SCHEME

&

SYLLABUS

Scheme of Even Semester

Department of Applied Sciences (UIE)						
Name of the Degree – B.E.						
Name of the Degree-B.E.(IT Segment)CSE,IT,ECE,EE						
Semester –II						
Subject		**Subject Code**	**L**	**T**	**P**	**Cr.**
Probability and Statistics	BS	SMT-172	3	2	0	5
Object Oriented Programming using C++	ES	CST-157	0	2	0	2
Biology For Engineers	BS	SZT-172	3	0	0	3
Professional Communication Skills	HSM	PCT-154	2	0	0	2
Digital Electronics	EC	ECT-155	3	1	0	4
Innovation and Inventions in Computer Science and Engineering	ES	CST-156	2	0	0	2
Object Oriented Programming Using C++ Lab	ES	CSP-157	0	0	4	2
Professional Communication Skills Lab	HSM	PCP-158	0	0	2	1
Digital Electronics Lab	EC	ECP-156	0	0	2	1
Workshop Practice	ES	MEP-160	0	1	2	2
Total			13	6	10	23
Life Skills and Mentoring	MNG	UCY-196	0	1	0	1

Syllabus

Subject Code: CST-157	Object Oriented Programming using C++	L	T	P	C
	Total Contact Hours :30 Hours	0	2	0	2
	Common to all Branches of First Year				

Marks-100	
Internal-40	External-60

Course Objectives

- To understand the principals of Programming
- To apply different programming techniques for modeling real world problems.

Unit	Course Outcomes
I-III	Understanding Features of C++, Various programs based on classes and objects.
	Programs utilizing Inheritance and Polymorphism
	To provide in-depth knowledge of Pointers and File handling

Unit-I **[10L]**

Fundamentals of C++: Features of object-oriented programming, Difference between object oriented and procedure oriented programming, Difference between structure and class, Data types. Input and output streams (cin, cout), introduction to namespace. [3]

Classes and Objects: Specifying a class, creating objects, accessing class members, defining a member function inside and outside class, access specifiers, inline function, static data members & member functions. Objects as function arguments, friend function, returning objects to functions. [4]

Constructors and Destructors: Need for constructors, types of constructors: default, parameterized, copy constructor, order of execution of constructors, destructors and their need. [3]

Unit-II **[10L]**

Inheritance: Defining derived class, modes of inheritance, types of inheritance, ambiguity in inheritance, virtual base class, Function overriding, Member Classes: Nesting of Classes. [4]

Polymorphism: Introduction & types of polymorphism, Function overloading, operator overloading, rules for overloading operators, overloading of unary & binary operators, Constructor Overloading. [3]

Exception Handling: Try, Throw, Catch, Throwing an Exception, Catching an Exception. [2]

Unit-III [10L]

Pointers, Virtual Functions: Declaring & initializing pointers, pointer to objects, this pointer, pointer to derived classes, static and dynamic binding. [4]

Dynamic memory allocation: Dynamic memory allocation using new and delete operator. [2]

Files: Introduction to File streams, Hierarchy of file stream classes, File operations, File I/O, File opening Modes, Reading/Writing of files, Random-access to files. [4]

Text books:

1. E Balagurusamy., "Object Oriented Programming in C++", Tata McGraw-Hill.

2. Robert Lafore, "Object Oriented Programming in C++", Waite Group, December 1998.

Reference Books:

1. **Herbert Schildt , "C++- The Complete Reference", Tata McGraw-Hill 2003, New Delhi.**
2. **Bjarne Stroustrup: "The C++ Programming Language" (4th Edition). Addison-Wesley. May 2013.**
3. Ravichandran , "Programming with C++",Tata McGraw-Hill Education, 2001.
4. Joyce M. Farrell," Object Oriented Programming Using C++",Cengage Learning, January 1998.
5. Programming Languages: Design and Implementation (4th Edition), by Terrence W. Pratt, Marvin V. Zelkowitz, Pearson, 2000.
6. Programming Language Pragmatics, Third Edition, by Michael L. Scott, Morgan Kaufmann, 2009.

Course Code- CST-157	**Object Oriented Programming using C++**									
Department Teaching the Subject	**University Institute of Engineering, Academic Unit-I & II**									
Program Outcome	A	b	c	d	e	f	g	H	i	J
Mapping of Course outcome with Program outcome	I,II, III									
Category	BS	ES		PD	PC		PE	OE	Project/Training	
					x					
Approval	Date of meeting of the Board of Studies									

Instructions for the paper-setter

Please go through these instructions thoroughly and follow the same pattern while setting the paper as the students have been prepared according to this format.

Maximum Marks = 60 Time: 3 Hrs

The syllabus has been divided into three equal units. The paper setter is required to set ten questions in all, three questions from each unit and a compulsory question consisting of five sub parts and based on the whole syllabus. The candidate will be required to attempt six questions including the compulsory question number no 1 and not more than two questions from each unit.

Abstract

This monograph is presented with a strong intent to teach or to give an experience on program development in C++ to an undergraduate student. This monograph presents a collection of C++ concepts that are usually attempted by a student who is a beginner or in the lower semesters of an undergraduate programme. It is presumed that the reader has some previous acquaintance with programming in C language i.e., reader needs to know what a variable is and what a function is -- but doesn't need much experience. This monograph presents a compendium of workable C++ concepts. It is essential to follow the order of the units rigorously; if the reader is a beginner to C++. The example and programs covered is self explanatory. Further, the material presented in this monograph is expected to serve as guidelines for writing programs on C++ or for writing programs on engineering domain specific problems.

UNIT-1

1.1 Object Oriented programming

Object Oriented Programming (OOP) is a paradigm in which real-world objects are each seen as separate entities having their own particular state which is adjusted just by worked in systems, called techniques.

Since objects work freely, they are encapsulated into modules which contain both local environments and techniques. Communication with an object is finished by message passing. Objects are sorted out into classes, from which they acquire techniques and Equivalent variables. The object-oriented paradigm gives key advantages of reusable code and code extensibility.

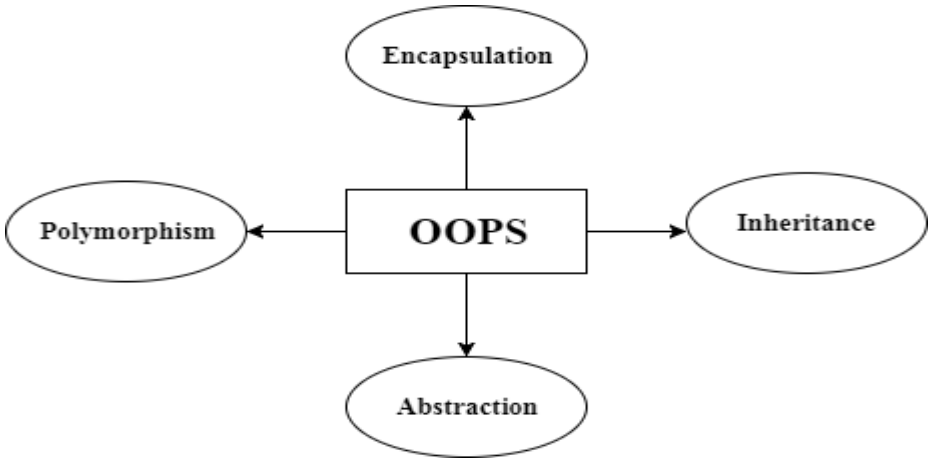

Fig 1.1: Concepts of OOPS [4]

1.2 Features of object-oriented Programming

Features of Object-oriented programming are as follows:

- More reliable software development is possible.
- Enhanced form of c programming language.
- The most important Feature is that it’s procedural and object oriented nature.
- Much suitable for large projects.
- Fairly efficient languages.
- It has the feature of memory management.

1.3 Concepts of OOPS

The basic concepts of OOPS are as follows:

- **Encapsulation and Data abstraction:** Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding. Data abstraction refers to, providing only needed information to the outside world and hiding implementation details. For example, consider a class Complex with public functions as getReal() and getImag(). We may implement the class as an array of size 2 or as two variables. The advantage of abstractions is, we can change implementation at any point, users of Complex class wont't be affected as out method interface remains same. Had our implementation be public, we would not have been able to change it.
- **Inheritance:** inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.
- **Polymorphism:** polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading. Operator overloading is the process of making an operator to exhibit different behaviors in different instances is known as operator overloading. Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.
- **Dynamic Binding:** In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.
- **Message Passing:** Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

1.4 Difference between object oriented and procedure-oriented programming

The following table 1.1 lists out the differences between Object Oriented and Procedural Oriented programming languages.

Table 1.1: POP vs. OOP

BASIS FOR COMPARISON	**POP**	**OOP**
Basic	Procedure/Structure oriented.	Object oriented.
Approach	Top-down.	Bottom-up.
Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program.	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing mode	No access specified observed.	Access specifiers are "public", "private", "protected".
Overloading/ Polymorphism	Neither has it overloaded functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	There is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. Hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Friend functions/ classes Note: "friend" keyword is used only in C++	No concept of friend function.	Classes or function can become a friend of another class with the keyword "friend".
Virtual classes/ function	No concept of virtual classes.	Concepts of virtual function appear during inheritance.
Example	C, VB, FORTRAN, Pascal	C++, JAVA, VB.NET, C#.NET.

1.5 Comparison between Class and Structure

The following table 1.2 lists out the differences between Class and Structure.

Table 1.2: Class vs. Structure

Basis for Comparison	**Class**	**Structure**
Definition	A class in C++ can be defined as a collection of related variables and functions encapsulated in a single	A structure can be referred to as a user defined data type

	structure.	possessing its own operations.
Keyword for the declaration	Class	Struct
Default access specifier	Private	Public

Example	class myclass{
	private:
	int data;
	public:
	myclass(int data_):
	data(data_)
	{}
	virtual void foo()=0;
	virtual ~class()
	{}
	};
		struct myclass{
		private:
		int data;
		public:
		myclass(int data_):
		data(data_)
		{}
		virtual void foo()=0;
		virtual ~class()
		{}
		};

Purpose	Data abstraction and further inheritance	Generally, grouping of data
Type	Reference	Value
Usage	Generally used for large amounts of data.	Generally used for smaller amounts of data.

1.6 Data Types

Data types in any of the language means that what are the various type of data the variables can have in that particular language. Information is stored in a computer memory with different data types. Whenever a variable is declared it becomes necessary to define data type that what will be the type of data that variable can hold.

a) **Primary (Built-in) Data Types**
   - character
   - integer
   - floating point
   - Boolean
   - double floating point
   - void
   - wide character

b) **User Defined Data Types**
   - Structure
   - Union
   - Class
   - Enumeration

c) **Derived Data Types**
   - Array
   - Function
   - Pointer
   - Reference

2.1 Classes and Objects

Object is a collection of number of entities. Objects take up space in the memory. Objects are instances of classes. When a program is executed, the objects interact by sendingmessages to one another. Each object contains data and code to manipulate the data. Objects caninteract without having any details of each other's data or code.

Class is a collection of objects of similar type. Objects are variables of the type class.Once a class has been defined, we can create any number of objects belonging to that class.

2.1.1 Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

When a class is defined only the specification for the object is defined, no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax: Classnameobjectname;

2.1.2Accessing a Class Member

The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

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.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.

2.1.3Defining a member function inside and outside class

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

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

Example 2.1: Defining function inside class

```
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.

Example 2.2: Defining function inside class

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

2.1.4Access Specifiers

Access specifier in a class defines the access control rules.There are three access modifiers: public, private and protected.

2.2Inline Function

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time.

Defining the inline function

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

2.3Static Keyword

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

2.3.1Static Data Members & Member functions

It is a variable which is declared with thestatic keyword, it is also known as class member, thus only single copy of the variable creates for all objects.Any changes in the static data member through one member function will reflect in all other object's member functions.

Declaration:static data_typemember_name

To definestatic data member outside the class:data_typeclass_name :: member_name =value;

2.4Friend function and returning objects to functions

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.The complier knows a given function is a friend function by the use of the keyword friend.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaring Friend Function

```
class class_name
{ ... .. ...

friend return_type function_name(argument/s);

   ... .. ...

}
```

3.1 Constructor

A constructor is a special member function that initializes the objects of its class. It is special because its name is the same as the class name. It is invoked automatically whenever an object is created. It is called constructor because it constructs the values of data members of the class.

Example 3.1: Constructor

```
Class student
{
int rn;
int total ;
public:
student ( )
{
rn = 0 ; total = 0 ;
}
```

3.1.1 Need of constructor

A **constructor** is a special method of a class or structure in **object-oriented programming** that initializes an object of that type. A **constructor** is an instance method that usually has the same name as the class, and can be used to set the values of the members of an object, either to default or to user-defined values.

3.1.2 Types of Constructor

a) Default Constructor

A constructor that accepts no parameter is called default constructor. If no such constructor is defined, then the compiler supplies a default constructor. In that case, it is called nothing-to-do constructor.

b) Parameterized Constructors

The constructors that can take arguments are called parameterized constructors.

Example 3.2: Parameterized Constructor

```
class student
{
int rn, total ;
public
student (int x, int y)
{
rn = x ; total = y ;
}
```

When the object is created, we must supply arguments to the constructor function.
This can be done in two ways:

- By calling the function explicitly
- By calling the function implicitly

The explicit function call is implemented as follows:

```
student S1 = student ( 1, 70 ) ;
```

The implicit function call is implemented as follows:

```
student S1 ( 1, 70 ) ;
```

The second method is used very often as it is shorter.

3.1.3 Copy Constructor

A copy constructor takes a reference to an object of the same class as itself as an argument.

Example 3.3: Copy Constructor

```
class student
{
int rn, total ;
public :
student (int x, int y )
{
rn = x ; total = y ;
}

student (Student & i )
{
rn = i. rn ;
total = i. total ;
}

};
```

3.1.3 Order of calling constructors and destructors

1. Constructors
    a) base class (classes in order of declaration)
    b) class members in order of declaration
    c) constructor's body
2. Destructors – simply opposite order
    a) destructor's body
    b) destructors of class members (order opposite to declaration)
    c) destructor of the base class (classes in order opposite to declaration)

3. objects defined in blocks (local, automatic)
4. constructors are called when the definition is executed (met)

    a) Destructors after leaving the block, order opposite to constructor.

5. global objects (static)
    a) constructors are called in an order of objects' definitions, before calling the main() function
    b) Destructors in order opposite to constructors, after finishing main().

3.1.4 Destructor

It is used to destroy the objects that have been created by a constructor. The destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example the destructor of the class student can be defined as:

- student () ;

It never takes any argument nor does it return any value. It will be invoked by the compiler upon exit from the program (or function or block) to clean the storage. It is a good practice to declare destructor in a program because it releases memory space for future use.

Unit-2

4.1 Inheritance

Inheritance, encapsulation, abstraction, and polymorphism are four fundamental concepts of object-oriented programming. This article focuses on inheritance.

4.1.1 Introduction

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 **base** class, 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.

- 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.
- Derive quality and characteristics from parents or ancestors. Like you inherit features of your parents.

  Example: "She had inherited the beauty of her mother"

- Inheritance in Object Oriented Programming can be described as a process of creating new classes from existing classes.
- New classes inherit some of the properties and behavior of the existing classes. An existing class that is "parent" of a new class is called a base class. New class that inherits properties of the base class is called a derived class.
- Inheritance is a technique of code reuse. It also provides possibility to extend existing classes by creating derived classes.When an existing class is inherited, all its methods and fields become available in the new class, hence code is reused.
- All members of a class except Private, are inherited.

4.1.2 Basic Syntax of Inheritance in C++

```
class Subclass_name : access_modeSuperclass_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 super class will be inherited into subclass, public, private or protected.

Example 4.1: Inheritance in C++

```
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

4.1.3 Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

- **Public mode**: In this mode, the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
- **Protected mode**: In this mode, both public member and protected members of the base class will become protected in derived class. Private members of the base class will never get inherited in sub class.
- **Private mode**: In this mode, both public member and protected members of the base class will become Private in derived class. Private members of the base class will never get inherited in sub class.

The below table 4.1 summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes.

Table 4.1: Access Specifiers [5]

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

4.1.4 Types of Inheritance

In C++, there are 5 different types of Inheritance. Namely,

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)

a) Single Inheritance

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

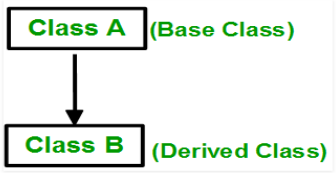

Fig 4.1: Single Heritance [5]

b) Multiple Inheritance

Multiple Inheritances is a feature of C++ where a class can inherit from more than one classes i.e. one **sub class** is inherited from more than one **base classes**.

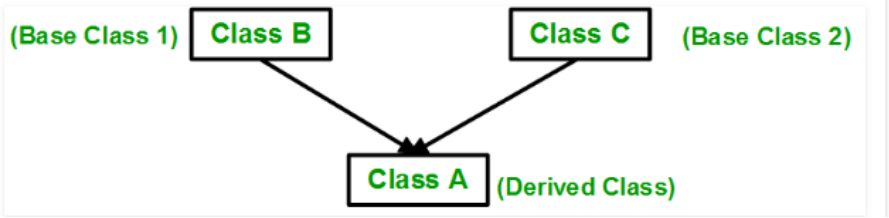

Fig 4.2: Multiple Inheritance [5]

c) Multilevel Inheritance

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

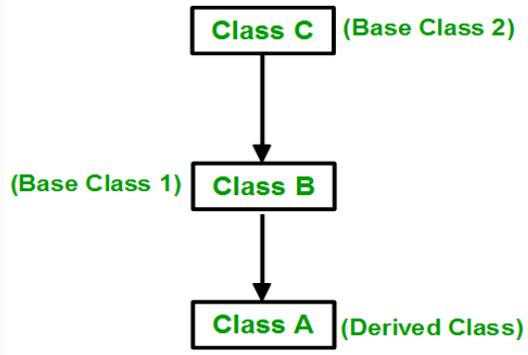

Fig 4.3: Multilevel Inheritance [5]

d) Hierarchical Inheritance

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

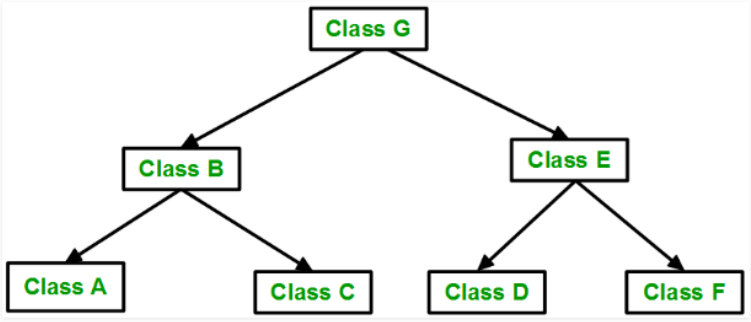

Fig 4.4: Hierarchical Inheritance [5]

e) Hybrid (Virtual) Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. Fig 4.5 shows the combination of hierarchical and multiple inheritances.

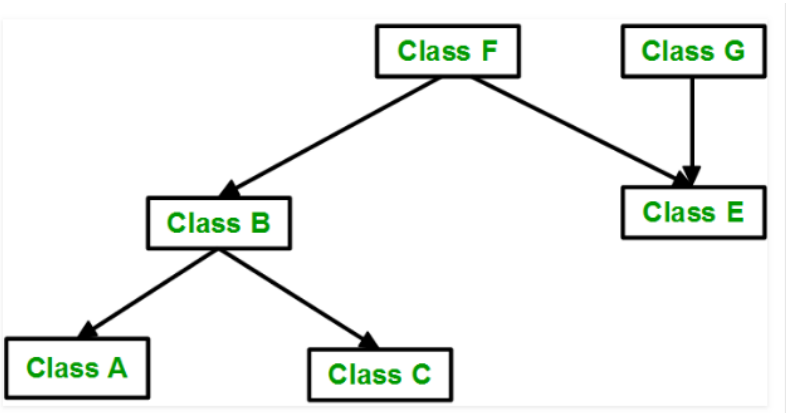

Fig 4.5: Hybrid Inheritance [5]

5.1 Polymorphism

5.1.1 Introduction

- The word polymorphism means having many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee. So a same person possess have different behavior in different situations.
- Polymorphism is considered as one of the important features of Object Oriented Programming.

5.1.2 Types of Polymorphism

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

1. Compile time polymorphism:

Function overloading or operator overloading achieves this type of polymorphism.

a) Function Overloading

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

Example 5.1: Function Overloading

```
#include <bits/stdc++.h>
using namespace std;
class ABC{
   public:
   void func(int x)                    // function with 1 int parameter
   {
      cout << "value of x is " << x << endl; }
   void func(double x)                 // function with same name but 1 double parameter
   {
      cout << "value of x is " << x << endl;}
   void func(int x, int y)             // function with same name and 2 int parameters
   {
      cout << "value of x and y is " << x << ", " << y << endl;
   }};
int main() {
   ABC obj1;
   obj1.func (7);
   obj1.func (9.132);
   obj1.func (85,64);
   return 0; }
```

b) Operator Overloading

C++ also provides option to overload operators.

For example, we can make the operator (‘+’) for string class to concatenate two strings.

We know that this is addition operator whose task is to add to operands. So a single operator ‘+’ when placed between integer operands, adds them and placed between string operands, concatenatesthem.

Example 5.2: Operator Overloading

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

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;
   }
   void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
```

2. Runtime polymorphism

Function Overriding achieves this type of polymorphism.

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

Example 5.3: Function Overriding

```
#include <bits/stdc++.h>
using namespace std;
 class Parent
{
   public:
   void print()
   {
      cout << "The Parent print function was called" << endl;
   }
};
class Child : public Parent
{
   public:
   void print()          // definition of a member function already present in Parent
   {
      cout << "The child print function was called" << endl;
   }   };
 int main()  {
   //object of parent class
   Parent obj1;
   //object of child class
   Child obj2 = Child();
   // obj1 will call the print function in Parent
   obj1.print();
   // obj2 will override the print function in Parent
   // and call the print function in Child
   obj2.print();
   return 0;
}
```

5.1.3 Rules for overloading operators

Only existing operators can be overloaded. New operators cannot be overloaded.

a) The overloaded operator must have at least one operand that is of user defined type.

b) We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.

c) Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.

d) There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(.), pointer to member operator(.*), scope resolution operator(::), conditional operators(?:) etc

e) We cannot use “friend” functions to overload certain operators.However, member function can be used to overload them. Friend Functions cannot be used with assignment operator(=), function call operator(()), subscripting operator([]), class member access operator(->) etc.

f) Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).

g) Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

h) When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

i) Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

5.1.4Overloading Unary and Binary Operators

a) 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--.

Example 5.4: Overloading Unary Operator

```
#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; }
```

b) 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.

5.1.5 Constructor Overloading

Constructor can be overloaded in a similar way as function overloading.

Overloaded constructors have the same name (name of the class) but different number of arguments.Depending upon the number and type of arguments passed, specific constructor is called.

Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Example 5.5: Overloading Binary Operator

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

Example 5.6: Constructor Overloading

```
#include <iostream>
using namespace std;
class Area{
   private:
 int length;
int breadth
   public:
      // Constructor with no arguments
      Area(): length(5), breadth(2) { }
      // Constructor with two arguments
      Area(int l, int b): length(l), breadth(b){ }
      void GetLength(){
         cout << "Enter length and breadth respectively: ";
         cin >> length >> breadth; }
      int AreaCalculation() {  return length * breadth;  }
      void DisplayArea(int temp)
      {
         cout << "Area: " << temp << endl;}
};
int main(){
   Area A1, A2(2, 1);
   int temp;
   cout << "Default Area when no argument is passed." << endl;
   temp = A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout << "Area when (2,1) is passed as argument." << endl;
   temp = A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0; }
```

6.1 EXCEPTION HANDLING

6.1.1 Introduction

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

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

a) **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.

b) **catch:** A program catches an exception with an exception handler at the place in a programwhere you want to handle the problem. The **catch** keyword indicates the catching of an exception.

c) **try:** A **try** block identifies a block of code for which particular exceptions will be activated.

It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following example 8.1.

Example 6.1:

```
try
{
    // protected code
}
catch( ExceptionName e1 )
{
    // catch block
}
catch( ExceptionName e2 )
{
    // catch block
}
catch( ExceptionName eN )
{
    // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

6.1.2 Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following example 7.2 is of throwing an exception when dividing by zero condition occurs.

Example 6.2

```
double division(int a, int b)
{
    if( b == 0 )
    {
        throw "Division by zero condition!";
    }
    return (a/b);
}
```

6.1.3 Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

Example 6.3

```
try
{
      // protected code
}
catch( ExceptionName e )
{
    // code to handle ExceptionName exception
}
```

Above example 7.3 will catch an exception of **ExceptionName** type.

In the below example 7.4, which throws a division by zero exception and we catch it in catch block.

Example 6.4:

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

 double division(int a, int b)
 {
       if( b == 0 )
       {
            throw "Division by zero condition!";
       }
       return (a/b);
 }
 int main ()
 {
       int x = 50;
       int y = 0;
       double z = 0;

       try {
           z   =divisio
           n(x, y); cout
           << z <<
           endl;
       }
       catch (const char* msg) {
           cerr << msg << endl;
       }

       return 0;
 }
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result:

Division by zero condition!

Unit-3

7.1 Pointers

A **pointer** is a derived datatype that refers to another data variable by storing memory address rather than data.

7.1.1 Declaring and Initializing Pointers

The following is the syntax used to declare a pointer.

Syntax

datatype *pointer_variable;

The pointer_variable is the name of the pointer, and datatype is any valid C++ datatype (either primary datatype or secondary datatype).

Initialization of Pointers

Int *ptr, a; // declaration

Ptr=&a; // initialization

7.1.2 Pointer to Objects

Example 6.1:

item x;

item *ptr = &x;

A pointer can point to an object created by a class.

In the above example 6.1, item is a class and x is an object defined to be of type item. Similarly we define a pointer ptr of type item.

7.1.3Pointers to Derived Classes

Pointers are not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes.

Consider example 6.2, if B is a base class and D is a derived class from B, and then a pointer declared as a pointer to B can also be a pointer to D.

7.1.4 Static and dynamic binding

The pointer (or reference) type is known at compile time while object type might be determined at runtime. Interpreting a function call in the source code as executing a particular block of function code is called **binding** the function name. Binding that takes place during compile time is **static binding** or **early binding**. With the virtual function, the binding task is more difficult. The decision of which function to use can't be made at compile time because the compiler doesn't know which object the user is going to choose to make.In other words, when a request (message) is sent to an object, the particular operation that's performed depends on **both** the request **and** receiving object. Different objects that support identical request may have different implementation of the operations that fulfill these requests. The run-time association of a **request** to an object and one of its **operations** is known as **dynamic binding**. This means that issuing a request doesn't commit us to a particular implementation until run-time. So, we can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.

8.1 Dynamic Memory

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

If you are not in need of dynamically allocated memory anymore, you can use delete operator, which de-allocates memory that was previously allocated by new operator.

8.1.1 New and Delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, data-type could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows.

```
delete pvalue;        // Release memory pointed to by pvalue
```

9.1 File Handling in C++

Files are used frequently for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files.

Files are not only used for data. Our programs are also stored in files.

The editor which you use to enter your program and save it simply manipulates files for you.

The UNIX commands cat, cp, cmp are all programs which process your files.

In order to use files we have to learn about ***File I/O*** i.e. how to write information to a file and how to read information from a file.

File I/O is almost identical to the terminal I/O that we have being using so far.

The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use.

As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use.

When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both.

Because you may use a number of different files in your program, you must specify when reading or writing which file you wishes to use. This is accomplished by using a variable called a **file pointer.**

Every file you open has its own file pointer variable.

Deaclaring File Pointer:

```
FILE  *fopen(), *fp1, *fp2, *fp3;
```

The variables fp1, fp2, fp3 are file pointers.

The file <stdio.h> contains declarations for the Standard I/O library and should always be **include**d at the very beginning of C programs using files.

Constants such as FILE, EOF and NULL are defined in <stdio.h>.

Note: A file pointer is simply a variable like an integer or character.

It does **not***point* to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to.

A file number is used in the Basic language and a unit number is used in FORTRAN for the same purpose.

9.1.1 Opening of the file

The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

```
fp = fopen( "prog.c",  "r") ;
```

The above example **opens** a file called prog.c for **reading** and associates the file pointer fp with the file.

To access file for I/O, use the file pointer variable fp to refer to it.

9.1.2 File I/O

The Standard I/O Library provides similar routines for file I/O to those used for standard I/O.
The routine getc(fp) is similar to getchar() and putc(c,fp) is similar to putchar(c).

Thus the statement

```
c = getc(fp);
```

reads the next character from the file referenced by fp and the statement

```
putc(c,fp);
```

writes the character c into file referenced by fp.

References

[1]http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html

[2]http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html

[3] http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html

[4] http://www.studytonight.com/cpp/cpp-and-oops-concepts.php

[5] https://www.geeksforgeeks.org/inheritance-in-c

[6] https://www.computerhope.com/jargon/i/imp-programming.htm

[7] http://people.cs.aau.dk/paradigms_themes-paradigm-overview-section.html

[8]https://www.tutorialspoint.com/functional_programming/functional_programming introduction.html

[9] http://www.doc.ic.ac.uk

[10] https://www.tutorialspoint.com/cplusplus/cpp_operators.htm

[11] http://www.studytonight.com/cpp/operators-and-their-types.php

[12] http://www.cplusplus.com/doc/tutorial/operators

[13] https://books.google.co.in/books?isbn=0070669074

[14] E Balagurusamy., "Object Oriented Programming in C++", Tata McGraw-Hill.

[15] https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm

[16] http://www.studytonight.com/cpp/class-and-objects.php

[17] http://www.cplusplus.com/doc/tutorial/classes

[18] https://zh.scribd.com/presentation/94885768/Constructor-PPT

[19] www.slideshare.net/vinod242306/constructor-ppt

[20] https://www.pdfdrive.net/e_balagurusamy-object_oriented_programming_with cpdf-d28853085.html

[21] https://www.computerhope.com/jargon/i/imp-programming.htm

[22] http://people.cs.aau.dk/paradigms_themes-paradigm-overview-section.html

[23] http://www.doc.ic.ac.uk

[24] https://en.wikipedia.org/wiki/Functional_programming

[25] https://www.tutorialspoint.com/functional_programming/functional_programming introduction.htm

[26] www.cs.nott.ac.uk/~pszgmh/functional.ppt

[27] http://wiki.c2.com/?FunctionalProgramming

[28] https://en.wikipedia.org/wiki/Logic_programming

[30] https://www.nku.edu/~foxr/CSC407/NOTES/ch16.ppt

[31] www.cs.uah.edu/~weisskop/Notes424-524/ch15b-withReview.ppt

[32] https://www.cs.jhu.edu/~jason/325/PowerPoint/14prolog.ppt

[33] courses.cs.washington.edu/courses/cse341/00wi/ppt-src/Logic-Prog-Intro.ppt

[34] cse.iitkgp.ac.in/~pallab/PDS-2011-SPRING/Lec-9.pdf

[35] https://web.cs.wpi.edu/~cs2303/c10/Protected/...C10/Week7_ExceptionHandling.ppt

[36] www.it.uom.gr/teaching/deitel/chtp3e15to23ppt/C_chap23.ppt

[37] https://www.cse.unr.edu/~bebis/CS308/PowerPoint/Inheritance.ppt

[38] https://cs.wmich.edu/~rhardin/Spring04/cs112/lecturespecial.ppt

[39] https://www.slideshare.net/gauravsitu/polymorphism-12270448

[40] https://www.slideshare.net/tareq1988/08-c-operator-overloadingppt