of OOP's

- Through inheritance, we can eliminate redundant code and extend the use of existing classes which is not possible in procedure oriented approach.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch which happens procedure oriented approach. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects .
- The data-centered design approach enables us to capture more details of a model in implementable from.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.
- C++ is a highly portable language and is often the language of selection for multi-device, multi-platform app development.
- C++ is an object-oriented programming language and includes concepts like classes, inheritance, polymorphism, data abstraction, and encapsulation which allow code reusability and makes programs very maintainable.
- C++ use multi-paradigm programming. The Paradigm means the style of programming .paradigm concerned about logics, structure, and procedure of the program. C++ is multi-paradigm means it follows three paradigm Generic, Imperative, Object Oriented.
- It is useful for the low-level programming language and very efficient for general purpose.
- C++ gives the user complete control over memory management. This can be seen both as an advantage and a disadvantage as this increases the responsibility of the user to manage memory rather than it being managed by the Garbage collector.
- The wide range of applications: From GUI applications to 3D graphics for games to real-time mathematical simulations, C++ is everywhere.
- C++ has a huge community around it. Community size is important, because the larger a programming language community is, the more

support you would be likely to get. C++ is the 6th most used and followed tag on StackOverflow and GitHub.

- C++ has a very big job market as it is used in various industries like finance, app development, game development, Virtual reality, etc.
- C++'s greatest strength is how scalable it could be, so apps that are very resource intensive are usually built with it. As a statically written language, C++ is usually more performant than the dynamically written languages because the code is type-checked before it is executed.
- Compatibility with C: C++ is compatible with C and virtually every valid C program is a valid C++ program.

C vs. C++

Similarities between C and C++ are:

- Both the languages have a similar syntax.
- Code structure of both the languages are same.
- The compilation of both the languages is similar.
- They share the same basic syntax. Nearly all of C's operators and keywords are also present in C++ and do the same thing.
- C++ has a slightly extended grammar than C, but the basic grammer is the same.
- Basic memory model of both is very close to the hardware.
- Same notions of stack, heap, file-scope and static variables are present in both the languages.

Differences between C and C++ are:

C++ can be said a superset of C. Major added features in C++ are Object-Oriented Programming, Exception Handling and rich C++ Library. Below is the table of differences between C and C++:

C	C++
C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.
C does no support polymorphism, encapsulation, and inheritance which means that C does not support object oriented programming.	C++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language.
C is a subset of C++.	C++ is a superset of C.
C contains 32 keywords.	C++ contains 52 keywords.
For the development of code, C supports procedural programming.	C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.
Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in form of an object in C++.
C does not support information hiding.	Data is hidden by the Encapsulation to ensure that data structures and operators are used as intended.

Built-in data types is supported in C.	Built-in & user-defined data types is supported in C++.
C is a function driven language because C is a procedural programming language.	C++ is an object driven language because it is an object oriented programming.
Function and operator overloading is not supported in C.	Function and operator overloading is supported by C++.
C is a function-driven language.	C++ is an object-driven language
Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Namespace features are not present inside the C.	Namespace is used by C++, which avoid name collisions.
Header file used by C is stdio.h.	Header file used by C++ is iostream.h.
Reference variables are not supported by C.	Reference variables are supported by C++.
Virtual and friend functions are not supported by C.	Virtual and friend functions are supported by C++.
C does not support inheritance.	C++ supports inheritance.
Instead of focusing on data, C focuses on method or process.	C++ focuses on data instead of focusing on method or procedure.
C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
Direct support for exception handling is not supported by C.	Exception handling is supported by C++.
scanf() and printf() functions are used for input/output in C.	cin and cout are used for input/output in C++.

Object Oriented Languages

Depending upon the features they support, they can be classified into the following two categories:

1. Object –based programming languages and
2. Object-oriented programming lamguages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major futures that are required for object-based programming are:

- Date encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Object –based programming languages do not support inheritance and dynamic binding.
Object-oriented programming incorporates all of Object-based programming features along with two additional futures, namely, inheritance and dynamic binding

Object-based features + Inheritance + Dynamic Binding

Languages that support these futures include C++, Smalltalk, Object Pascal and Java.

Object Based languages are different from Object Oriented Languages:

Object Based Languages

- Object based languages supports the usage of object and encapsulation.
- They does not support inheritance or, polymorphism or, both.
- Object based languages does not supports built-in objects.
- Javascript, VB are the examples of object bases languages.

Object Oriented Languages

- Object Oriented Languages supports all the features of Oops including inheritance and polymorphism.
- They support built-in objects.
- C#, Java, VB. Net are the examples of object oriented languages.

Here are the significant difference between Object-oriented Programming Language and Object-based Programming Language:

Object-based Programming Language	***Object-oriented Programming Language***
All characteristics and features of object-oriented programming, such as inheritance and polymorphism are not supported.	All the characteristics and features of object-oriented programming are supported.
These type of programming languages have built-in objects. Example: JavaScript has a window object.	These type of programming languages don't have a built-in object. Example: C++.
VB is another example of object-based language as you can create and use classes and objects but inheriting classes is not supported.	Java is an example of object-oriented programing language which supports creating and inheriting (which is reusing of code) one class from another.

Applications of Object Oriented Programming

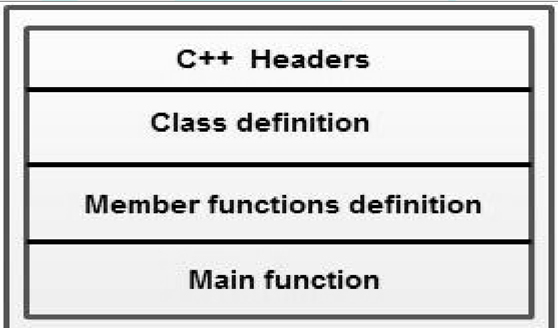
- OOP has become one of the programming buzz words today. There appears to be a great deal of excitement and interest among software engineers in using OOP.
- Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques. Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem.

Main application areas of OOP are:

- User interface design such as windows, menu
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- Hypertext, hypermedia and expertext
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc
- CIM/CAM/CAD systems

Structure of C++ Program

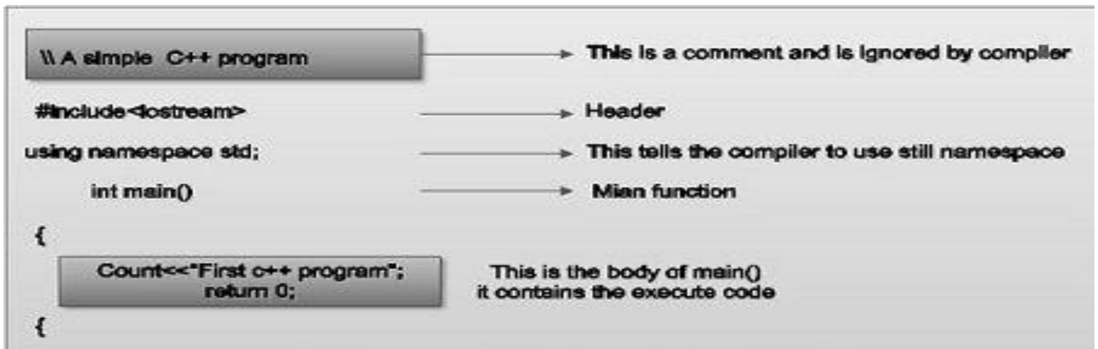
Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely, headers, class definition, member functions definitions and main function.



Structure of a C++ Program

writing a program with or without a class and its member functions definitions. A simple C++ program (without a class) includes comments, headers, namespace, main() and input/output statements.

Comments are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and hence, do not increase the size of a file.



```
\\ A simple  C++ program            This is a comment and is ignored by compiler
#include<iostream>                  Header
using namespace std;                This tells the compiler to use still namespace
     int main()                     Mian function
{
   Count<<"First c++ program";      This is the body of main()
          return 0;                 it contains the execute code
{
```

A simple c++ program(without a class)

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```cpp
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
}
```

```
Hello World!
```

```cpp
// my second program in C++
#include <iostream>

int main ()
{
  std::cout << "Hello World! ";
  std::cout << "I'm a C++ program";
}
```

```
Hello World! I'm a C++ program
```

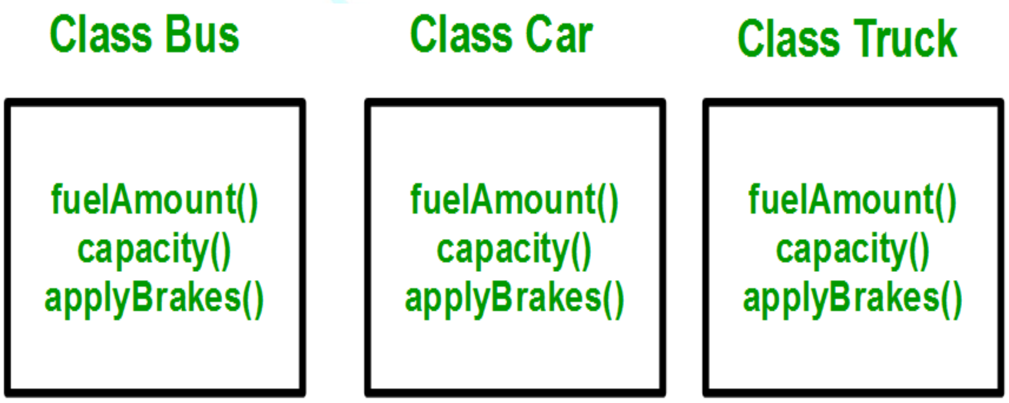
Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

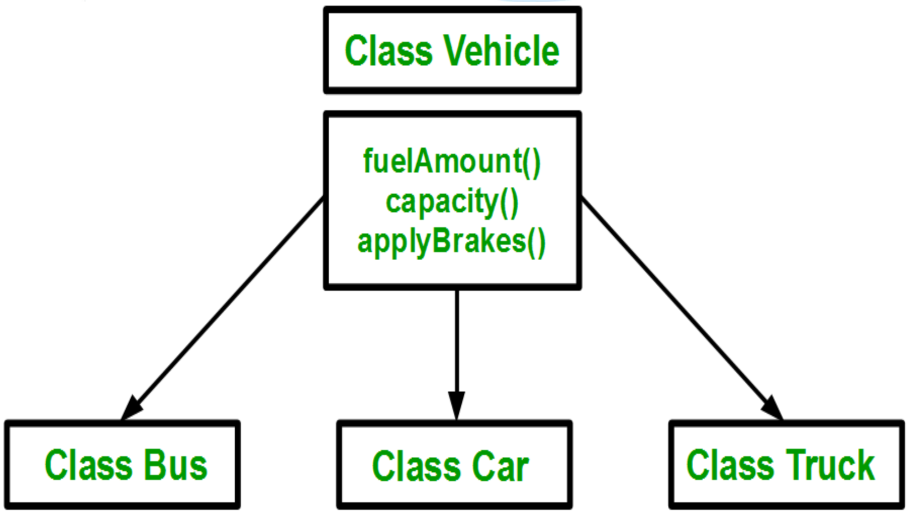
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the 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 in below figure:



You can clearly see that above process results in duplication of 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:



```
// C++ program to demonstrate implementation of Inheritance

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

//Base class
class Parent
{
    public:
      int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
      int id_c;
};

//main function
int main()
   {

        Child obj1;

        // An object of class child has all data members
        // and member functions of class parent
        obj1.id_c = 7;
        obj1.id_p = 91;
        cout << "Child id is " <<  obj1.id_c << endl;
        cout << "Parent id is " <<  obj1.id_p << endl;

        return 0;
   }
```

Output:

```
Child id is 7
Parent id is 91
```

In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Modes of Inheritance

Public mode: If we derive a sub class 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 derived class.

Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in 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 below example. It is just question of access.

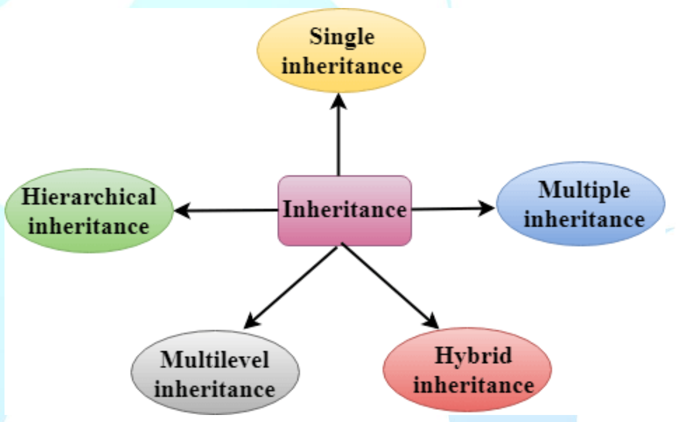
```
// C++ Implementation to show that a derived class doesn't inherit access to private
//data members. However, it does inherit a full parent object
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

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
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types Of Inheritance

C++ supports five types of inheritance:

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



<u>Derived Classes</u>

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```
Where,
derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

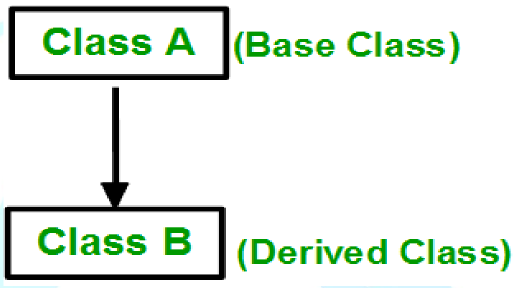
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

1) <u>C++ Single Inheritance</u>

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

Syntax:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};

// C++ program to explain Single inheritance
#include <iostream>
using namespace std;
```

```
// base class
        class Vehicle {
          public:
            Vehicle()
            {
              cout << "This is a Vehicle" << endl;
            }
        };
// sub class derived from two base classes
        class Car: public Vehicle{

        };

        // main function
        int main()
        {
            // creating object of sub class will
            // invoke the constructor of base classes
            Car obj;
            return 0;
        }
```

Output:

```
This is a vehicle
```

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

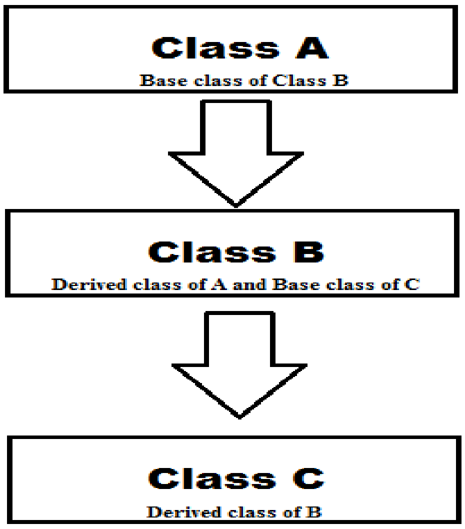
```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

```
Eating...
Barking...
```

2) C++ Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class.



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

Example 1:

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;
 // first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};
```

```
// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};
int main()
{
    // creating object of sub class will invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

Example 2:

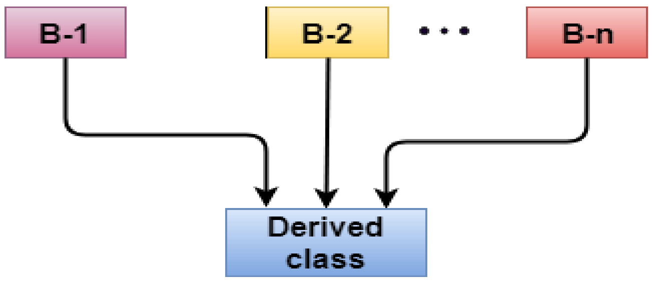
```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {
       public:
     void weep() {
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

3) C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

Let's see a simple example of multiple inheritance.

Example 1:

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;
 // base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
        cout<<"Car has 4 Wheels"<<endl;
     }
};
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

Example 2:

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};
class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
```

```
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();
   return 0;
}
```

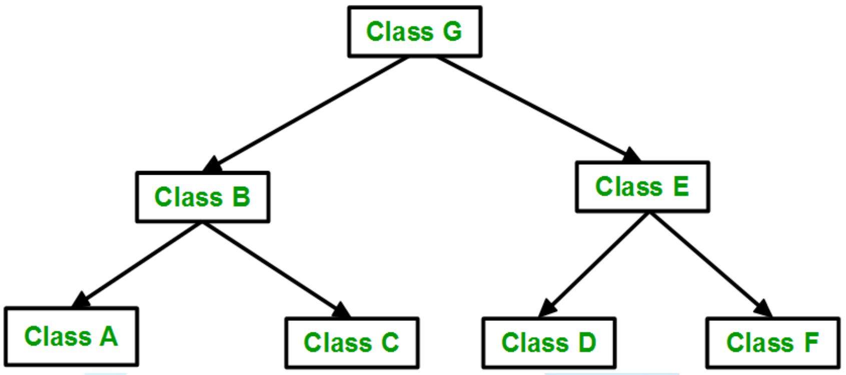
Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

4) C++ Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax of Hierarchical inheritance:

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Let's see a simple example:
Example 1:

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// 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
```

Example 2:

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
```

```
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
int main()
{
   Rectangle r;
   Triangle t;
   int length,breadth,base,height;
   std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   std::cout << "Area of the rectangle is : " <<m<< std::endl;
   std::cout << "Enter the base and height of the triangle: " << std::endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   std::cout <<"Area of the triangle is : "  << n<<std::endl;
   return 0;
}
```

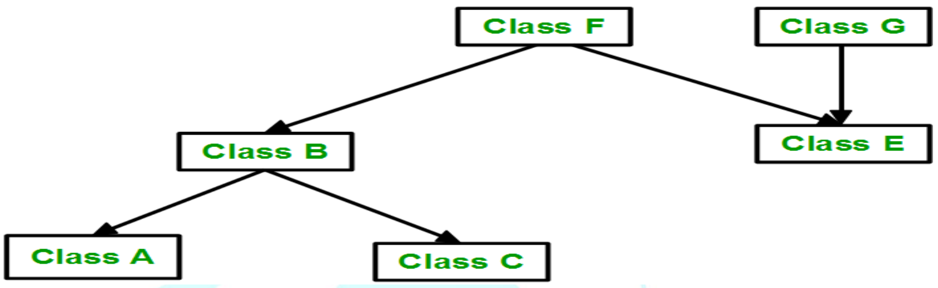
Output:

```
Enter the length and breadth of a rectangle:
23
20
```

```
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

5) C++ Hybrid (Virtual) Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



Let's see a simple example:

Example 1:

```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
// first sub class
```

```
class Car: public Vehicle
{

};
 // second sub class
class Bus: public Vehicle, public Fare
{

};
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

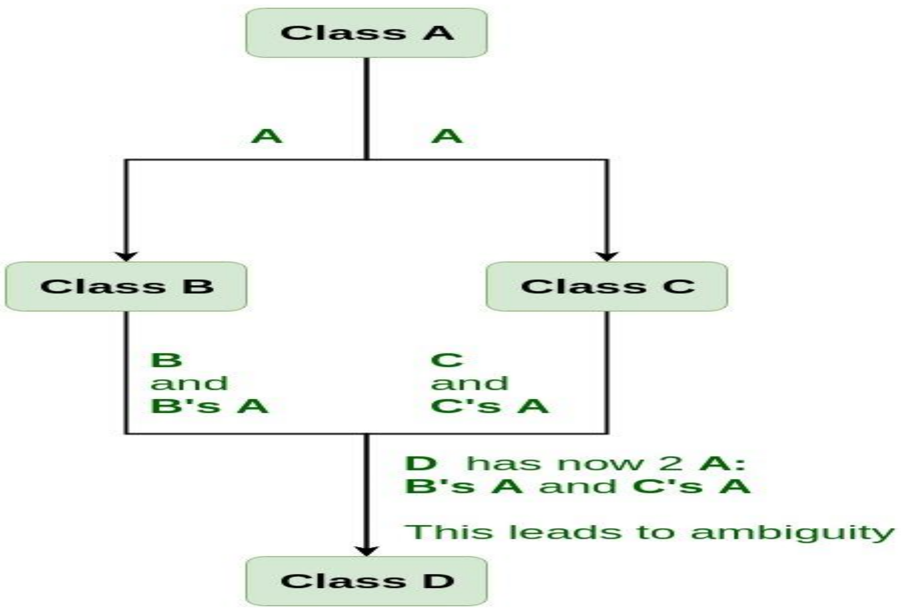
```
This is a Vehicle
Fare of Vehicle
```

Example 2:

```
#include <iostream>
using namespace std;
class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       std::cout << "Enter the value of 'a' : " << std::endl;
       cin>>a;
    }
};
class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};
 class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Virtual base class

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.



Need for Virtual Base Classes:
Consider the situation where we have one class A .This class is A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.

As we can see from the figure that data members/function of class A are inherited twice to class D. One through class B and second through class C. When any data / function member of class A is accessed by an object of class D, ambiguity arises as to which data/function member would be called? One inherited through B or the other inherited through C. This confuses compiler and it displays error.

Example 1
```
#include <iostream>
using namespace std;
class A {
public:
    int a;
    A() // constructor
    {
        a = 10;
    }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;
      return 0;
}
```

Output:
```
a = 10
```

Explanation : The class A has just one data member a which is public. This class is virtually inherited in class B and class C. Now class B and class C becomes virtual base class and no duplication of data member a is done.

Example 2:
```
#include <iostream>
using namespace std;

class A {
public:
    void show()
    {
        cout << "Hello from A \n";
    }
};
 class B : public virtual A {
};
 class C : public virtual A {
};
  class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

Output:

```
Hello from A
```

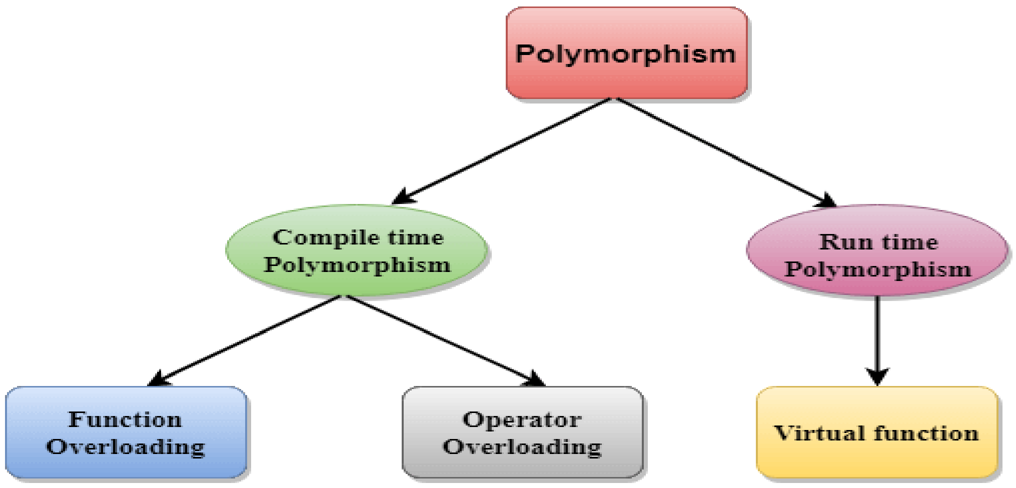
<u>Polymorphism</u>

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



Compile time polymorphism: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
class A                              //  base class declaration.
  {
       int a;
       public:
       void display()
       {
            cout<< "Class A ";
        }
  };
class B : public A                      //  derived class declaration.
{
    int b;
    public:
   void display()
  {
       cout<<"Class B";
  }
};
```

In the above case, the prototype of display() function is the same in both the base and derived class. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as run time polymorphism.

Run time polymorphism: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	**Run time polymorphism**
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.

Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {           cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

```
// an example with virtual keyword.

#include <iostream>
using namespace std;
class Shape {                                  //  base class
    public:
virtual void draw(){                          // virtual function
cout<<"drawing..."<<endl;
    }
};
class Rectangle: public Shape                   //  inheriting Shape class.
{
 public:
 void draw()
   {
       cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape                      //  inheriting Shape class.

{
 public:
 void draw()
   {
      cout<<"drawing circle..."<<endl;
   }
};
int main(void) {
    Shape *s;                           //  base class pointer.
    Shape sh;                           // base class object.
       Rectangle rec;
        Circle cir;
      s=&sh;
     s->draw();
        s=&rec;
     s->draw();
    s=?
    s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {                              //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                   // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
    Animal d= Dog();
   cout<<d.color;
}
```

Output:

```
Black
```

Virtual Function

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1) Virtual functions cannot be static and also cannot be a friend function of another class.
2) Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3) The prototype of virtual functions should be same in base as well as derived class.

4) They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5) A class may have virtual destructor but it cannot have a virtual constructor.

```
// CPP program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;
 class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show()
    {
        cout << "show base class" << endl;
    }
};
class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};
int main()
{
    base* bptr;
    derived d;
    bptr = &d;
   // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output:

```
print derived class
show base class
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

Output:

```
Derived class is derived from the base class.
```

VALLUVAR COLLEGE OF SCIENCE AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE

Important Questions

Unit – I:

Basic Concepts of Object- Oriented Programming - Benefits of OOP – Object Oriented Languages - Applications of OOP – Structure of C++ Program - Tokens, Expressions and Control Structures – Functions in C++

2 Marks Questions

1. What are the characteristics of procedure oriented programming?
2. Write the principal advantages of Object Oriented Programming?
3. Define Object.
4. What is Object Oriented Programming?
5. What is Class?
6. Write few words about Object Oriented Languages.
7. List the applications of object oriented programming?
8. What is C++? in PHP?
9. Give the structure of C++ program.
10. Write note on iostream File.
11. What is Namespace?
12. What are tokens in C++?
13. What is a variable in C++? How is it declared?
14. Enumerate the rules of naming variables in C++.
15. Write about C++ data types?
16. What are the new operators in C++?
17. What is scope resolution operator?
18. What are manipulators?
19. Write about Type Cast Operator.
20. What is Function Overloading?
21. Define function prototyping?
22. What is inline function?
23. What is data abstraction?
24. Explain GoTo Statement.

5 Marks Questions

1. Write short notes on object oriented languages.
2. What is inline function? Explain with an example.
3. Explain the basic data types in C++
4. Explain C++ tokens briefly.
5. List some of the striking features of OOPs.
6. What is scope resolution operator? Explain with an example.
7. Explain the Output and Input Operators in C++.
8. Write about User defined data types used in C++ with examples.

10 Marks Questions

1. Write about
   a) Basic data type
   b) User defined data type used in C++ with examples
2. Explain the basic concept of object oriented programming.
3. Explain the oops concepts with its benefits.

Unit – II:

Classes and Objects – Constructors and Destructors –Operator Overloading and Type Conversions

2 Marks Questions

1. Define Classes and Objects.
2. Distinguish between Object and Classes.
3. What is Static member function?
4. What is Friend function in C++?
5. List the special characteristics of friend function.
6. What is meant by Constructors and list various types of Constructors?
7. What is a Parameterized Constructor?
8. What is copy constructor?
9. What is meant by Destructors? Give example.
10. Describe the importance of destructor.
11. How is a member function of a class defined outside?
12. Write a program to overload unary minus operator.
13. Give any two rules of Operator Overloading.
14. What do you mean by type conversion?

5 Marks Questions

1. Explain classes and object with example.
2. Write short notes on Type conversion.
3. Explain the concepts of class in C++.
4. What is destructor? Write a program to explain the concept of destructor.
5. What is operator overloading? Explain how binary operator can be overloaded with example.
6. Explain about multilevel constructor with example in C++.
7. Explain about Defining Member Functions.
8. List the Rules for Overloading Operators.

10 Marks Questions

1. What is a friend function? Explain the friend function with and example and also list its characteristics.
2. Discuss various types of constructors in C++.
3. What is operator overloading? Explain with example program.

Unit – III:

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

2 Marks Questions

1. What is derived class?
2. What is Inheritance mean in C++?
3. What are the different types/forms of Inheritance?
4. What is a virtual base class in C++?
5. What is meant by Pointers? Give example.
6. Explain This Pointer?
7. List any two uses of pointers.
8. Define virtual function.
9. What is a pure function?
10. What is Pure Virtual Functions?
11. What is an abstract class in C++?
12. Define Polymorphism.
13. What are the different types/forms of Polymorphism?

5 Marks Questions

1. Write the visibility of inherited members in C++.
2. Explain the usage of virtual function with example.
3. Explain multilevel inheritances with example.
4. Explain pure virtual function with syntax and example.
5. Write short notes on polymorphism.
6. Describe how an object of a class that contains objects of others classes created with an example.
7. Explain the uses of pointers in C++.
8. Explain Virtual Function in detail.

10 Marks Questions

1. Explain the different ways by which we can access public member function of object with examples.
2. Distinguish multiple and multilevel inheritance with example.
3. What are abstract classes? Explain their use.
4. Write short notes on various types of inheritance with example.
5. Define Polymorphism. Explain the types of Polymorphism with suitable example.
6. What is Inheritance? Explain Multilevel Inheritance and Hybrid Inheritance with example.

Unit – IV:

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

2 Marks Questions

1. What is stream?
2. Explain the classes for file stream operation.
3. What is an iostream class?
4. How will be the I/O structure in C++?
5. Give any two features of I/O system supported by C++.
6. What is a file?
7. What is meant by file input and output streams?
8. Write a program to copy a content of file to another file.
9. List out any two manipulators and their meanings.
10. What is an exception?
11. Give the rules of Exception Handling.
12. What are the advantages of using exception handling mechanism in C++?
13. What is Generic Programming?

5 Marks Questions

1. Write short notes on console I/O operations.
2. Explain exception handling in C++.
3. Write short notes on class template.
4. Explain about unformatted input function with suitable example.
5. Discuss file stream classes in detail.
6. Write general format of function template and write a program to swap two values using function template.
7. Distinguish between overloaded function and function templates
8. How is an exception handle in C++? Explain with example.

10 Marks Questions

1. Discuss about sequential input and output operations on files in C++.
2. What is exception handling? Explain the basics of exception handling with example.
3. Explain formatted console I/O operations.

Unit – V:

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

2 Marks Questions

1. What is STL? What are its components?
2. Explain iterators.
3. List the types of containers.
4. List the types of Iterators.
5. Draw the relationship between the three STL components.
6. Write three most popular container classes and its usage.
7. What is String?
8. What is string class?
9. Give any two examples of manipulation of strings.
10. Write commonly used string constructors?
11. What is data flow diagram?
12. Draw the classic software development life cycle.
13. What is prototyping?

5 Mark Questions

1. Write short notes on STC.
2. Write various system development tools
3. Write short notes on applications of container classes.
4. Write a C++ program to compare two strings.
5. Explain containers support by the STL.
6. Discuss the steps used in object oriented design.
7. What is an algorithm? How STL algorithms are different from the conventional algorithms.
8. Explain the steps in object oriented design.

10 Marks Questions

1. Distinguish between object oriented paradigms and procedure oriented paradigms.
2. Explain object oriented analysis.
3. Discuss in detail procedure oriented paradigms and object oriented paradigms.
4. Describe about manipulating strings in C++.

All the Best

PROGRAMMING IN C++

<u>Activity - I</u>

I-B.SC (CS) **16.04.2020**

1) Write a C++ program to Calculate Factorial of a Number Using Recursion
2) Write a C++ Program to Multiply Two Matrix Using Multi-dimensional Arrays
3) Write a C++ Program to Display Fibonacci Series using functions.
4) Write a C++ Program to Find GCD using recursion
5) Write a C++ Program to Reverse a Number
6) Write code to Increment ++ and Decrement -- Operator Overloading in C++ Programming
7) Write a C++ Program to Check Whether a Number is Palindrome or Not
8) Write a C++ Program to Check Prime Number By Creating a Function
9) Write a C++ Program to Check Armstrong Number
10) Write a C++ program to Find Sum of Natural Numbers using Recursion
11) Write a C++ Program to Access Elements of an Array Using Pointer.
12) Write a C++ Program to Find the Number of Vowels, Consonants, Digits and White Spaces in a String.
13) Write a C++ Program to Calculate Difference Between Two Time Period using function
14) Write a C++ Program to Find Largest Element of an Array
15) Write a C++ Program to Calculate Average of Numbers Using Arrays
16) Write a C++ Program to Make a Simple Calculator to Add, Subtract, Multiply or Divide Using switch...case
17) Write a C++ program to swap two numbers without using third variable.
18) Write a C++ program to read and print students information using two classes and simple inheritance
19) Write a C++ program to read and print employee information using multiple inheritance.
20) Write a C++ program to find area of square, rectangle, circle and triangle by using function overloading

<u>Note:</u>

1) Write the above programs using mentioned language concepts (Recursion, inheritance, pointers, array, etc...)

2) While writing the program you have to explain the program concept, for example

→ Prime number: you have to explain what prime number is? Likewise you have to explain GCD, factorial, Fibonacci Series, Armstrong Number, Sum of Natural Numbers, Palindrome and etc...

<u>ALL THE BEST</u>