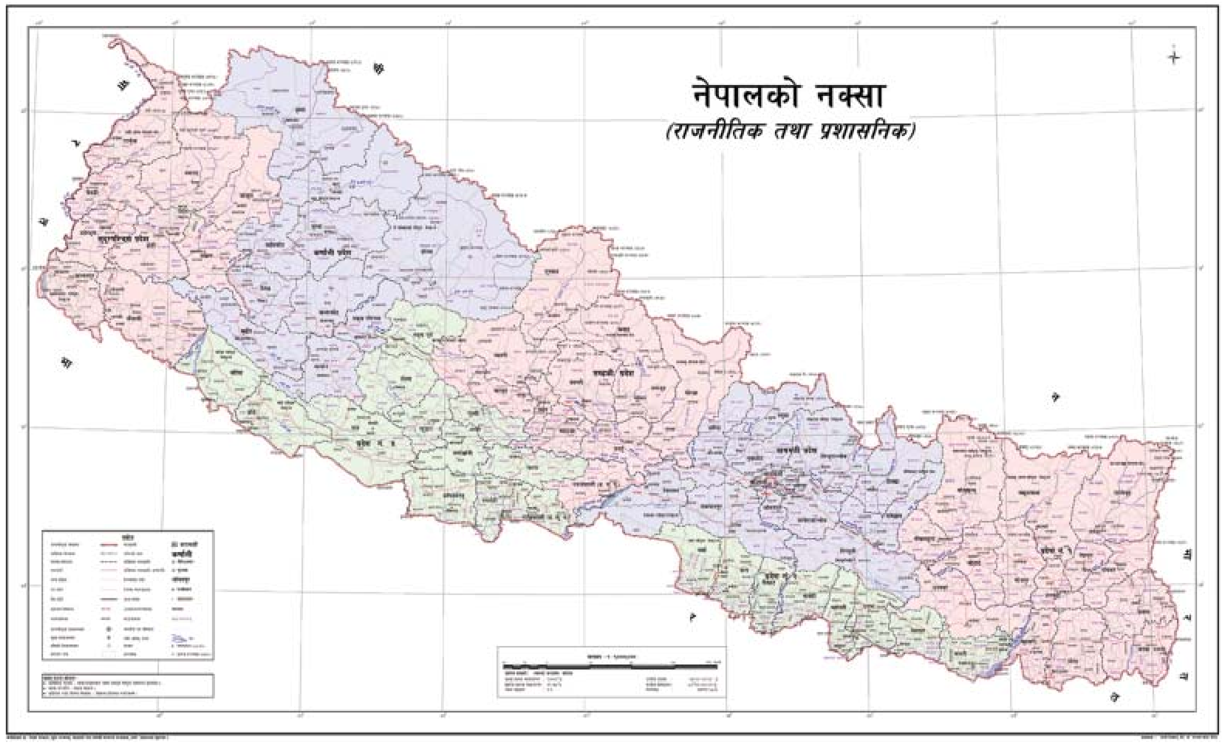


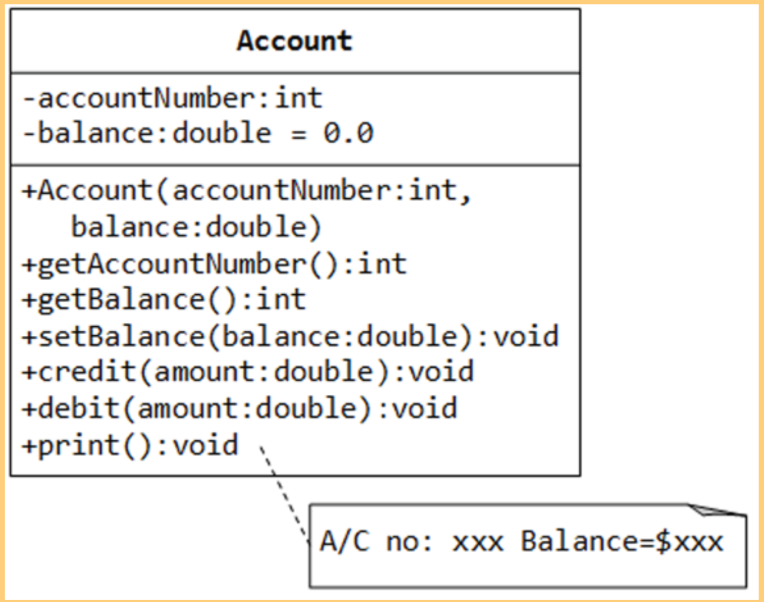
Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur

Phone : 5639122/6634373/6635046/6630088
Website : www.moecdc.gov.np

Computer Engineering

10

Object Oriented Programming



```
Account
-accountNumber:int
-balance:double = 0.0
+Account(accountNumber:int,
   balance:double)
+getAccountNumber():int
+getBalance():int
+setBalance(balance:double):void
+credit(amount:double):void
+debit(amount:double):void
+print():void
```

A/C no: xxx Balance=$xxx

Technical and Vocational Stream

Learning Resource Material

Object Oriented Programming
(Grade 10)

Secondary Level

Computer Engineering

Government of Nepal
Ministry of Education, Science and Technology

Curriculum Development Centre

Sanothimi, Bhaktapur

Publisher : Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur

Layout by Khados Sunuwar

Preface

The curriculum and curricular materials have been developed and revised on a regular basis with the aim of making education objective-oriented, practical, relevant and job oriented. It is necessary to instill the feelings of nationalism, national integrity and democratic spirit in students and equip them with morality, discipline and self-reliance, creativity and thoughtfulness. It is essential to develop in them the linguistic and mathematical skills, knowledge of science, information and communication technology, environment, health and population and life skills. it is also necessary to bring in them the feeling of preserving and promoting arts and aesthetics, humanistic norms, values and ideals. It has become the need of the present time to make them aware of respect for ethnicity, gender, disabilities, languages, religions, cultures, regional diversity, human rights and social values so as to make them capable of playing the role of responsible citizens with applied technical and vocational knowledge and skills. This Learning Resource Material for Computer Engineering has been developed in line with the Secondary Level Computer Engineering Curriculum with an aim to facilitate the students in their study and learning on the subject by incorporating the recommendations and feedback obtained from various schools, workshops and seminars, interaction programs attended by teachers, students and parents.

In bringing out the learning resource material in this form, the contribution of the Director General of CDC Dr. Lekhnath Poudel, Pro, Dr. Subarna Shakya, Bibha Sthapit, Anil Barma, Bhuwan Panta, Yogesh Parajuli, Satyaram Suwal, Asharam Suwal, Shankar Yadav is highly acknowledged. The book is written by Bimal Thapa and the subject matter of the book was edited by Badrinath Timalsina and Khilanath Dhamala. CDC extends sincere thanks to all those who have contributed in developing this book in this form.

This book is a supplimentary learning resource material for students and teachrs. In addition they have to make use of other relevnt materials to ensure all the learning outcomes set in the curriculum. The teachers, students and all other stakeholders are expected to make constructive comments and suggestions to make it a more useful learning resource material.

2076 BS

Ministry of Education, Science and Technology
Curriculum Development Centre

Table of Contents

Unit-1
Overview....1
Learning Outcomes....1
1.1 Procedural programming....1
1.2 Benefits of OOP:....2
1.3 The object-oriented approach:....3
1.5. Difference between C and C++.....7
Unit-2....9
C++ Basic Input/output....9
Learning Outcomes....9
2.1 I/O Library Header Files:....9
2.2. The Standard Output Stream (cout)....10
2.3. The Standard Input Stream (cin)....10
Unit-3....15
Objects and Classes....15
Learning Outcomes....15
3.1. Class –Object Concept....15
3.2. Difference between structures and classes....16
3.3. Accessing members of structures:....17
3.4. Simple class construction/ Declaration of Class....18
3.5. Class Definitions....19
3.6. Class Variables and methods:....19
3.7. Accessing data members and member functions of class:....20
3.8. Access Specifiers....24
3.9. Public, private and protected....26
3.10. Initializing class objects....32
3.11. Constructors and Destructor:....35
3.12. Default copy constructor....41
3.14 Static data member in class....45
3.14 Static Function Members of Class:....47
3.15. Inline Function....49
3.16. Data encapsulation and its example....51
3.17 Passing parameters to a constructor function....54
3.18Difference between constructors and member function:....57
Unit 4....64
Polymorphism....64
Learning Outcomes:-....64
4.1 Introduction to Polymorphism....64
4.2 Function Overriding....67

4.3. Virtual Function....69
4.4. Runtime polymorphism....70
4.5 Static Binding and Dynamic Binding....70
4.6 Abstract class and pure virtual function....74
Unit-5....78
Operator Overloading....78
Learning Outcomes....78
5.1 Overloading unary operators:....82
5.2. Overloading binary operators:....86
Unit 6....93
Inheritance....93
Learning Outcomes....93
6.1. Introduction to inheritance....93
6.2. Basic Concepts:....94
6.3. Base class and Derived class:....95
6.4 Accessing base class members:....97
6.5 Public, Private and Protected Inheritance in C++:....98
6.6 Abstract Base Class:....99
6.7 Forms/Types of inheritance:....100

UNIT-1
Overview

Learning Outcomes

After completion of this unit you will be able to

- To explain/describe the formalism of object oriented programming.
- To explain/describe benefits of OOP.
- To explain/describe object, class, data abstraction and encapsulation, inheritance, polymorphism etc.
- To describe about difference between C and C++.

1.1 Procedural programming

A computer programming language that executes a set of commands in order is called procedural Language. It is written as a list of instructions, telling the computer, step-by-step, what to do. For Eg. Open a file, read a number, multiply by 4, display something. Program units include the main or program block, subroutines, functions, procedures; file scoping; includes/modules; libraries.

Procedural programming is fine for small projects. It is the most natural way to tell a computer what to do, and the computer processor's own language, machine code, is procedural, so the translation of the procedural high-level language into machine code is straightforward and efficient.

Examples of computer procedural languages are BASIC, C, FORTRAN, and Pascal.

Advantages of Procedural Programming:

- Its relative simplicity, and ease of implementation of compilers and interpreters
- The ability to re-use the same code at different places in the program without copying it.
- An easier way to keep track of program flow.
- The ability to be strongly modular or structured.
- Needs only less memory.

Disadvantages of Procedural Programming:

- Data is exposed to whole program, so no security for data.
- Difficult to relate with real world objects.
- Difficult to create new data types reduces extensibility.
- Importance is given to the operation on data rather than the data.

Comparison of Procedural with Object-Oriented Programming:

The focus of procedural programming is to break down a programming task into a collection of data structures and subroutines, whereas in object oriented programming it is to break down a programming task into objects. Both method can be applicable for complete a specific programming task.

The most popular programming languages usually have both OOP and procedural feature.

Some differences between object-oriented and Procedural languages:

Pure Object Oriented	Pure Procedural
Methods	Functions
Objects	Modules
Message	Call
Attribute	Variable

1.2 Benefits of OOP:

OOP offers several benefits to both the program designer and the user.
The principle advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing code.
- Classes.
- 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.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.

1.3 The object-oriented approach:

Definition:

Object oriented programming is a programming methodology that associates data structures with set of operators which act upon it. In OOP, an instance of such an entity is known as object. In other words, OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents and instance of some class are all members of hierarchy of classes united through the property called inheritance.

The OOP is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with powerful new concepts. The object oriented approach is:

- The recent concept among programming paradigms.
- Fundamental idea it to combine data and functions those operate on those data into single unit-called object.
- The data of an object can be accessed only by the function associated with these objects.

In OO Programming

- Emphasis is on data rather than procedures.
- Programs are divided into objects.
- Data is hidden and can't accessed by external functions.
- Data structures are designed such that, they characterize the objects.
- Functions and data are tied together in the data structures so that data abstraction is introduced in addition to procedural abstraction.
- Object can communicate with each other through function.
- New data and function can be easily added.

1.4.1 Objects:

Objects are the entities through which we perceive the world around us. We naturally see our environment as being composed of things which have recognizable identities and behavior. The entities are then represented as objects in the program.

In OO system, an object is the run time entity i.e. a region of storage with associated semantics. In OO/C++ "Object" usually means an instance of a class. In object oriented language the problem be into objects. Where in pop, problem is divided into functions.

Example of objects:

Physical Objects:

- Automobiles in traffic flow simulation
- Countries in Economic model
- Air craft in traffic – control system.

Computer user environment objects:

- Window, menus, icons etc.
- Data storage constructs:
- Stacks, trees etc.

Human entities:

- Employees, student, teacher etc.
- Geometric objects:
- Point, line, triangle etc.

Object mainly serve the following purposes:

- Understanding the real world and a practical base for designers.
- Decomposition of a problem into objects depends on the nature of problem.

1.4.2. Classes:

A class defines a data type, much like a struct in c. It specifies what data and functions will be included in objects of that class. Defining class doesn't create an object but class is the description of object's attributes and behaviors. Thus a class is a collection of objects of similar type e.g. class vehicle includes objects car, bus, etc.

Person Class:	Attributes: Name, Age, Sex, etc.
	Behaviors: Speak(), Listen(), Walk()
Vehicle Class:	Attributes: Name, model, color, height etc.
	Behaviors: Start(),stop(), accelerate()

When class is defined, object are created as

```
<classname> <objectname>;
```

Each class describes a possibly infinite set of individual objects, each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operations with other instances of the class. The following points gives the idea of class:

- A class is a template that unites data and operations.
- A class is an abstraction of the real world entities with similar propertied.
- Ideally, the class is an implementation of abstract data type.

1.4.3. Data Abstraction and Encapsulation:

The wrapping up of data and operations into a single unit is called encapsulation. Encapsulation is most striking feature of a class. The data is not accessible from outside of class. Only member function can access data on that class. The insulation of data from direct access by the program is called data hiding.

Abstraction is representing essential features of an object without including the background details or explanation. It focuses the outside view of an object, separating its essential behavior from its implementation.

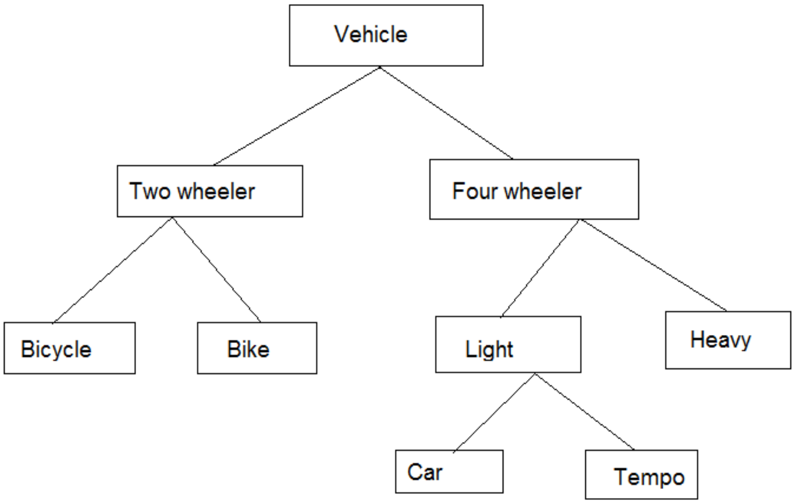
The class is a construct in C ++ for creating user –defined data types called Abstraction Data Types(ADT).

1.4.4. Inheritance:

Inheritance is the process by which objects of one class inherits the characteristics of another class as part of its definition. It supports the concept of hierarchical classification. It allows the extension and reuse of existing code without having to rewrite the code.

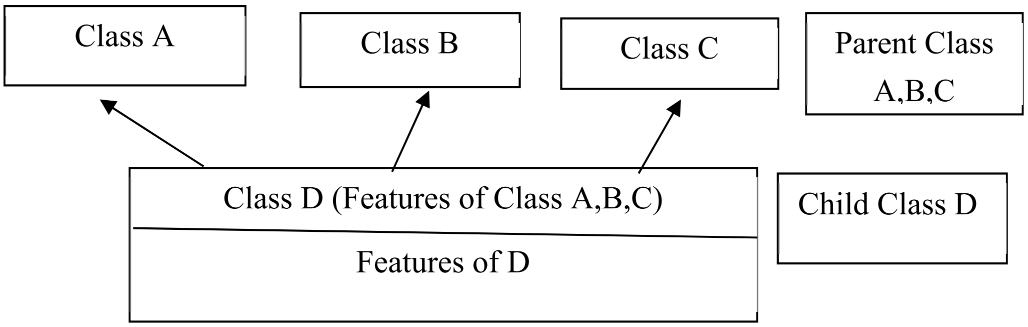
Inheritance is appropriate when a class is a kind of other class. E.g. in OOP, the concept of inheritance provides the idea of reusability. We can use additional features to an existing class without modifying it. This process of deriving a new class from the existing base class is called inheritance.

Example of inheritance Class Hierarchichy



Multiple Inheritances: If derived class inherits the features of more than one base class it is called multiple inheritances.

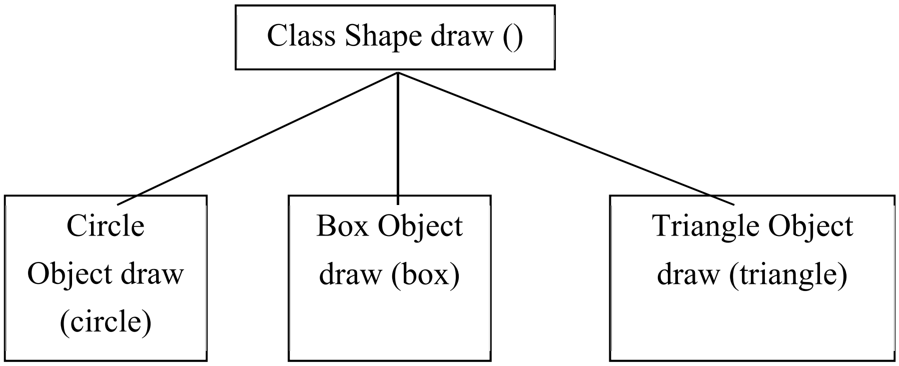
Multiple Inheritances



1.4.5. Polymorphism:

Polymorphism means "having many forms". The polymorphism allows different object to respond to the same message in different ways, the response specific to the type of object. It is important when object oriented programs dynamically creating and destroying the objects in run time example of polymorphism in OOP is operator overloading function overloading etc.

E.g. Operator symbol '+' is used for arithmetic operation between two numbers however by overloading it can be used over complex object like currency (that has Rs and Paisa as its attributes). By overloading same operator '+' can be used for different purpose like concatenation of strings.



1.5. Difference between C and C++.

C	C++
C is procedural language.	C++ is non procedural (object oriented) language.
Polymorphism is not possible.	Polymorphism is the most important feature of OOP.
Operator overloading is not possible in C.	Operator overloading is one of the greatest feature of c++.
Top down approach is used in program design.	Bottom up approach adopted in program design.
No namespace feature is present in C.	Namespace feature is present in C++ for avoiding name collision.
In C Scanf() function used for input Printf() function used for output	In C++ Cin>> function used for input Cout << function used for output
No inheritance is possible in C.	Inheritance is possible in C++.

SUMMARY

- Procedural Language: A computer programming language that executes a set of commands in order is called procedural Language
- Object oriented programming is a programming methodology that associates data structures with set of operators which act upon it
- A class defines a data type, much like a struct in c. It specifies what data and functions will be included in objects of that class. Defining class doesn't create an object but class is the description of object's attributes and behaviors. Thus a class is a collection of objects of similar type.
- Objects are the entities through which we perceive the world around us.
- The wrapping up of data and operations into a single unit is called encapsulation.
- Abstraction is representing essential features of an object without including the background details or explanation.
- Inheritance is the process by which objects of one class inherits the characteristics of another class as part of its definition.
- Polymorphism means "having many forms". The polymorphism allows different object to respond to the same message in different ways, the response specific to the type of object.

Self Evaluation

1. Write very short answer of the following question.

a) What is the full form of OOP?
b) What is a class?
c) What is an object?
d) What is encapsulation?
e) What is abstraction?

2. Write short answer of the following question.

a) What is inheritance? Write its types.
b) What is polymorphism? Write its example.
c) Write advantage and disadvantage of procedural programming.
d) What are the comparison of procedural with object oriented programming?

3. Write long answer of the following question.

a) Why C++ is called object-oriented programming language? Explain
b) What are the difference between C and C++?
c) What is procedural programming? Explain its merits and demerits.

UNIT-2

C++ Basic Input/output

Learning Outcomes

After completion of this unit you will be able to

- To explain/describe input output library header files.
- To explain/describe io stream.
- To explain/describe io mainp, Fstream etc.
- To describe the standard input stream(cin) and output stream(cout).

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called input operation and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called output operation.

2.1 I/O Library Header Files:

There are following header files important to C++ programs −

S.No	Header File & Function and Description
1	<iostream> This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2	<iomanip> This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.
3	<fstream> This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

2.2. The Standard Output Stream (cout)

The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
 using namespace std;
 int main() {
   char str[] = "Hello C++";
   cout << "Value of str is : " << str << endl;
}
```

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

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and endl is used to add a new-line at the end of the line.

2.3. The Standard Input Stream (cin)

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
 using namespace std;
 int main() {
   char name[50];
  cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

```
cin >> name >> age;
```

This will be equivalent to the following two statements −

```
cin >> name;
cin >> age;
```

<iosfwd>	forward declarations of all classes in the input/output library
<ios>	std::ios_base class, std::basic_ios class template and several typedefs
<istream>	std::basic_istream class template and several typedefs
<ostream>	std::basic_ostream, std::basic_iostream class templates and several typedefs
<iostream>	several standard stream objects
<fstream>	std::basic_fstream, std::basic_ifstream, std::basic_ofstream class templates and several typedefs
<sstream>	std::basic_stringstream, std::basic_istringstream, std::basic_ostringstream class templates and several typedefs
<syncstream> (since C++20)	std::basic_osyncstream, std::basic_syncbuf, and typedefs
<strstream>(deprecated)	std::strstream, std::istrstream, std::ostrstream
<iomanip>	Helper functions to control the format or input and output
<streambuf>	std::basic_streambuf class template

<cstdio>	C-style input-output functions

C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Following table lists these error handling functions and their meaning:

Function	Meaning
int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures. For example :

```
:
ifstream fin;
fin.open("master", ios::in);
while(!fin.fail())
{
        :       // process the file
}
if(fin.eof())
{
        :        // terminate the program
}
else if(fin.bad())
{
        :        // report fatal error
}
else
{
        fin.clear();     // clear error-state flags
        :
}
:
```

SUMMARY

- iostream:This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
- iomanip:This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.

- Fstream:This file declares services for user-controlled file processing.
- Standard Output Stream (cout): The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen.
- The cin object is said to be attached to the standard input device, which usually is the keyboard.

Self Evaluation

1. **Write very short answer of the following question.**

a) What is io stream?

b) What is the use of insertion operator?

2. **Write short answer of the following question.**

a) What are I/O library header files with their functions?

b) What do you mean by Standard Output Stream (cout)? Explain it.

c) What do you mean by Standard input Stream (cin)? Explain it.

d) Mention four errors that handle the functions during file operation.

e) Write a program to display the following output using single cout statement.

f) DBMS=80

g) Network=90

h) C++=95

i) Write a program to input length and breadth of a room and calculate the area.

UNIT-3

Objects and Classes

Learning Outcomes

After completion of this unit you will be able to

- To explain/describe Class-Object Concept.
- To explain/describe difference between class and structures.
- To explain/describe accessing members of structures.
- To describe simple class construction; defining class, class variables and methods; accessing data members and member functions of class etc.
- To explain/describe access specifies, public, private and protected.
- To explain/describe initializing class objects, Constructors and Destructor, Default copy constructor etc.
- To explain/describe Static data member and member function of a class.
- To explain/describe Inline Function and passing parameters to a constructor function.
- To explain/describe Data encapsulation and its example.
- To explain/describe difference between Constructors and member function.

3.1. Class –Object Concept

There are various objects like book, box, computer, furniture etc. in real world. The computers in our school lab or our home are similar objects. These different computers may be manufactured by different companies in different models. They have common characteristics to identify them. For example: all have memory, hard disk, monitor, keyboard etc.

The general or common name to describe objects of common or similar characteristics and behaviors is called class. Thus the name 'computer' may be class name which represents my personal computer, lab computer and other computers. A class is blue print or template which describes characteristics of similar objects.

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

The data types such as int, float, double may be class which represent similar data items. For eg. Int represents all integer numbers such as 1, 2, 3,4 and so on. So, int is general name for representation of any integer values. In OOP, we can define class name to represent its individual objects. Defining a class in programming language means creating user defined data type and it behaves like the built-in data types. Once a class has been defined, we can create any number of objects belonging to that class. An object is an instance of a class. A class is a description of a group of objects with common properties (attributes or characteristics or variables), behavior (operations or methods) and relationships. The possible classes may be book, student, employee, bus, computer, vehicle, account and so on.

3.2. Difference between structures and classes

Structure	Class
It is a value type.	It is a reference type.
Its object is created on the stack memory.	Its object is created on the heap memory.
It does not support inheritance.	It supports inheritance.
The member variable of structure cannot be initialized directly.	The member variable of class can be initialized directly.
It can have only parameterized constructor.	It can have all the types of constructor and destructor.

Syntax
Struct structure_name
{
    Data_member;
};
	Syntax
	Class class_name
	{
	    data_member;
	    function_members;
	};

3.3. Accessing members of structures:

The members of structure are usually processed individually, as separate entity. Therefore, we must be able to access the individual structure members. A structure member can be accessed by using period or dot (i.e. ".") operator. The syntax for accessing member of a structure variable is as follows:

Structure_variable.member

Here, structure_variable refers to the name of a structure-type variable and member refers to the name of a member within the structure. Again, dot separates the variable name from the member name. The dot operator must have a structure variable on its left and a legal member on its right. In the case of nested structure (if a structure member is itself a structure), the member within inner structure is accessed as

Structure_variable.member.submember

i.e., structure_variable.member_of_outer_sturucture.member_of_inner_sturcture
Here is an example, demonstrating how to access members of a structure in C++

```
/* C++ Access Structure Member */

#include<iostream.h>
#include<conio.h>

struct st
{
        int a;    // structure member
        int b;    // structure member
        int sum;  // structure member
}st_var;  // structure variable

void main()
{
        clrscr();
        cout<<"Enter any two number:\n";
        // accessing structure member a and b
        cin>>st_var.a>>st_var.b;
        // accessing structure member sum, a, and b
        st_var.sum = st_var.a + st_var.b;
        // accessing structure member sum
        cout<<"\nSum of the two number is "<<st_var.sum;
        getch();
}
//output
```

```
Enter any two number:
3
4
Sum of the two number is 7
```

3.4. Simple class construction/ Declaration of Class

A class is used to specify blue print of similar objects and it combines data and methods for manipulating that data. The data and functions within a class are called members of the class.

For example, using the keyword **class** as follows –

```
        Class class_name
        {
        private:
                Variable declaration;
                Function declaration;
        public:
                Variable declaration;
                Function declaration;
        protected:
                Variable declaration;
                Function declaration;
        };
```

Here, class is a C++ keyword; class _name is name of class defined by user. The body of class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These function and variables are collectively called as members. They are grouped under two sections, private and public. By using the keyword "private" we can hide data which can be accessed by functions defined within the class. The keyword "public" is used so that public members can be accessed by any function even outside the class. The keyword private and public are known as visibility labels. By default, the members of class are private. In OOP, generally date are made private and functions are made public.

3.5. Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

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

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

The keyword public determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

3.6. Class Variables and methods:

A variable must be defined before you can use it in a program. When you define a variable the type is specified and an appropriate amount of memory reserved. This memory space is addressed by reference to the name of the variable. A simple definition has the following syntax:

SYNTAX: typ name1 [name2 ...];

This defines the names of the variables in the list name1 [, name2 ...] as variables of the type type. The parentheses [...] in the syntax description indicate that this part is optional and can be omitted. Thus, one or more variables can be stated within a single definition.

EXAMPLES: char c;
int i, counter;
double x, y, size;

In a program, variables can be defined either within the program's functions or outside of them. This has the following effect:

- a variable defined outside of each function is *global*, i.e. it can be used by all functions
- a variable defined within a function is *local*, i.e. it can be used only in that function.
- Local variables are normally defined immediately after the first brace—for example at the beginning of a function. However, they can be defined wherever a statement is permitted.

This means that variables can be defined immediately before they are used by the program.

3.7. Accessing data members and member functions of class:

Accessing Data Members of Class

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called Accessors and Mutator methods orgetter and setter functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class student
{
public:
int rollno;
string name;
};
int main()
{
student A;
student B;
A.rollno=1;
A.name="Sarthak";
B.rollno=2;
B.name="Sweekar"
cout<<"Name and Roll no of A is :"<<A.name<<A.rollno;
cout<<"Name and Roll no of B is :"<<B.name<<B.rollno;
}
```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```
class Student
{
 private:    // private data member
 int rollno;
 public:     // public accessor and mutator functions
 int getRollno()
 {
  return rollno;
 }
 void setRollno(int i)
 {
  rollno=i;
 }
};
int main()
{
 Student A;
 A.rollono=1;  //Compile time error
 cout<< A.rollno; //Compile time error
 A.setRollno(1);  //Rollno initialized to 1
 cout<< A.getRollno(); //Output will be 1
}
```

So this is how we access and use the private data members of any class using the getter and setter methods. We will discuss this in more details later.

Accessing Protected Data Members

Protected data members, can be accessed directly using dot (.) operator inside the subclass of the current class, for non-subclass we will have to follow the steps same as to access private data member.

Member Functions in Classes

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution: operator along with class name along with function name.

```
Example :
class Cube
{
 public:
 int side;
 int getVolume();      // Declaring function get Volume with no argument and
return type int.
};
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
{
 public:
 int side;
 int getVolume()
 {
  return side*side*side;      //returns volume of cube
 }
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
 public:
 int side;
 int getVolume();
}
int Cube :: getVolume()     // defined outside class definition
{
 return side*side*side;
}
```

The maine function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

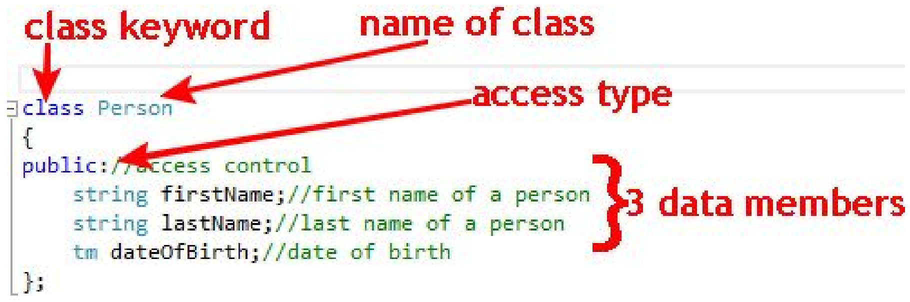
```
int main()
{
 Cube C1;
 C1.side=4;   // setting side value
 cout<< "Volume of cube C1 ="<< C1.getVolume();
}
```

Similarly we can define the getter and setter functions to access private data members, inside or outside the class definition.

3.8. Access Specifiers

C++ offers possibility to control access to class members and functions by using access specifiers. Access specifiers are used to protect data from misuse.

In the Person class, we used only public access specifiers for all data members:



```
class Person
{
public://access control
    string firstName;//first name of a person
    string lastName;//last name of a person
    tm dateOfBirth;//date of birth
};
```

Types of access specifiers in C++

1. public
2. private
3. protected

Public Specifier

Public class members and functions can be used from outside of a class by any function or other classes. You can access public data members or function directly by using dot operator (.) or (arrow operator-> with pointers).

Protected Specifier

Protected class members and functions can be used inside its class. Protected members and functions cannot be accessed from other classes directly. Additionally protected access specifier allows friend functions and classes to access these data members and functions. Protected data members and functions can be used by the class derived from this class. More information about access modifiers and inheritance can be found in C++ Inheritance

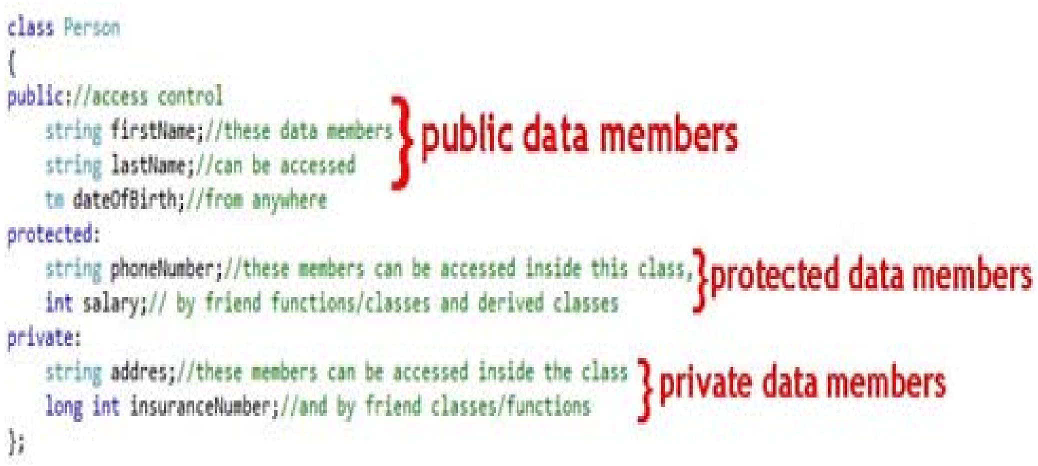
Private Specifier

Private class members and functions can be used only inside of class and by friend functions and classes.

We can modify Person class by adding data members and function with different access specifiers:

```
class Person
{
public://access control
        string firstName;//these data members
        string lastName;//can be accessed
        tm dateOfBirth;//from anywhere
protected:
        string phoneNumber;//these members can be accessed inside this class,
        int salary;// by friend functions/classes and derived classes
private:
        string addres;//these members can be accessed inside the class
        long int insuranceNumber;//and by friend classes/functions
};
```

Access specifier affects all the members and functions until the next access specifier:



```
class Person
{
public://access control
    string firstName;//these data members
    string lastName;//can be accessed
    tm dateOfBirth;//from anywhere
protected:
    string phoneNumber;//these members can be accessed inside this class,
    int salary;// by friend functions/classes and derived classes
private:
    string addres;//these members can be accessed inside the class
    long int insuranceNumber;//and by friend classes/functions
};
```

For classes, default access specifier is **private.** The default access specifier for unions and structs is public.

3.9. Public, private and protected

Class Access Modifiers:

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled public, private, and protected sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

```
class Base {
public:
// public members go here
protected:

// protected members go here
private:
// private members go here
 };
```

The public Members

A public member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example −

```
#include <iostream>
 using namespace std;
 class Line {
   public:
      double length;
      void setLength( double len );
      double getLength( void );
};
 // Member functions definitions
double Line::getLength(void) {
   return length ;
}
 void Line::setLength( double len) {
   length = len;
}
 // Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;

   return 0;
}
```

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

```
Length of line : 6
Length of line : 10
```

The private Members

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

By default all the members of a class would be private, for example in the following class width is a private member, which means until you label a member, it will be assumed a private member −

```
class Box {
double width;
   public:
  double length;
  void setWidth (double wid );
  double getWidth (void );
 };
```

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

```
#include <iostream>

using namespace std;

class Box {
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );

  private:
    double width;
 };

 // Member functions definitions
 double Box::getWidth(void) {
  return width ;
 }

void Box::setWidth( double wid ) {
  width = wid;
 }

 // Main function for the program
 int main() {
  Box box;

  // set box length without member function
  box.length = 10.0; // OK: because length is public
  cout << "Length of box : " << box.length <<endl;

  // set box width without member function
  // box.width = 10.0; // Error: because width is private
  box.setWidth(10.0);  // Use member function to set it.
  cout << "Width of box : " << box.getWidth() <<endl;

  return 0;
  }
```

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

```
Length of box : 10
Width of box : 10
```

The protected Members

A protected member variable or function is very similar to a private member but it provides one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class SmallBox from a parent class Box.

Following example is similar to above example and here width member will be accessible by any member function of its derived class SmallBox.

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

class Box {
  protected:
    double width;
};
```

```
class SmallBox:Box { // SmallBox is the derived class.
  public:
    void setSmallWidth( double wid );
    double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void) {
  return width ;
}

void SmallBox::setSmallWidth( double wid ) {
  width = wid;
}

// Main function for the program
int main() {
  SmallBox box;

  // set box width using member function
  box.setSmallWidth(5.0);
  cout << "Width of box : "<< box.getSmallWidth() << endl;

  return 0;
}
```

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

```
Width of box : 5
```

3.10. Initializing class objects

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

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

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

The keyword public determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

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

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

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear –

```
#include <iostream>
using namespace std;
class Box {
  public:
     double length;   // Length of a box
     double breadth;  // Breadth of a box
     double height;   // Height of a box
};

int main() {
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;

  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;

  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below −

Sr.No	Concept & Description
1	Class Member Functions A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
2	Class Access Modifiers A class member can be defined as public, private or protected. By default members would be assumed as private.
3	Constructor & Destructor A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.
4	Copy Constructor

	The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
5	Friend Functions A friend function is permitted full access to private and protected members of a class.
6	Inline Functions With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.
7	this Pointer Every object has a special pointer this which points to the object itself.
8	Pointer to C++ Classes A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.
9	Static Members of a Class Both data members and function members of a class can be declared as static.

3.11. Constructors and Destructor:

Class Constructor:

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor −

```
#include <iostream>
 using namespace std;
 class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor
   private:
      double length;
};
 // Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}
// Main function for the program
int main() {
   Line line;
    // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
    return 0;
}
```

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

Object is being created

```
Length of line : 6
```

Parameterized Constructor

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example −

```
#include <iostream>
 using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};
 // Member functions definitions including constructor
Line::Line( double len) {
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}
```

```
// Main function for the program
int main() {
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization Lists to Initialize Fields

In case of parameterized constructor, you can use following syntax to initialize the fields −

```
Line::Line( double len): length(len) {
   cout << "Object is being created, length = " << len << endl;
}
```

Above syntax is equal to the following syntax −

```
Line::Line( double len) {
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows −

```
C::C( double a, double b, double c): X(a), Y(b), Z(c) {
   ....
}
```

Class Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor −

```
#include <iostream>
 using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}

// Main function for the program
int main() {
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

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

```
Object is being created
Length of line : 6
Object is being deleted
```

3.12. Default copy constructor

Copy Constructor:

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to −

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here −

```
classname (const classname &obj) {
   // body of constructor
}
```

Here, **obj** is a reference to an object that is being used to initialize another object.

```
#include <iostream>
using namespace std;
class Line {
   public:
      int getLength( void );
      Line( int len );             // simple constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                     // destructor

   private:
      int *ptr;
};
```

```
// Member functions definitions including constructor
Line::Line(int len) {
   cout << "Normal constructor allocating ptr" << endl;

   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}

Line::Line(const Line &obj) {
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void) {
   cout << "Freeing memory!" << endl;
   delete ptr;
}

int Line::getLength( void ) {
   return *ptr;
}

void display(Line obj) {
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main() {
   Line line(10);

   display(line);

   return 0;
}
```

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

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Let us see the same example but with a small change to create another object using existing object of the same type −

```
#include <iostream>

using namespace std;

class Line {
  public:
    int getLength( void );
    Line( int len );            // simple constructor
    Line( const Line &obj);  // copy constructor
    ~Line();                    // destructor

  private:
    int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len) {
  cout << "Normal constructor allocating ptr" << endl;

  // allocate memory for the pointer;
  ptr = new int;
  *ptr = len;
}
```

```
Line::Line(const Line &obj) {
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void) {
   cout << "Freeing memory!" << endl;
   delete ptr;
}

int Line::getLength( void ) {
   return *ptr;
}

void display(Line obj) {
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main() {

   Line line1(10);

   Line line2 = line1; // This also calls copy constructor

   display(line1);
   display(line2);

   return 0;
}
```

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

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
Freeing memory!
```

3.14 Static data member in class

We can define class member static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator: to identify which class it belongs to.

Let us try the following example to understand the concept of static data members −

```
#include <iostream>
 using namespace std;
  class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
```

```
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;

      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
 Box Box1(3.3, 1.2, 1.5);    // Declare box1
 Box Box2(8.5, 6.0, 2.0);    // Declare box2

 // Print total number of objects.
 cout << "Total objects: " << Box::objectCount << endl;

 return 0;
}
```

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

```
Constructor called.

Constructor called.

Total objects: 2
```

3.14 Static Function Members of Class:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator :

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members

```
#include <iostream>

using namespace std;

class Box {
   public:
      static int objectCount;

      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
```

```
        breadth = b;
        height = h;

        // Increase every time object is created
        objectCount++;
      }
      double Volume() {
        return length * breadth * height;
      }
      static int getCount() {
        return objectCount;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;
   return 0;
```

```
}
```

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

```
Inital Stage Count: 0

Constructor called.

Constructor called.

Final Stage Count: 2
```

3.15. Inline Function

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them.

For an inline function, declaration and definition must be done together. For example,

```
inline void fun(int a)
{
  return a++;
}
```

Some Important points about Inline Functions

1. We must keep inline functions small, small inline functions have better efficiency.
2. Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to code bloat, and might affect the speed too.
3. Hence, it is advised to define large functions outside the class definition using scope resolution :: operator, because if we define such functions inside class definition, then they become inline automatically.

4. Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

Access Functions

We have already studied this in topic Accessing Private Data variables inside class. We use access functions, which are inline to do so.

```
class Auto
{
 int i;
 public:
 int getdata()
  {
   return i;
  }
 void setdata(int x)
 {
  i=x;
 }
};
```

Here getdata() and setdata() are inline functions, and are made to access the private data members of the class Auto. getdata(), in this case is called Accessor function and setdata() is a Mutator function.

There can be overlaoded Accessor and Mutator functions too. We will study overloading functions in next topic.

Limitations of Inline Functions

1. Large Inline functions cause Cache misses and affect performance negatively.
2. Compilation overhead of copying the function body everywhere in the code on compilation, which is negligible for small programs, but it makes a difference in large code bases.

3. Also, if we require address of the function in program, compiler cannot perform inlining on such functions. Because for providing address to a function, compiler will have to allocate storage to it. But inline functions doesn't get storage, they are kept in Symbol table.

3.16. Data encapsulation and its example

All C++ programs are composed of the following two fundamental elements−

- **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
- **Program data** − The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private. For example −

```
class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members addNum and getTotal are the interfaces to the outside world and a user needs to know them to use the class. The private member total is something that is hidden from the outside world, but is needed for the class to operate properly.

3.17 Passing parameters to a constructor function

It is possible to pass arguments to constructor functions. 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. For example,

here is a simple class that includes a parameterized constructor:

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
myclass(int i, int j) {a=i; b=j;}
void show() {cout << a << " " << b;}
};
int main()
{
myclass ob(3, 5);
ob.show();
return 0;
}
```

Notice that in the definition of myclass(), the parameters i and j are used to give initial
values to a and b.
The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor function. Specifically, this statement myclass ob(3, 4);
causes an object called ob to be created and passes the arguments 3 and 4 to the i and j
parameters of myclass(). You may also pass arguments using this type of declaration statement:
myclass ob = myclass(3, 4);

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors.
Here is another example that uses a parameterized constructor function. It creates a class that stores information about library books.

```
#include <iostream>
#include <cstring>
using namespace std;
const int IN = 1;
const int CHECKED_OUT = 0;
class book {
char author[40];
char title[40];
int status;
public:
book(char *n, char *t, int s);
int get_status() {return status;}
void set_status(int s) {status = s;}
void show();
};
book::book(char *n, char *t, int s)
{
strcpy(author, n);
strcpy(title, t);
status = s;
}
void book::show()
{
cout << title << " by " << author;
cout << " is ";
if(status==IN) cout << "in.\n";
else cout << "out.\n";
}
int main()
{
book b1("Twain", "Tom Sawyer", IN);
book b2("Melville", "Moby Dick", CHECKED_OUT);
b1.show();
b2.show();
return 0;
}
```

Parameterized constructor functions are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. Also, notice that the short get_status() and set_status() functions are defined in line, within the book class. This is a common practice when writing C++ programs.

Constructors with One Parameter:

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor. For example, consider the following short program.

```
#include <iostream>
using namespace std;
class X {
int a;
public:
X(int j) { a = j; }
int geta() { return a; }
};
int main()
{
X ob = 99; // passes 99 to j
cout << ob.geta(); // outputs 99
return 0;
}
```

Here, the constructor for X takes one parameter. Pay special attention to how ob is declared in main(). In this form of initialization, 99 is automatically passed to the J parameter in the X() constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

```
X ob = X(99);
```

In general, any time you have a constructor that requires only one argument, you can use either ob(i) or ob = i to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Remember that the alternative shown here applies only to constructors that have exactly one parameter.

3.18 Difference between constructors and member function:

Here is the difference between constructor and member function in C++ programming. Constructor name must be same as class name but functions cannot have same name as class name.

```
C++ Code Example :
class Car{
        int count;

public:
        //This constructor has same name as class name
        Car(){
        }

        //function cannot have same name as class
        void CarAvailable(){
        }
};
```

Constructor does not have return type whereas functions must have.

```
Example :
class Car{
        int count;

public:
        //Constructor :- No return type
        Car(){
        }

        //@return type int, in C++ function must have return type
        // it can also be void type, means return nothing, but must mention its
type.
        int CarAvailable(){

                return count;
        }
        void CarSold(){}
};
```

Member function can be virtual, but, there is no concept of virtual-constructor in C++. (NOTE: virtual destructor to maintain destructor call order in inheritance is available in C++ language)

```
Example :
class Car{
        int count;

public:
        //Constructor :- Can never be VIRTUAL,No provision.
        Car(){
                //
        }

        //Function can be virtual, so that it can be overriden in derived classes.

        virtual void CarAvailable(){
        }
};
```

Constructors are invoked at the time of object creation automatically and cannot be called explicitly but functions are called explicitly using class objects.

C++ Code Example :

```
class Car{
 public:
        Car(){
cout << "Car's Constructor\n";
}

void CarAvailable(){
                cout << "Car's Function\n";
}
};

 int main()
{
        //Constructor will be invoked automatically
        //during object creation.
Car obj;
//Functin can be called using class object, no automatic
obj.CarAvailable();
return 0;
}
```

SUMMARY

- In OOP, we can define class name to represent its individual objects. Defining a class in programming language means creating user defined data type and it behaves like the built-in data types. Once a class has been defined, we can create any number of objects belonging to that class.
- An object is an instance of a class. A class is a description of a group of objects with common properties (attributes or characteristics or variables), behavior (operations or methods) and relationships.
- The members of structure are usually processed individually, as separate entity. Therefore, we must be able to access the individual structure members. A structure member can be accessed by using period or dot (i.e. ".") operator.
- In OOP, generally date are made private and functions are made public.
- A variable must be defined before you can use it in a program. When you define a variable the type is specified and an appropriate amount of memory reserved.
- Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.
- Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.
- C++ offers possibility to control access to class members and functions by using access specifiers. Access specifiers are used to protect data from misuse.
- Public class members and functions can be used from outside of a class by any function or other classes.
- Protected class members and functions can be used inside its class. Protected members and functions cannot be accessed from other classes directly.
- Private class members and functions can be used only inside of class and by friend functions and classes.

- The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
- A class member can be defined as public, private or protected. By default members would be assumed as private.
- The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- A friend function is permitted full access to private and protected members of a class.
- With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.
- Every object has a special pointer this which points to the object itself.
- A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.
- Both data members and function members of a class can be declared as static.
- A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
- A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object
- A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- We can define class member static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator :

- Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them.
- Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Self Evaluation

1. **Write very short answer of the following question.**

a) What is a class?
b) What is an object?
c) What is a structure member?
d) What is a variable?
e) What is accessing a data member?
f) What are access specifiers?
g) What is a class member?
h) What is class constructor?
i) What is a structure member?
j) What is copy constructor?
k) What is friend function?
l) What is a destructor?
m) What is a default constructor?
n) What is called static member function?
o) What is the inline function?
p) What is a data encapsulation?

2. **Write short answer of the following question.**

a) What are accessing data members of class?
b) What do you mean by member function in class? Explain with example.

c) Explain class constructors with examples.
d) Explain class destructors with examples.
e) Define copy constructor. Where is it used?
f) Explain static data member with examples.
g) What are the different between constructors and member function?
h) Write a program to add two variables and disply the sum using class and object.

3. Write long answer of the following question.

a) What are the difference between classes and structures?
b) What are the types of access specifiers ? Explain in detail.
c) What is inline function? Write its importance.
d) Write about data encapsulation with its example.
e) Explain copy constructor with a program.

UNIT 4
Polymorphism

Learning Outcomes:-

After completion of this unit you will be able to

- To explain/describe polymorphism.
- To explain/describe function overriding.
- To explain/describe virtual function.
- To describe run time polymorphism.
- To explain/describe static binding and dynamic binding.
- To explain/describe Abstract class and pure virtual function.

4.1 Introduction to Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

```
#include <iostream>
using namespace std;
 class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
```

```
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }



    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);

  // store the address of Rectangle
```

```
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

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

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is calledstatic resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this −

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
```

```
            height = b;
         }
         virtual int area() {
            cout << "Parent class area :" <<endl;
            return 0;
         }
      };
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

4.2 Function Overriding

If we inherit definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding

Requirements for Overriding

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.

Function Overriding Example

To override a function you must have the same signature in child class. By signature I mean the data type and sequence of parameters. Here we don't have any parameter in the parent function so we didn't use any parameter in the child function.

```
#include <iostream>
using namespace std;
class BaseClass {
public:
  void disp(){
    cout<<"Function of Parent Class";
  }
};
class DerivedClass: public BaseClass{
public:
  void disp() {
    cout<<"Function of Child Class";
  }
};
int main() {
  DerivedClass obj = DerivedClass();
  obj.disp();
  return 0;
}
Output: Function of ChildClass
```

Function of Child Class

Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

4.3. Virtual Function

A virtual function is a function in a base class that is declared using the keywordvirtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

```
class Shape {
  protected:
    int width, height;

  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

4.4. Runtime polymorphism

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the compiler.	In Run time Polymorphism, call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operatoroverloading.	It is achieved by virtual functions and pointers.
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

4.5 Static Binding and Dynamic Binding

Definitions of Static Binding

When compiler acknowledges all the information required to call a function or all the values of the variables during compile time, it is called “static binding”. As all the required information are known before runtime, it increases the program efficiency and it also enhances the speed of execution of a program.

Static Binding makes a program very efficient, but it declines the program flexibility, as ‘values of variable’ and ‘function calling’ are predefined in the program. Static binding is implemented in a program at the time of coding.

Overloading a function or an operator are the examples of compile time polymorphism i.e. static binding.

Implementation of static binding in C++ with example of overloading

```
class overload{
int a, b;
public:
int load(int x){ // first load() function.
a=x;
return a;
}
int load(int x, int y){ //second load() function.
a=x;
b=y;
return a*b;
}
};
int main(){
overload O1;
O1.load(20); //This statement binds the calling of function to 'first' load()
function.
O1.load(20,40); //This statement binds the calling of function 'second' load()
function.
}
```

Now, here the statement ‘O1.loads(20)’ clearly binds the function calling to first ‘load(int x)’ function as it is the only function which accepts single integer argument. The statement O1.loads(20,40) binds the function calling to second ‘load(int x, int y)’ function as it is the only function which accepts two integer arguments. Hence, no time will be wasted in deciding which function to invoke; this will make program execution efficient and fast.

Definition of Dynamic Binding

Calling a function or assigning a value to a variable, at run-time is called "Dynamic Binding". Dynamic binding can be associated with run time 'polymorphism' and 'inheritance' in OOP. Dynamic binding makes the execution of program flexible as it can be decided, what value should be assigned to the variable and which function should be called, at the time of program execution. But as this information is provided at run time it makes the execution slower as compared to static binding.

Implementation of dynamic binding using 'virtual functions' in C++.

```
class base{
public:
virtual void funct(){ // Virtual function.
cout<<"This is a base class's funct()";
}
};
class derived1 : public base{
public:
void funct(){ //overridden virtual function.
cout<<"This is a derived1 class's funct()";
}
};
class derived2 : public base{
public:
void funct(){ //overridden virtual function.
cout<<"This is a derived2 class's funct()";
}
};
int main()
{
```

```
base *p, b;
derived1 d1;
derived2 d2;
*p=&b;
p->funct(); //The above statement decides which class's function is to be
invoked.

*p=&d1; // Vlaue of the pointer changes.
p->funct(); //The above statement decides which class's function is to be
invoked.

*p=&d2; // Again vlaue of the pointer changes.
p->funct(); //The above statement decides which class's function is to be
invoked.
return 0;
}
```

Here the value of the pointer changes as the program is in execution and the value of the pointer decides which class’s function will be invoked. So here, the information is provided at run time, it takes the time to bind the data which slow downs the execution.

Key Differences Between Static and Dynamic Binding.

1. Events that occur at compile time like, a function code is associated with a function call or assignment of value to a variable, are called static/early binding, and when these tasks are accomplished during runtime they are called dynamic/late binding.
2. ‘Efficiency’ increases in static binding, as all the data is gathered before the execution. But in dynamic binding, the data is acquired at runtime so we can decide what value to assign a variable and which function to invoke at runtime this make execution ‘flexible’.
3. ‘Static binding’ make execution of a program ‘faster’ as all the data needed to execute a program is known before execution. In ‘dynamic binding’ data needed to execute a program is known to the compiler at the time of execution

which takes the time to bind values to identifiers hence, it makes program execution slower.

4. Static binding is also called early binding because the function code is associated with function call during compile time, which is earlier than dynamic binding in which function code is associated with function call during runtime hence it is also called late binding.

Conclusion:

However, we conclude that when we have the prior knowledge of the values of variable and function calling, we apply static binding whereas, in dynamic binding, we provide all the information at the time of execution.

4.6 Abstract class and pure virtual function

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

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

Example of Abstract Class

```
class Base          //Abstract base class
{
 public:
 virtual void show() = 0;           //Pure Virtual Function
};

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base obj;       //Compile Time Error
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output :

```
Implementation of Virtual Function in Derived class
```

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE (studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an error message whenever you try to do so.

Pure Virtual definitions

Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.

Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.c

```
 //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

void Base :: show()        //Pure Virtual definition
{
 cout << "Pure Virtual definition\n";
}

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
Output :
Implementation of Virtual Function in Derived class
```

SUMMARY

- The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding
- A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
- When compiler acknowledges all the information required to call a function or all the values of the variables during compile time, it is called “static binding”.
- Calling a function or assigning a value to a variable, at run-time is called “Dynamic Binding”.
- Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes

Self Evaluation

1. Write very short answer of the following question.

a. What is polymorphism?
b. What is Function Overriding?
c. What is virtual function?
d. What is static binding?
e. What is Dynamic Binding?
f. What is Abstract Class?

2. Write short answer of the following question.

a. What is polymorphism? Explain with example.
b. What are the requirements for overriding?
c. Explain static binding with example.
d. What are the key differences between static binding and dynamic binding?
e. What are the characteristics of abstract class?

3. Write long answer of the following question.

a. What are the different between Compile time Polymorphism and Run time Polymorphism?
b. Explain polymorphism with a program.
c. What is virtual function? Explain virtual function with a program.

UNIT-5

Operator Overloading

Learning Outcomes

After completion of this unit you will be able to

- To explain/describe unary operators overloading.
- To explain/describe operator argument.
- To explain/describe operator return values.
- To describe postfix notation.
- To explain overloading binary operators
- To explain/describe arithmetic operators and concatenating strings.

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below −

```
#include <iostream>
using namespace std;
class Box {
  public:
    double getVolume(void) {
      return length * breadth * height;
    }
    void setLength( double len ) {
      length = len;
    }
    void setBreadth( double bre ) {
      breadth = bre;
    }
    void setHeight( double hei ) {
      height = hei;
    }

    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};

// Main function for the program
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   Box Box3;              // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here
```

```
   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
   }
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadable Operators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	New	new []	delete	delete []

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

Sr.No	Operators & Example
1	Unary Operators Overloading
2	Binary Operators Overloading
3	Relational Operators Overloading

4	Input/Output Operators Overloading
5	++ and -- Operators Overloading
6	Assignment Operators Overloading
7	Function call () Operator Overloading
8	Subscripting [] Operator Overloading
9	Class Member Access Operator -> Overloading

5.1 Overloading unary operators:

The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
        feet = 0;
        inches = 0;
      }
      Distance(int f, int i) {
        feet = f;
        inches = i;
      }

      // method to display distance
      void displayDistance() {
        cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
        feet = -feet;
        inches = -inches;
        return Distance(feet, inches);
      }
};

int main() {
   Distance D1(11, 10), D2(-5, 11);

   -D1;                  // apply negation
   D1.displayDistance();    // display D1

   -D2;                  // apply negation
   D2.displayDistance();    // display D2
  return 0;
}
```

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

```
F: -11 I:-10
F: 5 I:-11
```

Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!).

5.1.1. Operator Argument:

There is restrictions on overloaded operators. We cannot change the precedence, grouping, or number of operands of the standard C++ operators. An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.

5.1.2. Operator Return Values:

If we have to use overloaded operator function for return a value as:

```
Obj 2 = obj1 ++; // returned object of obj++ is assigned to obj2.
Example:
#include<iostream.h>
Class sample
{
        private:
                int x;
        public:
                sample () { x=10;}            // Constructor
                int getvalue (){ return x;}
                sample operator ++()
                {
                        x++;
                        sample temp;          // temporary object
                        temp.x = x;
                        return temp;
                }
  };
void main()
{
        sample obj1, obj2;
        cout<<end1<<intial obj1="<<obj1..getvalue();
        cout<<end1<<intial obj2="<<obj2..getvalue();
        obj1++; obj2++;
        obj2 =obj1++;
        cout<<end1<<final obj1="<<obj1.getvalue();
        cout<<end1<<final obj2="<<obj2.getvalue();
}
//output
intial obj1=10
intial obj2=10
final obj1=13
final obj2=13
```

5.1.3 Postfix Notation:

Postfix notation is also known as Reverse Polish Notation (RPN) in which every operator follows all of its operands. This notation is parenthesis free. For e.g, (A + B) is expressed as AB+ in postfix notation.

```
// postfix operation
Sample operator ++ (int)
{
        Return(val++)       // object is created with val++
        // i.e old value and value is returned.
We can give increment role to – operator and decrement role to ++ operator
defining operator function as
sample operator ++()
{
x--;
return sample(x);
}
// decrements
sample operator –()
{
X++;
Return sample(x);        //increments
}
```

5.2. Overloading binary operators:

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box

   public:

   double getVolume(void) {
      return length * breadth * height;
   }

   void setLength( double len ) {
      length = len;
   }
    void setBreadth( double bre ) {
      breadth = bre;
   }

   void setHeight( double hei ) {
      height = hei;
   }

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
 };

 // Main function for the program
 int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
```

```
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

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

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

5.2.1 Arithmetic operators:

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

```
#include <iostream>
using namespace std;
 main() {
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   cout << "Line 1 - Value of c is :" << c << endl ;

   c = a - b;
   cout << "Line 2 - Value of c is  :" << c << endl
   ;
   c = a * b;
   cout << "Line 3 - Value of c is :" << c << endl ;

   c = a / b;
   cout << "Line 4 - Value of c is  :" << c << endl ;

   c = a % b;
   cout << "Line 5 - Value of c is  :" << c << endl ;
   c = a++;
   cout << "Line 6 - Value of c is :" << c << endl ;

   c = a--;
   cout << "Line 7 - Value of c is  :" << c << endl ;

   return 0;
   }
```

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

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

5.2.2 Concatenating strings:

In C, + operator cannot concatenate two strings. In C++, it is possible to concatenate strings using overloaded + operator. Following example shows the concatenation of strings with operator +.

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
Class string
{
        private:
                char str[100];
        public:
                string() {strcpy(str,"");}        // initialization with constructor
                string(char*s)
                {
                        strcpy(str,s);        //one argument constructor
                void show()
                {
                        cout<<str;  // puts the string
                string operator+(string ss)
                {
                        String temp;
                        If ((strlen(str) + strlen(ss.str))<100)
                        {
                                // copies first string to temp
                                strcpy(temp.str,str);
                                //adds argument string
                                strcat(temp.str,ss.str);
                        }
                        Return temp;
                }
        };      //end class
        void main()
        {
                string s1("FirstString");
                string s2("SecondString");
                string s3;
                cout<<"FirstString:"; s1.show();
                cout<<"\n SecondString:"; s2.show();
                s3 = s1 +s2;        // adds s2 and s1 and assigns to s3
                cout<<"\n String after concatenating two strings:"<<end1;
                s3.show();
        }
        // output
        First string: FirstString
        Second String: SecondString
        String after concatenating two strings:
        First string: FirstStringSecond StringSecondString
```

SUMMARY

- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined.
- The unary operators operate on a single operand and following are the examples of Unary operators −
  - The increment (++) and decrement (--) operators.
  - The unary minus (-) operator.
  - The logical not (!) operator.
- Postfix notation is also known as Reverse Polish Notation (RPN) in which every operator follows all of its operands. This notation is parenthesis free. For e.g, (A + B) is expressed as AB+ in postfix notation.
- The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.
- . In C++, it is possible to concatenate strings using overloaded + operator.

Self Evaluation

1. **Write very short answer of the following question.**
   a) What is a operator overloading?
   b) What is a postfix notation?
   c) What do you mean by function overloading?
   d) What is operator function?
2. **Write short answer of the following question.**
   a) What is a unary operator? Explain with examples.
   b) What is arithmetic operator? Explain with examples.
   c) What are concatenate strings? Explain with examples.
   d) In the context to operator overloading, what do you understand by the term "nameless temporary object"?
   e) What is a postfix notation? Explain with example.
3. **Write long answer of the following question.**
   a) Explain overloading binary operators with examples.
   b) Write a program to overload "<" operator to compare two objects.
   c) What is string concatenation? Write a program to show string concatenation using overloaded +operator.

UNIT 6
Inheritance

Learning Outcomes

After completion of this unit you will be able to

- To explain/describe inheritance and its basic concepts.
- To explain/describe basic class and derive class.
- To explain/describe accessing base class members.
- To describe public, private and protected inheritance in C++.
- To explain abstract base class.
- To explain/describe forms of inheritance.

Inheritance is like a child inheriting the features of its parents. It is a technique of organizing information in a hierarchical (tree) form.

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the baseclass, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

6.1. Introduction to inheritance

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the Parent or Base or Super class. And, the class which inherits properties of other class is called Child or Derived or Sub class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

Purpose of Inheritance

1) Code Reusability
2) Method Overriding (Hence, Runtime Polymorphism.)
3) Use of Virtual Keyword

6.2. Basic Concepts:

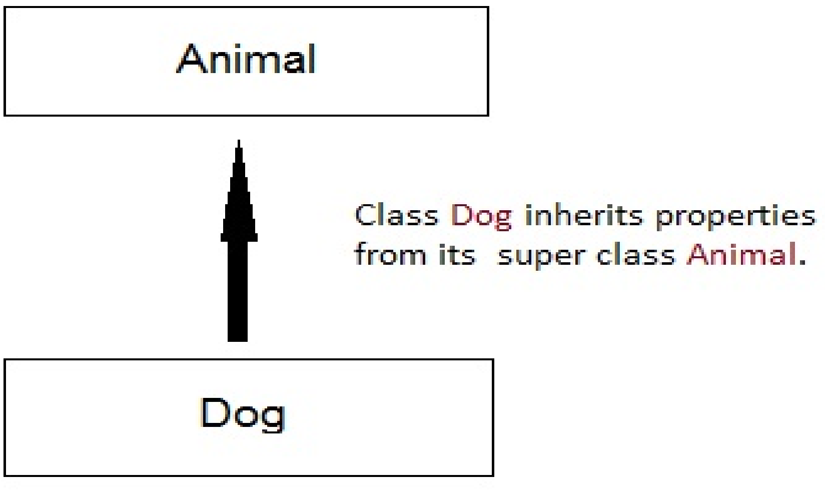
Basic Syntax of Inheritance

class Subclass_name: access_mode Superclass_name

While defining a subclass like this, the super class must be already defined or at least declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, private or protected.

Example of Inheritance



```
class Animal
{ public:
  int legs = 4;
};
class Dog : public Animal
{ public:
  int tail = 1;
};

int main()
{
 Dog d;
 cout << d.legs;
 cout << d.tail;
}
Output :
4 1
```

6.3. Base class and Derived class:

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

```
class derived-class: access-specifier base-class
```

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

Consider a base class Shape and its derived class Rectangle as follows −

```
#include <iostream>

using namespace std;

// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;

  Rect.setWidth(5);
```

```
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

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

```
Total area: 35
```

6.4 Accessing base class members:

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	Yes
Derived classes	yes	yes	No
Outside classes	yes	no	No

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

6.5 Public, Private and Protected Inheritance in C++:

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied −

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

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

Private Inheritance − When deriving from a private base class, public and protected members of the base class become privatemembers of the derived class.

Table showing all the Visibility Modes

	Derived Class	Derived Class	Derived Class
Base class	Public Mode	Private Mode	Protected Mode
Private	Not Inherited	Not Inherited	Not Inherited

	Derived Class	Derived Class	Derived Class
Base class	Public Mode	Private Mode	Protected Mode
Protected	Protected	Private	Protected
Public	Public	Private	Protected

6.6 Abstract Base Class:

The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes. An abstract class is one that has no instances and is not designed to create objects. It is only designed to be inherited.

```
class one
stack2 s;
s.push (11);
s.push (22);
s.push (33);
cout<<"\nNumber Popped"<<s.pop();
cout<<"\nNumber Popped"<<s.pop();
cout<<"\nNumber Popped"<<s.pop();
cout<<"\nNumber Popped"<<s.pop();
getch();
```

class stack2 is derived from class stack. Object of stack2 behave in exactly the same way as those of stack, except if attempt is made to push too many items on the stack, or to pop an item from an empty stack.

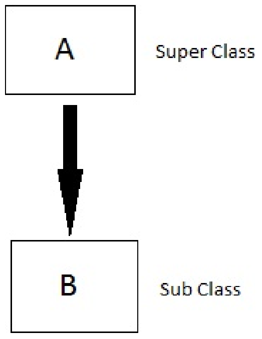
6.7 Forms/Types of inheritance:

In C++, we have 5 different types of Inheritance. Namely,

1) Single Inheritance
2) Multiple Inheritance
3) Hierarchical Inheritance
4) Multilevel Inheritance
5) Hybrid Inheritance (also known as Virtual Inheritance)

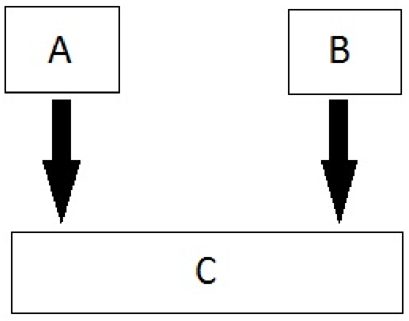
1. Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



2. Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



C++ class can inherit members from more than one class and here is the extended syntax −

```
class derived-class: access baseA, access baseB....
```

Where access is one of public, protected, or private and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

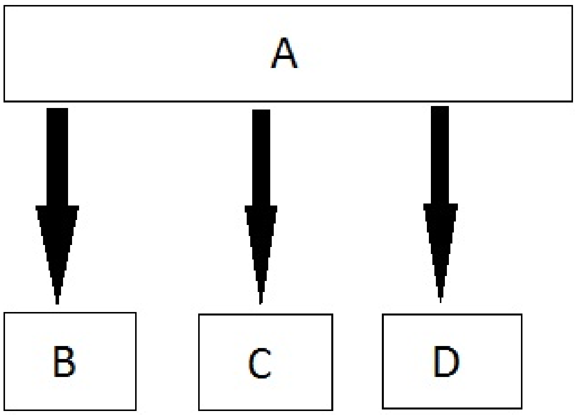
```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;

  return 0;
}
```

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

```
Total area: 35
Total paint cost: $2450
```

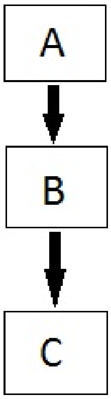
3. Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



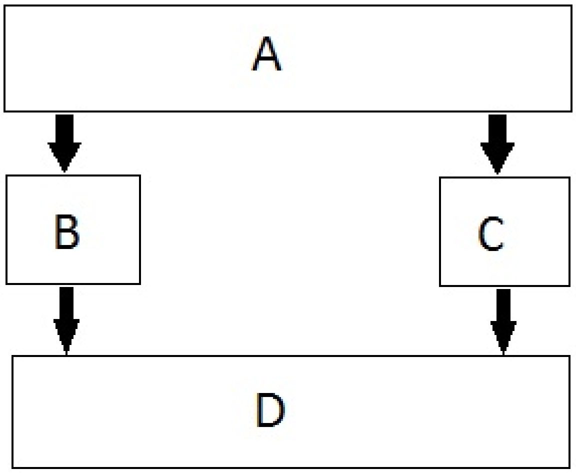
4. Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



5. Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multi level Inheritance.



SUMMARY

- Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.
- A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

- A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
- When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance
- When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class.
- When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived
- When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.
- The objects created are often the instance of a derived class but not the base class. The base class is just the foundation for building new classes and hence such classes are called abstract base classes or abstract classes.
- Single Inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.

- Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.
- Hierarchical Inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.
- Multilevel Inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other
- **Hybrid Inheritance** is combination of Hierarchical and Mutilevel Inheritance.

Self Evaluation

1. Write very short answer of the following question.

1) What is the basic syntax of inheritance?
2) What is a derived class?
3) What is an abstract base class?
4) What is multilevel inheritance?

2. Write short answer of the following question.

1) What is a single inheritance? Write the syntax of it.
2) What do you mean by base class and derived class.
3) What do you mean by multiple inheritance? Explain with example.
4) What do you mean by multilevel inheritance? Explain with example.

3. Write long answer of the following question.

1) What is inheritance? Explain different types of inheritance.
2) What is ambiguity in multiple inheritances? How do you resolve it? Explain with examples.
3) Explain about public, private and protected inheritance.

References:

https://chipkidz.wordpress.com/2009/08/07/procedural-programming/

https://www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/8043

https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

https://cs-fundamentals.com/tech-interview/c/difference-between-c-and-cpp.php

https://www.tutorialspoint.com/cplusplus/cpp_basic_input_output.htm

https://codescracker.com/cpp/cpp-error-handling.htm

https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

https://www.tutorialspoint.com/cplusplus/cpp_polymorphism.htm

https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

https://www.studytonight.com/cpp/overview-of-inheritance.php

C++ HANDBOOK by SSI

A Text book of OBJECT ORIENTED PROGRAMMING in C++ by Ram Datta Bhatt