Semester-2

Created by Sudhanshu Sharma

Unit-1
Object modeling

Object and Classes-

- **Class-** A template or blueprint that defines the characteristics of an object and describes how the object should look and behave.
- **Object-**Object is an instance of a class. Combining both data and member functions. Objects are the basic run-time entities in an object oriented system.

Links and Association-

- Association is a relationship between two objects. In other words, association defines the multiplicity between objects.
- When the association is between two classes, its called Binary Association. When the association is between two instances of the same class, then its called reflexive or unary association.
- **Example : Binary Association :** Employee works for Employee **Unary Association:** A Employee supervises Employees

Link-Link is the relationship between two objects. There are no multiplicity concepts in links. Links always exists between two objects.

Generalization And Inheritance

Generalization-

- Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass.

Inheritance-

- 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.

Aggregation-

- Aggregation is a way of composing different abstractions together in defining a class. For example, a car class can be defined to contain other classes such as engine class, seat class, wheels class etc.

Abstract Class-

- An abstract class is a class that is designed to be specifically used as a baseclass. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

Multiple Inheritance-

- Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inheritcharacteristics and features from more than one parent object or parent class.

Meta Data-

- Metadata is data that describes other data. Meta is a prefix that in most information technology usages means "an underlying definition or description."

Metadata is data that describes other data. Meta is a prefix that in most information technology usages means "an underlying definition or description."

Candidate Key-

- A candidate key is a column, or set of columns, in a table that can uniquely identify any database record without referring to any other data. Each table may have one or more candidate keys, but one candidate key is unique, and it is called the primary key.

Constraints-

- A constraint is a packageable element which represents some condition, restriction or assertion related to some element (that owns the constraint) or several elements. Constraint is usually specified by a Boolean expression which must evaluate to a true or false.

Unit-2
Dynamic Modeling

Dynamic Modeling is used to represent the behavior of the static constituents of a software , here static constituents includes, classes , objects, their relationships and interfaces Dynamic Modeling also used to represents the interaction, workflow, and different states of the static constituents in a software.

Event And States

Events-

- Events are some occurrences that can trigger state transition of an object or a group of objects. Events have a location in time and space but do not have a time period associated with it.
  Examples of events are mouse click, key press, an interrupt, stack overflow, etc.

States-

- State transition diagrams or state machines describe the dynamic behavior of a single object. It illustrates the sequences of states that an object goes through in its lifetime, the transitions of the states, the events and conditions causing the transition and the responses due to the events.

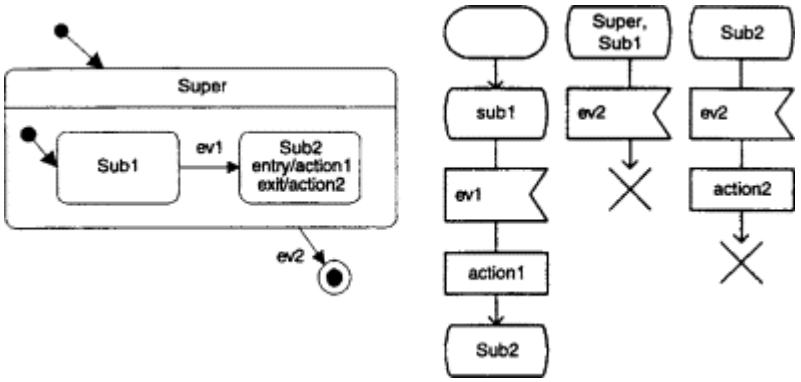
Operations-

- Activity is an operation upon the states of an object that requires some time period. They are the ongoing executions within a system that can be interrupted.

Activities are shown in activity diagrams that portray the flow from one activity to another.

Nested State Diagrams and Concurrency-

- In addition to states of an order that are based on the availability of the items, there are also states that are based on payment authorization. If we look at these states, we might see a state diagram like the one in Figure.



Advanced Dynamic Modeling Concepts-

- Entry and exit actions are features of advanced dynamic modelling.
- An entry action is performed when any transition enters the state and an exit action is performed when a state is exited.

A Sample Dynamic Model-

- The dynamic model is used to express and model the behaviour of the system over time. ... Sequence diagrams are used to display the interaction between users, screens, objects and entities within the system. It provides a sequential map of message passing between objects over time.

Functional Modeling

Data Flow Diagram-

- A Data Flow Diagram (DFD) is traditional visual representation of the information flows within a system.
- It shows how information enters and leaves the system, what changes the information and where information is

- It may be used as a communications tool between a analyst and any person who plays a part in the system that acts as the starting point for redesigning a system.

Specifying Operations-

- Functional Modelling gives the process perspective of the object-oriented analysis model and an overview of what the system is supposed to do.
- It defines the function of the internal processes in the system with the aid of Data Flow Diagrams (DFDs).
- It depicts the functional derivation of the data values without indicating how they are derived when they are computed, or why they need to be computed.

Constraints-

- A constraint is a packageable element which represents some condition, restriction or assertion related to some element

(that owns the constraint) or several elements. Constraint is usually specified by a Boolean expression which must evaluate to a true or false.

A Sample Functional Model-

- Functional Modelling gives the process perspective of the object-oriented analysis model and an overview of what the system is supposed to do. It defines the function of the internal processes in the system with the aid of Data Flow Diagrams (DFDs).

OMT(Object modeling techniques)Methdologies-

- The object modeling techniques is an methodology of object oriented analysis, design and implementation that focuses on creating a model of objects from the real world and then to use this model to develop object–oriented software.
- Object modeling technique, OMT was developed by James Rambaugh.
- Now-a-days, OMT is one of the most popular object

Unit-3
Introduction

OOP Paradigm-

- Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability.
- Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

Basic concepts-

=>there are various type of basic concept such as

1) Class
2) Object
3) Data Abstaraction
4) Polymorhpihsm
5) Encapsulation
6) Inheritance
7) Dynamic Binding
8) Message passing

Benefits and its applications

Benefits of OOP=>

- It is easy to model a real system as real objects are represented by programming objects in OOP. The objects are processed by
  member data and functions. It is easy to analyze the user requirements.
- With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated and the use of existing class is extended. This saves time and.

- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that can not be invaded by code in other part of the program.
- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.
- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.
- It is possible to have multiple instances of an object to co-exist without any interference i.e. each object has its own separate member data and function.

Basics of C++ -

- In this section we will cover the basics of C++, it will include the syntax, variable, operators, loop types, pointers, references and information about other requirements of a C++ program. You will come across lot of terms that you have already studied in C language.

Concept of Structure and Class –

Class-

- A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the

three access specifiers private, protected or public (by default access to members of a class is private).

- The private members are not accessible outside the class; they can be accessed only through methods of the class. The public members form an interface to the class and are accessible outside the class.

Structure-

- Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collecion of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

Private and Public members–

- Public- public means everyone is allowed to access.
- Private- private means that only members of the same class are allowed to access.

Tokens–

- A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of
  these tokens makes up a translation unit.

Data Types–

- In computer science and computer programming, a data type or simply type is a classification of data which tells the compiler or interpreter how the programmer intends to use the data.
- Most programming languages support various types of data for example: real, integer or Boolean.

Dynamic Initialization–

- According to the C/C++ standards global variables should be initialized before entering main(). In the above program, variable 'i' should be initialized by return value of function alpha(). Since the return value is not known

until the program is actually executed, this is called dynamic initialization of variable.

Referance Variables–

- A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

```
// C++ Program to demonstrate
// use of references
#include <iostream>
using namespace std;

int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

Operators–

- An operator is a character that represents an action, as for example x is an arithmetic operator that represents

multiplication. In computer programs, one of the most familiar sets ofoperators, the Boolean operators, is used to work with true/false values.

Dynamic Memory Allocation–

- C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to

free dynamically allocated memory. C++supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

Manipulators–

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

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

Control Structure–

- A program is usually not limited to alinear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

Functions In C++

Introduction-

- A function is a group of statements that together perform a task. Every C++ program has at least onefunction, which is main(), and all the most trivial programs can define additional functions. You can divide up your code into separate functions.

Main()function-

- the role of main() is to indicate to the compiler to convert source code from { open curly bracket to } close curly bracket.

Prototyping-

- The Prototyping Model is a systems development method (SDM) in which a prototype (an early approximation of a final system or product) is built, tested, and then reworked as necessary until an acceptable prototype is finally achieved from which the complete system or product can now be developed.

Call and return by reference-

```
#include<iostram.h>
int sum(int *a,int *b);
void main()
{
int a=5,b=6;
int c;
c=sum(&a,&b);
cout<<"Sum of Two NUmber="<<c;
getch();
```

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

Inline function-

- The inline functions are a C++ enhancement feature to increase the execution time of a program.
- Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.

The syntax for defining the function inline is:-

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

- ```
  #include <iostream>
  using namespace std;
  inline int cube(int s)
  {
   return s*s*s;
  }
  int main()
  {
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
  } //Output: The cube of 3 is: 27
  ```

Default arguments-
  ```

- A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value. Following is a simple C++ example to demonstrate use of default arguments.

Exm-

```
// CPP Program to demonstrate Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w
as 0
{
    return (x + y + z + w);
}

// Driver Code
int main()
{
    // Statement 1
    cout << sum(10, 15) << endl;

    // Statement 2
    cout << sum(10, 15, 25) << endl;

    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

function overloading-

- C++ allows specification of more than one function of the same name in the same scope. These are called overloaded functions and are described in detail in

Overloading. Overloaded functions enable programmers to supply different semantics for a function, depending on the types and number of arguments.

Exm-

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


void add(int a, int b)
{
cout << "sum = " << (a + b);
}

void add(double a, double b)
{
	cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
	add(10, 2);
	add(5.3, 6.2);

	return 0;
}
```

friend functions-

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

Exm-

```
#include <iostream>
class A {
private:
 int a;
public:
 A() { a=0; }
 friend class B; // Friend Class
};

class B {
private:
 int b;
public:
 void showA(A& x) {
 // Since B is friend of A, it can access
 // private members of A
 std::cout << "A::a=" << x.a;
 }
};

int main() {
 A a;
 B b;
 b.showA(a);
 return 0;
}
```

Private Member Functions-

```
# include <iostream.h>
# include <conio.h>
class student
{
 private:
 int rn;
 float fees;
```

```
void read()
{
rn=12;
fees=145.10;
}
public:
void show()
{
read();
cout<<"\n Rollno = "<<rn;
cout<<"\n Fees = "<<fees;
}


};
void main ( )
{
clrscr ( );
student st;
// st.read ( ); // not accessible
st.show ( );
getch();
}
```

Various storage classes-

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program=
  - o Auto
  - o register
  - o static
  - o extern
  - o mutable

The auto Storage Class-

- The auto storage class is the default storage class for all

local variables.

```
{
 int mount;
 auto int month;
}
```

The register Storage Class-

- The register storage class is used to define local variables that should be stored in a register instead of RAM.

```
{
 register int miles;
}
```

The static Storage Class-

- The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

Exm-

```
// C++ Program to illustrate the auto storage class
// variables
#include <iostream>
using namespace std;

void autoStorageClass()
{

    cout << "Demonstrating auto class\n";

    // Declaring an auto variable
    int a = 32;
    float b = 3.2;
    char* c = "GeeksforGeeks";
    char d = 'G';

    // printing the auto variables
    cout << a << " \n";
```

```
    cout << b << " \n";
    cout << c << " \n";
    cout << d << " \n";
}

int main()
{

    // To demonstrate auto Storage Class
    autoStorageClass();

    return 0;
}
```

The extern Storage Class-

- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been

previously defined.

The mutable Storage Class-

- The mutable specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function.

Static member functions-

- 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.

Unit-4
Constructor And Destructor

Introduction-

- constructor is a very special member function whose name is same as class name. consturtor is self executable when object is created. consturctor is always declare inside a public mode. the role of the consturcotr is inilized the value.
- constructor is not required any kind of return type not even void

Type of constuctor-

- constructe can be categoriese in 4 type

1) default
2) non paramertized
3) paramerized constructor
4) copy cocnstrucor

Parameterized constructors-

- It may be necessary to initialize the various data elements of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

Multiple constructors in a class-

```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;
```

```
class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
            area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
            area = a * b;
    }

    void disp()
    {
            cout<< area<< endl;
    }
};

int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

Constructors with default arguments-

Default arguments-

As the name says, it is a function argument with pre-specified value. For example, in function declaration 'void foo(int x, int y=10)', argument y has default value of 10. This also means that when you call this function, you have a choice of not passing value for y and its value will be taken as 10 inside the function foo.

Default Constructor-

As per C++ language, a constructor is considered to be default one if it does not have any arguments or it has arguments, but all the arguments have default values. As an example for a class named Foo, the following constructors will be default one.

```
Foo();
```

or

```
Foo(int x=10, int y=30);
```

In both the cases, even if you don't provide parameter values, you can get a valid object. As a follow up exercise, try putting both the above constructors in a class and try creating an object without passing any parameter. Find out what compiler has to say about it.

Dynamic initialization of objects-

Dynamic initialization of object refers to initializing the objects at run time i.e. the initial value of an object is to be provided during run time. Dynamic initialization can be achieved using constructors and passing parameters values to the constructors.

```
#include<isotram.h>
class number
{
int a,b;
public :
number(int i, int j)
{
a=i;
b=j'
}
void show()
{
cout<<a<<b;
}
void main()
{
number n(10,20);
n.show();
getch();
}
```

Copy constructor-

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

ClassName (const ClassName &old_obj);

Following is a simple example of copy constructor.

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
        int rno;
        char name[50];
        double fee;
        public:
        student(int,char[],double);
```

```
        student(student &t)   //copy constructor
        {
                rno=t.rno;
                strcpy(name,t.name);
                fee=t.fee;
        }
        void display();

};




        student::student(int no,char n[],double f)
        {
                rno=no;
                strcpy(name,n);
                fee=f;
        }

void student::display()
        {
                cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
        }

int main()
{
        student s(1001,"Manjeet",10000);
        s.display();

        student manjeet(s); //copy constructor called
        manjeet.display();

        return 0;
}
```

Destructors-

- 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.

```
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;
};
```

Operator Overloading-

Introduction-

- Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.). Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator.

```
// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading
class A {
       statements;
};
```

```
int main()
{
        A a1, a2, a3;

        a3 = a1 + a2;

        return 0;
}
```

Method of overloading-

- Function Overloading

- If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.
- Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but

with different number or types of arguments, then you can simply overload the function.

```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Overloading unary and binary operators-

- an unary operator is used, it works with one operand, therefore with the user defined data types, the operand becomes the caller and hence no arguments are required.
- Take a look at the following unary operator overloading example, in this case the unary operators increment (++) and decrement (–):

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

//Increment and decrement overloading
class Inc {
 private:
 int count ;
 public:
 Inc() {
 //Default constructor
 count = 0 ;
 }

 Inc(int C) {
 // Constructor with Argument   count = C ;
 }

 Inc operator ++ () {
 // Operator Function Definition   return
Inc(++count);   }

 Inc operator -- () {
 // Operator Function Definition   return
Inc(--count);
 }

 void display(void) {
 cout << count << endl ;   }
};
```

```
void main(void) {
 Inc a, b(4), c, d, e(1), f(4);

 cout << "Before using the operator ++()\n";
cout << "a = ";
 a.display();
 cout << "b = ";
 b.display();

 ++a;
 b++;

 cout << "After using the operator ++()\n";
cout << "a = ";
 a.display();
 cout << "b = ";
 b.display();
 c = ++a;
 d = b++;

 cout << "Result prefix (on a) and postfix (on
b)\n";
 cout << "c = ";
 c.display();
 cout << "d = ";
 d.display();

        cout << "Before using the operator --()\n";
 cout << "e = ";
 e.display();
 cout << "f = ";
 f.display();

 --e;
 f--;

         cout << "After using the operator --()\n";
 cout << "e = ";
 e.display();
```

```
 cout << "f = ";
 f.display();
}
```

Manipulation of strings using operators-

- C++ Manipulation of Strings. Manipulating of strings in C++ by operator overloading using character arrays, pointers and string functions. There are no operators for manipulating the strings. There are no direct operator that could act upon the strings or manipulate the strings.

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

int main ()
{
 string First = "This is First String and ";
string Second = "This is Second String.";
 string Third = First + Second;

 cout << Third;

 return 0;
}
```

Rules for overloading operators-

- In C++, following are the general rules for operator overloading

1) Only built-in operators can be overloaded. New operators can not be created.
2) Arity of the operators cannot be changed.
3) Precedence and associativity of the operators cannot be changed.
4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments. 5) Operators cannot be overloaded for built in

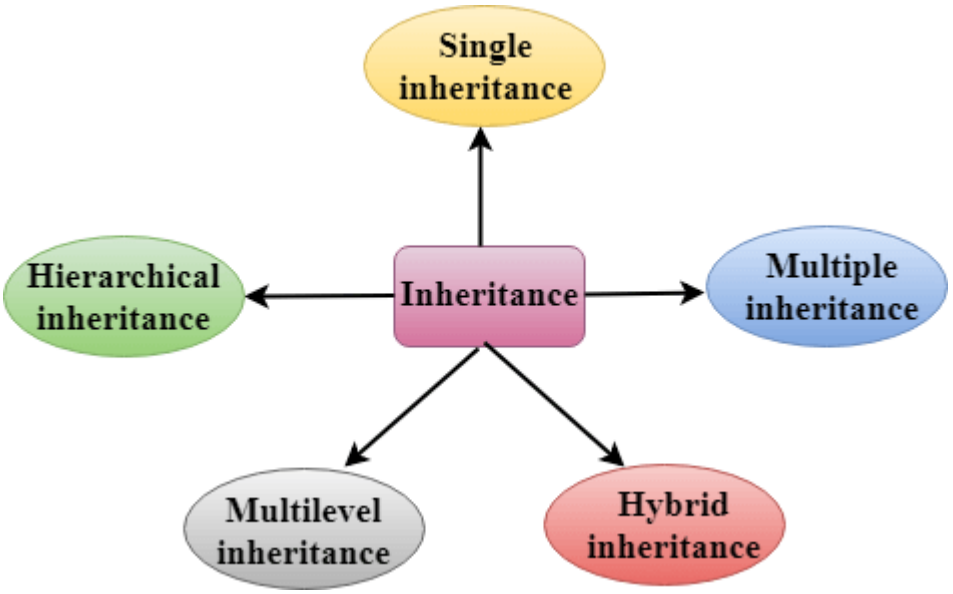
types only. At least one operand must be used defined type.
6) Assignment (=), subscript ([]), function call (“()”), and member selection (->) operators must be defined as member functions
7) Except the operators specified in point 6, all other operators can be either member functions or a non member functions. 8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

Unit-5

Inheritance

Definition-

- 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. ... Inheritance is a technique of code reuse.



Base and derived classes-

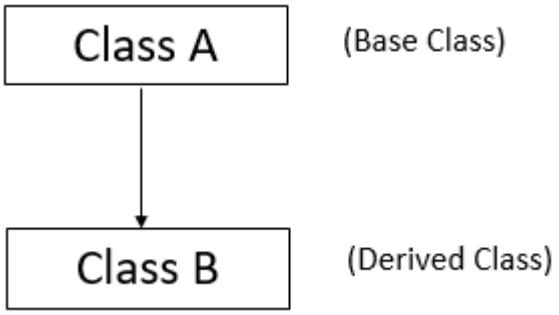
- A class can be used as the base class for a derived new class. The derived class inherits all of the properties of the base class. The derived class can add new members or change base class members. ... In a C++ program the OOP paradigm is centered around your class definitions.

Type of inheritance and their implementation-

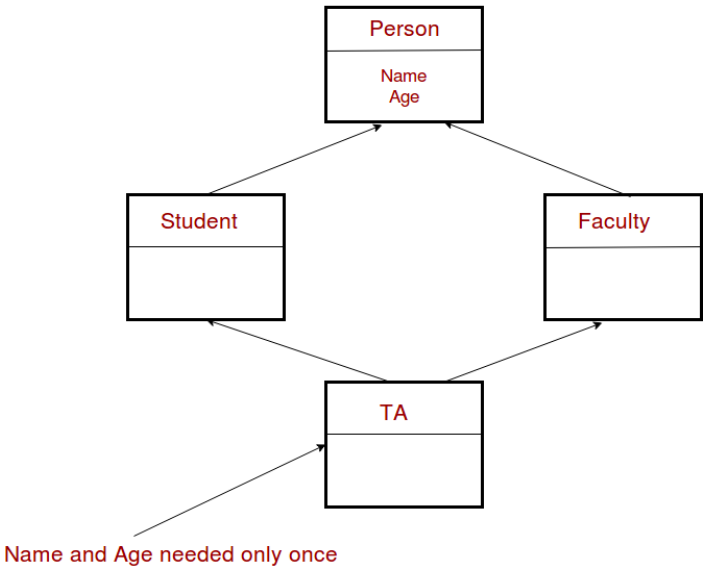
- In C++, we have 5 different types of Inheritance 1)Single Inheritance 2)Multiple Inheritance

3)Hierarchical Inheritance
4)Multilevel Inheritance
5)Hybrid Inheritance (also known as Virtual Inheritance)

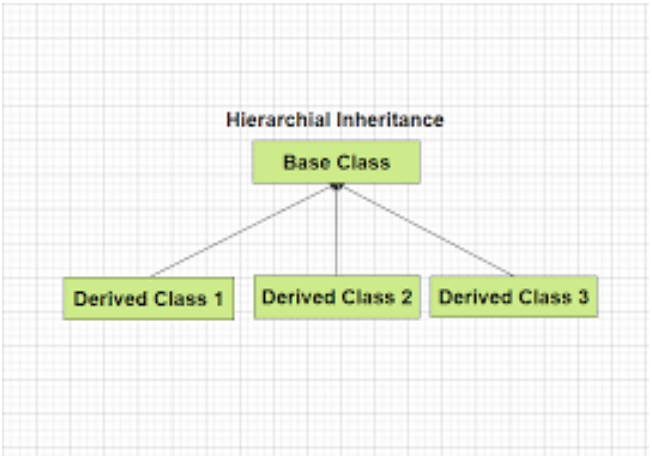
1)Single Inheritance



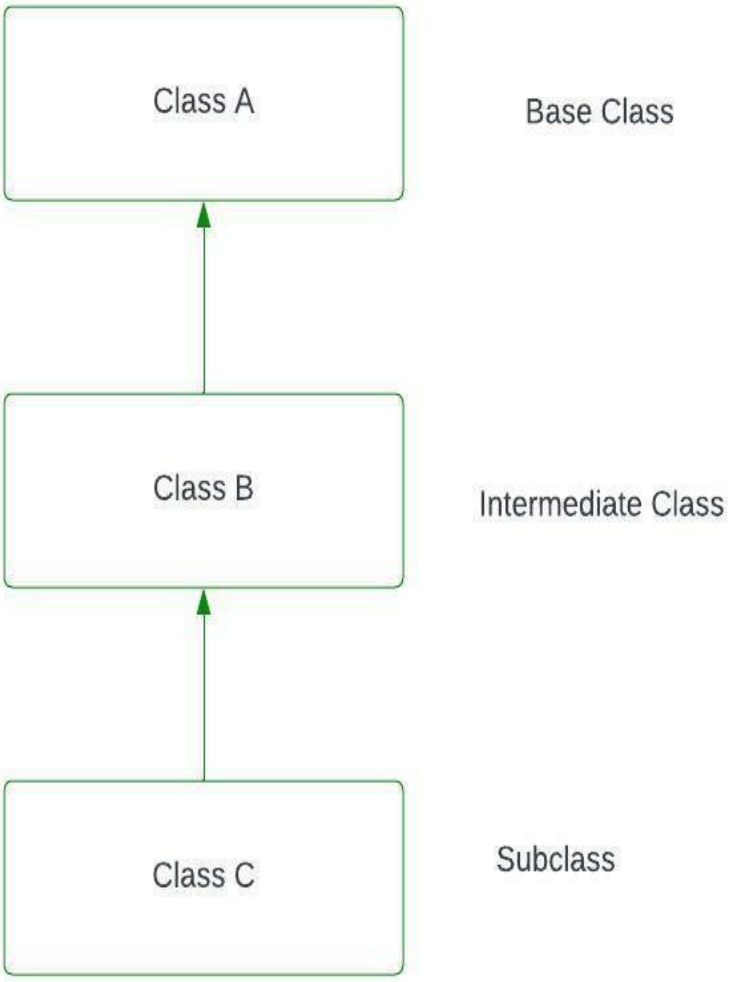
2)Multiple-Inheritance



3)Hierarchical Inheritance



4)Multilevel Inheritance



Virtual base classes-

- Virtual base classes (C++ only) Suppose you have two derived classes B and C that have a common base class A , and you also have another class D that inherits from B and C . You can declare the base class A as virtual to ensure that B and C share the same subobject of A .

```
class A
{
 public:
 int i;
};

class B : virtual public A
{
 public:
 int j;
};
```

```
class C: virtual public A
{
 public:
 int k;
};

class D: public B, public C
{
 public:
 int sum;
};
int main()
{
 D ob;
 ob.i = 10; //unambiguous since only one copy of i is
inherited.
 ob.j = 20;
 ob.k = 30;
 ob.sum = ob.i + ob.j + ob.k;
 cout << “Value of i is : ”<< ob.i<<”\n”;
 cout << “Value of j is : ”<< ob.j<<”\n”; cout << “Value
of k is :”<< ob.k<<”\n”;
 cout << “Sum is : ”<< ob.sum <<”\n”;
 return 0;
}
```

Abstract class-

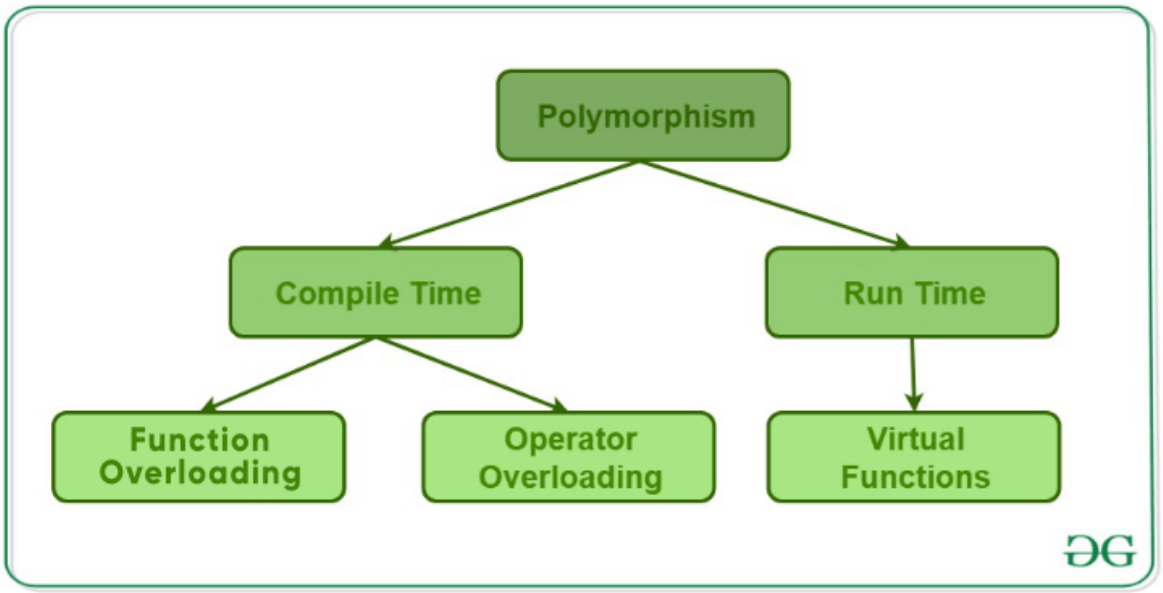
- Abstract class is declared in useof base class and contained at least one pure vartual classes.
- An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

Dynamic Polymorphism

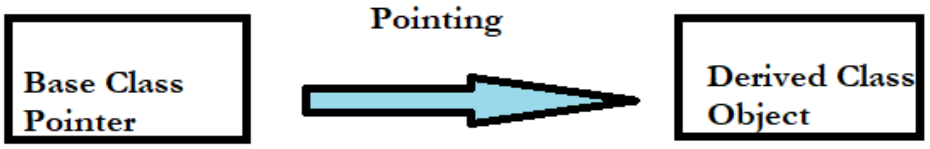
Introduction-

- Dynamic (run time) polymorphism is the polymorphism existed at run-time. Here, Java compiler does not understand which method is called at compilation time. Only JVM decides which method is called at run-time. Method overloading and method overriding using instance methods are the examples for dynamic polymorphism.
- Dynamic polymorphism, which in C++ is called Overriding, allows us to determine the actual function method to be executed at run-time rather than compile time.
- For example, if we are using the uC/OS-II RTOS and have developed a Mutex class, e.g.

```
class uCMutex
{
public:
 uCMutex();
 void lock();
 void unlock();
private:
 OS_EVENT* hSem;
 INT8U err;
 // not implemented
 uCMutex( const uCMutex& copyMe );
  uCMutex& operator=( const uCMutex& rhs );
};
```



Pointers to derived class



We Point Derived class object using Base Class Pointer

Virtual functions-

- A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a

reference to the base class, you can call a virtual function for that object and execute the derived class's version of the
function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime

polymorphism. Functions are declared with a virtual keyword in base class. The resolving of function call is done at Run-time.

Pure Virtual functions-

- A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract. Classes containing pure virtual methods are termed "abstract" and they cannot be instantiated directly.

‘this’ pointer in C++ -

- The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

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

/* local variable is same as a member's name */
class Test
{
private:
 int x;
public:
 void setX (int x)
 {
 // The 'this' pointer is used to retrieve the object's  x
 // hidden by the local variable 'x'
 this->x = x;
 }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
```

```
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
   }
```

- The pointers pointing to objects are referred to as Object Pointers.
- C++ Declaration and Use of Object Pointers

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

- **class-name * object-pointer ;**

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare optr as an object pointer of Sample class type, we shall write.

- **Sample *optr ;**
  where Sample is already defined class. When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

The following program illustrates how to access an object given a pointer to it. This C++ program illustrates the use of object pointer-

```
/* C++ Pointers and Objects. Declaration and Use
 * of Pointers. This program demonstrates the
 * use of pointers in C++ */

#include<iostream.h>
#include<conio.h>
class Time
{
 short int hh, mm, ss;
 public:
```

```
 Time()
 {
 hh = mm = ss = 0;
 }
 void getdata(int i, int j, int k)
 {
 hh = i;
 mm = j;
 ss = k;
 }
 void prndata(void)
 {
 cout<<"\nTime is
"<<hh<<":"<<mm<<":"<<ss<<"\n";
 }
};
void main()
{
 clrscr();
 Time T1, *tptr;
 cout<<"Initializing data members using the object, with
values 12, 22, 11\n";
 T1.getdata(12,22,11);
 cout<<"Printing members using the object ";
T1.prndata();
 tptr = &T1;
 cout<<"Printing members using the object pointer ";
tptr->prndata();
 cout<<"\nInitializing data members using the object
pointer, with values 15, 10, 16\n";
 tptr->getdata(15, 10, 16);
 cout<<"printing members using the object ";
T1.prndata();
 cout<<"Printing members using the object pointer ";
tptr->prndata();
 getch();
}
```