Object Oriented Programming CS304

Lecture No 1

What is Object-Orientation?

It is a technique in which we visualize our programming problems in the form of objects and their interactions as happens in real life.

Object orientation makes it easier for us to solve our real world problems by thinking solution of the problem in terms of real world objects.

So we can say that in our daily life everything can be taken as an object that behaves in a certain way and has certain attributes.

What is a Model?

A model is an abstraction of something real or conceptual. We need models to understand an aspect of reality.

Model Examples

Highway maps - Architectural models - Mechanical models

In the context of programming models are used to understand the problem before starting developing it.

Object-Orientation - Advantages

As Object Oriented Models map directly to reality

We can easily **develop** an object oriented model for a problem. Everyone can easily **understand** an object oriented model. We can easily implement an object oriented model for a problem using any object oriented language like c++ using its features[1] like classes, inheritance, virtual functions and so on…

What is an Object?

An object is,

1. Something tangible (Ali, School, House, Car).
2. Something conceptual (that can be apprehended intellectually for example time, date and so on…).

An object has,

1. State (attributes)
2. Well-defined behavior (operations)
3. Unique identity

Lecture No 2

Information Hiding:

Information hiding is one of the most important principles of OOP inspired from real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner. By Information Hiding we mean "*Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world."*

In the perspective of **Object Oriented Programming** Information Hiding is,

"Hiding the object details (state and behavior) from the users"

Information Hiding is achieved in Object Oriented Programming using the following principles,

- All information related to an object is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself

We can achieve information hiding using **Encapsulation** and **Abstraction**, so we see these two concepts in detail now,

Encapsulation

Encapsulation means *"we have enclosed all the characteristics of an object in the object itself"*

Encapsulation and information hiding are much related concepts (information hiding is achieved using Encapsulation)

Advantages of Encapsulation

Simplicity and clarity

As all data and functions are stored in the objects so there is no data or function around in program that is not part of any object and is this way it becomes very easy to understand the purpose of each data member and function in an object.

Low complexity

As data members and functions are hidden in objects and each object has a specific behavior so there is less complexity in code there will be no such situations that a functions is using some other function and that functions is using some other function.

Better understanding

Everyone will be able to understand whole scenario by simple looking into object diagrams without any issue as each object has specific role and specific relation with other objects.

Interface:

Interface is a set of functions of an object that he wants to expose to other objects.

Implementation

It is actual implementation of the behavior of the object in any Object Oriented language.

Messages

Objects communicate through messages they send messages (stimuli) by invoking appropriate operations on the target object. The number and kind of messages that can be sent to an object depends upon its interface

Lecture No 3

Abstraction

Real life objects have a lot of attributes and many kinds of behaviors but most of the time we are interested in only that part of the objects that is related to the problem. this concept is called "*Abstraction*". Abstraction is a way to cope with complexity and it is used to simplify things.

Principle of abstraction:

"Capture only those details about an object that are relevant to current perspective"

Abstraction has following major advantages,

1. It helps us understanding and solving a problem using object oriented approach as it hides extra irrelevant details of objects.
2. Focusing on single perspective of an object provides us freedom to change implementation for other aspects of for an object later.

Inheritance

A child inherits characteristics of its parents, besides inherited characteristics, a child may have its own unique characteristics

Inheritance in Classes

If a class B inherits from class A then it contains all the characteristics (information structure and behaviour) of class A
The parent class is called *base* class and the child class is called *derived* class Besides inherited characteristics, derived class may have its own unique characteristics

Inheritance – "IS A" or "IS A KIND OF" Relationship

Inheritance – Advantages

1. Reuse 2. Less redundancy 3. Increased maintainability

Reuse with Inheritance

Main purpose of inheritance is reuse; we can easily add new classes by inheriting from existing classes. Select an existing class closer to the desired functionality, create a new class and inherit it from the selected class, add to and/or modify the inherited functionality

Lecture No 4

Generalization

In OO models, some classes may have common characteristics. We extract these features into a new class and inherit original classes from this new class. Base class encapsulates the idea of commonality of derived classes. Base class is general class representing common behavior of all derived classes. This concept is known as Generalization.

Sub-typing & Specialization

We want to add a new class to an existing model
We have developed an existing class hierarchy
Find an existing class that already implements some of the desired state and behavior
Inherit the new class from this class and add unique behavior to the new class

Sub-typing (Extension)

Sub-typing means that derived class is behaviourally compatible with the base class
Derived class has all the characteristics of base class plus some extra characteristics
Subtyping and generalization are related concepts, Subtyping (extension) and generalization is a way to look same thing in two ways. Sub typing is looking at things from Top to bottom whereas in generalization we look at things from bottom to top.

Specialization (Restriction)

We want to add a class to existing hierarchy of classes having many similarities to already existing classes but some part of its behaviour is different or restricted. In that case we will use the concept of specialization. Specialization means that derived class is behaviourally incompatible with the base class Behaviourally incompatibility means that base class can't always be replaced by the derived class Derived class has some different of restricted characteristics than of base class.
Add method behaviour is present in both base and derived classes but derived class behaviour is different in derived class. Derived class will not exhibit the behaviour of base class but it is **overriding** behaviour of base class with its own behaviour.

Overriding

A class may need to override the default behaviour provided by its base class Derived class overrides the behaviour of its base class. Reasons for overriding Provide behaviour specific to a derived class (specialization) Extend the default behaviour (extension) Restrict the default behaviour (restriction) Improve performance It is used for the implementation of inheritance.

Abstract Classes

In our examples we made classes for shape and person. These are abstract concepts and the classes we make against abstract concepts are called abstract classes. They are present at or near the top in the class hierarchy to present most generalized behaviour. An abstract class implements an abstract concept Main purpose is to be inherited by other classes Can't be instantiated Promotes reuse
Abstract Classes cannot exist standalone in an object model

Concrete Classes

The entities that actually we see in our real world are called concrete objects and classes made against these objects are called concrete classes.

A concrete class implements a concrete concept These are used to instantiate objects in our programs Provides implementation details specific to the domain context

A concrete class may exist in an object model independently

Concrete classes mostly lie below the top of class hierarchy in a good object model.

Lecture No 5

Multiple Inheritance

Sometimes we want to reuse characteristics of more than one parent class, in that case we need to inherit a class from more than one classes.

Advantage of Multiple Inheritance:

As was the case with simple (single) inheritance multiple inheritance also decreases redundant code as we can inherit a class from many classes and can use their functions without the need to write them again.

However, there are more disadvantages of multiple inheritance, than its advantages.

Problems with Multiple Inheritance

Increased complexity

Amphibious vehicle hierarchy is a complicated as this class is derived from two classes that will make code more complex and less understandable however this is obvious as amphibious vehicle is a complicated vehicle. It is generic problem.

Reduced understanding

Due to increased complexity of class hierarchy the object model becomes difficult it understand especially for someone who is looking it first time.

Duplicate features

As we are deriving a single class from more than one class so there is a chance of duplication of features (same methods in both parents), following problems may arise due to duplicate features,

Solution to Diamond Problem

Some languages disallow diamond hierarchy others provide mechanism to ignore characteristics from one side. There are two Cases while solving diamond problem virtual inheritance and non-virtual inheritance

Association:

Interaction of different objects in OO model (or in problem domain) is known as association.

In object oriented model, objects interact with each other in order to perform some useful work, while modeling these objects (entities) is done using the association. Usually an object provides services to several other objects. An object keeps association with other objects to delegate tasks. This association can be represented with a line along an arrow head ( ) or without arrow head.

Kinds of Association:

There are two main types of association which are then further subdivided i.e

1. Class Association 2. Object Association

1. Class Association

Class association is implemented in terms of Inheritance. Inheritance implements generalization/specialization relationship between objects. Inheritance is considered class association.

- In case of public inheritance it is “IS-A” relationship.
- In case of private inheritance it is “Implemented in terms of” relationship.

2. Object Association

It is the interaction of standalone objects of one class with other objects of anther class.

It can be of one of the following types,

• Simple Association • Composition • Aggregation

Simple Association

The two interacting objects have no intrinsic relationship with another object. It is the weakest link between objects. It is a reference by which one object can interact with some other object.

Simple association can be categorized in two ways,

• With respect to direction (navigation) • With respect to number of objects (cardinality)

Kinds of Simple Association w.r.t Navigation

a. One-way Association b. Two-way Association

a. One-way Association

In One-way association we can navigate along a single direction only, it is denoted by an arrow towards the server object.

b. Two-way Association

In two-way association we can navigate in both directions, it is denoted by a line between the associated objects

Kinds of Simple Association w.r.t Cardinality

a. Binary Association b. Ternary Association c. N-ary Association

a. Binary Association

It associates objects of exactly two classes; it is denoted by a line, or an arrow between the associated objects.

b. Ternary Association

It associates objects of exactly three classes; it is denoted by a diamond with lines connected to associated objects.

c. N-ary Association

An association between 3 or more classes its practical examples are very rare.

Composition

An object may be composed of other smaller objects, the relationship between the "part" objects and the "whole" object is known as Composition, and Composition is represented by a line with a filled-diamond head towards the composer object

Composition is stronger relationship:

Composition is a stronger relationship, because composed object becomes a part of the composer composed object can't exist independently

Aggregation

An object may contain a collection (aggregate) of other objects, the relationship between the container and the contained object is called aggregation, Aggregation is represented by a line with unfilled-diamond head towards the container

Aggregation is weaker relationship

Aggregation is weaker relationship, because• Aggregate object is not a part of the container• Aggregate object can exist independently

Lecture No 6

Class Compatibility

A class is behaviorally compatible with another if it supports all the operations of the other class. Such a class is called subtype. A class can be replaced by its subtype. Derived class is usually a subtype of the base class. It can handle all the legal messages (operations) of the base class. Therefore, base class can always be replaced by the derived class.

Polymorphism in general, polymorphism refers to existence of ***different forms*** of a single entity. For example, both Diamond and Coal are different forms of Carbon.

Polymorphism in OO Model in OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus). Consequently, sender of a message does not need to know exact class of the receiver.

Polymorphism – Advantages Messages can be interpreted in different ways depending upon the receiver class New classes can be added without changing the existing model In general, polymorphism is a powerful tool to develop flexible and reusable systems

Identify Classes **Extract nouns in the problem statement**

Finding Associations: **Identify Associations** Find relationships between objects,

Lecture No 7

The basic concept "**Object**" of *Object Orientation (thinking in terms of objects)* is realized using classes in programming languages.

Class: It is a way (Mechanism) given by c++ to realize objects in a program. It is concrete Implementation of objects in c++. We capture any object attributes and behaviour in a programming language using classes. In other words it can be defined as facility given by c++ to create new types according to our requirement. (Class is composite data type made from basic c++ types like integers, chars and float).

Uses: Objects are structured in terms of class so our problem becomes easier to understand in the terms c++ program. We can implement interactions easily in terms of classes.

Type in C++: We implement generic concepts using types. We have to model generic concept of Student. But there is no built in type **student** in c++ like built-in c++ type's **int** or **float**. Class is mechanism in c++ that will allow us to define student as user defined type, similarly generic concept circle will also be implemented in the same way.

Abstraction We only include those details in the system that are required for making a functional system so we will leave out irrelevant attributes and behaviour from our objects.

Defining a New User Defined Type There are two ways to create user defined types for objects in c++ these are,

Structure Definition: Partially we can use Structures to define an object. In c we cannot define functions in a structure however in c++ we can add functions in both structure and classes.

Why Member Functions: Objects can make their data invisible (in accordance with the principle of data hiding). Setters and getters functions are provided by class to access the its members it also minimizes the changes to move the objects in inconsistent state as we can write checks in our setter functions for example we can check that whether the user has entered correct age value and has not entered negative value for age. Object remains in consistent state.

Object and Class: Object is an instantiation of a user defined type or class. Once we have defined a class we can create as many objects for that class as we require.

Declaring class variables Variables of classes (objects) are declared just like variables of structures and built-in data types as follows,

TypeName VariableName;
int var; // declaring built in int data type variable
Student aStudent; // declaring user defined class Student object

Accessing members Members of an object can be accessed using,
a. dot operator (.) to access via the variable name
Student aStudent; // declaring Student object

aStudent. rollNo = 5; **// declaring and initializing Student pointer**
b. arrow operator (->) to access via a pointer to an object **aStudent->rollNo = 5;**
Student * aStudent = new Student();
Member functions are accessed in the similar way using dot or arrow operator.
Access specifies These are used to enforce access restrictions to members of a class, there are three access specifiers,
1. **'public'** is used to tell that member can be accessed whenever you have access to the object
2. **'private'** is used to tell that member can only be accessed from a member function
3. **'protected'** to be discussed when we cover inheritance
Default access specifier When no access specifier is mentioned then default access specifier is **private**.

Lecture No 8

Member Functions • Member functions are the functions that operate on the data encapsulated in the class
• Public member functions are the interface to the class
Defining Member Functions We can define member functions in two ways,
a. We can define member functions of the class inside the class definition when we define any class in our program.
OR
b. We declare member function inside the class definition and declare them outside the class.
In this case class definition is added in ClassName.h file and class implementation code is added in ClassName.cpp file.
Inline Functions

• Inline functions is a way used by compilers to improve efficiency of the program, when functions are declared inline normal process of function calling (using stack) is not followed instead function code is added by compiler at all points where these functions have been called. Basic concept behind inline functions is that they are functions in our code but in compiler generated files these functions code is added by compiler at all places where they were called in the code.

• Normally small size functions that need to be called many times during program execution are declared inline. Inline functions decrease the code execution time because program in their case doesn't involve function call overhead.

• Keyword 'inline' is used to request compiler to make a function inline.

• However using inline keyword with function doesn't guarantee that function will definitely in inlined, it depends on the compiler if it finds it can make function inline it does so otherwise it ignores the keyword inline and treat the function as normal function.

The functions defined inside the class are by default inline (whether we mention keyword inline with them or not)
In case we define function outside the class then we must use the keyword 'inline' to make the function inline.
However compiler decides whether it will implement these functions code as inline or not.
Constructor: Constructor is used to **initialize** the objects of a class. Constructor is used to ensure that object is in well-defined state at the time of creation. The constructor of a class is automatically generated by compiler however we can write it by our self also. Constructor is automatically called when the object is created. Constructors are not usually called explicitly by us.
Constructor Properties
• Constructor is a special function having same name as the class name
• Constructor does not have return type
• Constructors are commonly public members
Default Constructor
• Constructor without any parameter or with all parameters with default values is called default constructor
• If we do not define a default constructor the compiler will generate a default constructor
• Compiler generated default constructor is called **implicit** and user written default constructor is called **explicit**
• This compiler generated default constructor initialize the data members to their default values
• If we have given any constructor for a class whether it is
• our own **explcit default constructor** (i.e parameterless or with parameters having default values)
• our own constructor with parameters
Then compiler will not create implicit default constructor.
Constructor Overloading: We can write constructors with parameters as well. These parameters are used to initialize the data members with user supplied data (passed as parameter). The example is shown below, here example Student class has four constructors their prototypes are,
1. **Student();** /* explicit default parameterless constructor */
2. **Student(char * aName);** /* constructor with one parameter* /
3. **Student(char * aName, int aRollNo);** /* constructor with two parameters */
4. **Student(int aRollNo, int aRollNo, float aGPA);** /* constructor with three parameters */
Copy Constructor: Copy constructors are used when:
• Initializing an object at the time of creation (we want to create an object with state of a pre existing object)
• When an object is passed by value to a function (As you know temporary copy of object is created on stack so we need copy constructor to create that temporary object with the state of actual object being passed).
Shallow Copy

• When we initialize one object with another then the compiler copies state of one object to the other using copy constructor by assigning data member values of previous object to newly created object.

Deep Copy: We write our own deep copy code in copy constructor so that when we create new object from an existing object using copy constructor we also allocate new dynamic memory for data members involving dynamic memory.

Lecture No 9

Copy Constructor: Copy constructors are used when: Initializing an object at the time of creation (we want to create an object with state of a pre existing object)**.** When an object is passed by value to a function (As you know temporary copy of object is created on stack so we need copy constructor to create that temporary object with the state of actual object being passed).

As was the case with default constructor compiler also generates copy constructor by itself however we can override that copy constructor by writing our own copy constructor.

Shallow Copy: When we initialize one object with another then the compiler copies state of one object to the other using copy constructor by assigning data member values of previous object to newly created object. This kind of copying is called shallow copying. **Shallow copy using default Copy Constructor**

Shallow copy works fine if our class doesn't include dynamic memory allocation but in case of dynamic memory allocation it leads to dangling pointer problem as explained below.

Problem is Shallow Copy: Student class data member name of char * type is added to store the name of student and it is using **dynamic memory** according to the length of name entered by user for student.

Student class data member **name** (*char* *) of object studentB is also pointing to memory allocated for datamember **name** of object studentA, due to this there may be two kinds of problems.

Suppose we delete first object studentA for some reason then its destructor will also free memory allocated by it hence memory area containing name "AHMAD" will also be freed and will be given to some other application by operating system, but studentB member **name** is still pointing to that area so issue of *"Dangling Pointer"* [Pointer pointing to incorrect memory location] will arose. Same will happen if object studentB is deleted then studentA object data member name will become dangling pointer.

Secondly if for some reason we change name of object studentA the value of object studentB will also be changed as it pointing to same memory location.

Deep Copy: We write deep copy code in copy constructor so that when we create new object from an existing object using copy constructor we also allocate new dynamic memory for data members involving dynamic memory as shown below,

```
Student::Student( const Student & obj){
        int len = strlen(obj.name);
        name = new char[len+1]; // assignming new
/*dynamic memory to data member name of char * type for newly created object.*/
        strcpy(name, obj.name);
…
        //copy rest of the data members in the same way
}
```

Important points about copy constructor:

1. In case our class doesn't involve dynamic memory then default copy constructor that performs shallow copy works fine.
2. In case our class has any data member involving dynamic memory we have to write our own code in copy constructor to perform deep copy.
3. Copy constructor is normally used to perform deep copy
4. If we do not make a copy constructor then the compiler performs shallow copy
5. Shallow copy performs bitwise copy.

Destructor

1. Destructor is used to free memory that is allocated through dynamic allocation. We have to free memory allocated using new operator by over self in destructor otherwise it remain occupied even after our program ends.
2. Destructor is used to perform house keeping operations.
3. Destructor is a function with the same name as that of class, but preceded with a tilde '~'

Overloading: Destructors cannot be overloaded.

Sequence of Calls

Constructors and destructors are called automatically
Constructors are called in the sequence in which object is declared
Destructors are called in reverse order

Accessor Functions: In accordance to principle of information hiding data members of a class are declared as private so that outside world can not access the private data of the object only an interface is provided to outside world in the form of functions. Accessor functions are also used to access private data of the object, we provide accessor functions to get and set private data members of the class. We also add error checking code in accessor functions to reduce errors so that object doesn't move in illegal state.

Good Practice: Never return a handle to a data member from getter function because you are never sure that function accessing the reference will not change the value of the variable.

this Pointer: • Address of each object is passed to the calling function.
• This address is de-referenced by the functions and hence they act on correct objects

Passing this Pointer: • Whenever a function is called the this pointer is passed as a parameter to that function.
• Function with n parameters is actually called with n+1 parameters

Lecture No 10

Uses of this Pointer

o There are situations where designer wants to return reference to current object from a function
o In such cases reference is taken from this pointer like (*this)

```
Student Student::setRollNo(int aNo)
{
…
return *this;
}
Student Student::setName(char *aName)
{
…
return *this;
}
```

```
Usage:
int main()
{
Student aStudent;
Student bStudent;
bStudent = aStudent.setName("Ahmad");
…
bStudent = aStudent.setName("Ali").setRollNo(2);
return 0;
}
```

Separation of interface and implementation o Public member functions exposed by a class are called interface.
o Separation of implementation from the interface is good software engineering.

Benefits of separating interface and implementation: Consider the example of following complex no. class, this complex no. class two forms of implementations one is new and one is old implementation you can observe that if you have separated interface and implementation then we can easily change implementation without changing interface,

Complex Number: o There are two representations of complex number

• Euler form $z = x + i\,y$ • Phasor form $z = |z| (\cos\theta + i \sin\theta)$

Advantages: 1. User is only concerned about ways of accessing data (interface)
2. User has no concern about the internal representation and implementation of the class

Separation of interface and implementation: In c++ generally we can relate the concept of interface of a class to its header (.h) file and and implementation of a class to its (.cpp) file. However it is not complete separation of interface and implementation.

• Usually functions are defined in implementation file (.cpp) while the class definition is given in header file (.h)

const Member Functions: Some functions in our programs are general purpose functions to show or access data, they are supposed to do read only tasks only however there are chances that they can change the state of data members of the class while accessing the data members due to programming mistake, c++ provides the solution of this problem using constant member functions. We make those functions as constant who need only read only access (for example such functions that will only display data or will return the value of data members). When we make them constant compiler generates an error if these functions try to change the value of data members of the class.

const Member Functions: Keyword const is placed at the end of the parameter list to make any function as constant.

const Functions: Constructors and Destructors cannot be **const because** Constructors and destructors are used to modify the object to a well-defined state or to clean the memory occupied by the object.

const Function: • Constant member function cannot change data member
• We cannot call non constant functions in constant functions because non constant member functions may have code for changing state of the object that is not allowed in the constant functions.

This Pointer and const Member Function: As we know that when a class function is called an implicit this pointer is passed to tell the function about the object it has to operate same is true for constant function with the difference that it will bbe passed as constant pointer to const data in case of constant member functions so that this pointer can not be used now to change the value of data members of the object,

Lecture No 11

Usage example of Constant member functions Problem:

Suppose we have requirement to change the class Student such that a student is given a roll number when the object is created and cannot be changed afterwards our existing class

Solution of this problem:

We can do this by making rollNo constant so that cannot be changed once it is defined. Now there is only one issue of initializing this roll no with initial value but the problem is that we cannot set the value of roll no in constructor, as when code in constructor is executed the data member roll no has already been created and when we try to assign value to it in constructor compiler generates error. Second solution is to write separate function but the problem remains same that we can't assign value to constant data member

We also know that we can only declare data members in structure or class but we cannot initialize them at the time of declaration in structure or class _____ because before executing constructor code, the class const member roll no has not got life it

will get life along with other class members when constructor will be invoked so we cannot assign any value to this constant member while declaring it. 6

Solution: so what is the solution of this problem as we cannot initialize constant members while declaring them and we cannot initialize them in constructor also because as soon as they go life they become constant to solve this problem C++ gives us new mechanism (syntax) for initialization of constant data members of the structure or class to resolve above mentioned issues,

Difference between Initialization and Assignment:

Initialization is assigning value along with creation of variable. int i = 2;

Assignment is assigning value after creation. int i; i = 7;

Member Initializer List: Member initialization list is used where we cannot modify the state of data members in the member functions of the class including constructor,

- A member initializer list is a mechanism to initialize data members
- It is given after closing parenthesis of parameter list of constructor
- In case of more than one member use comma separated list

```
class Student{                                                    initialization
const int rollNo;                                                 …
char *name;                                                       }
float GPA;                                                        …
public:                                                           };
Student(int aRollNo) : rollNo(aRollNo), name(Null), GPA(0.0){ //
```

Order of Initialization

- Data member are initialized in order they are declared in the class
- Order in member initializer list is not significant at all

const Objects

- Objects can be declared constant with the use of const keyword
- Constant objects cannot change their state

const Objects: const objects can access only const member functions so chances of change of state of const objects once they are created are eliminated.

Constant member functions: Make all functions that don't change the state of the object constant

This will enable constant objects to access more member functions

Static Variables: Static variables of a class are such variables which are independent of class objects. Lifetime of static variable is throughout the program life, if static variables are not explicitly initialized then they are initialized to 0 of appropriate type.

Static Data Member

Definition

"A variable that is part of a class, yet is not part of any object of that class, is called static data member"

Static Data Member

They are shared by all instances (objects) of the class. They do not belong to any particular instance of a class

Class vs. Instance Variable

Static data member is declared inside the class

But they are defined outside the class

Static data members should be initialized once at file scope

They are initialized at the time of definition

If static data members are not explicitly initialized at the time of definition then they are initialized to 0

Lecture No 12

Static Data Member:

Definition

"A variable that is part of a class, yet is not part of an object of that class, is called static data member"

Static Data members are shared by all instances of the class and they do not belong to any particular instance of a class.

Memory for static variables in allocated in class space whereas for instance variables it is separate for each object.

Static Data Member (Syntax) Keyword static is used to make a data member static.

Defining Static Data Member (allocating memory for them): Static data member is declared inside the class But they are defined outside the class.

Initializing Static Data Member (assigning them some initial value): Static data members should be initialized once at file scope**.** They are initialized at the time of definition

If static data members are not explicitly initialized at the time of definition then they are initialized to 0.

Accessing Static Data Member: To access a static data member there are two ways

- Access like a normal data member (using dot operator '**.**')
- Access using a scope resolution operator '**::**'

Life of Static Data Member

- They are created even when there is no object of a class

• They remain in memory even when all Objects of a class are destroyed
Uses: They can be used to store information that is required by all objects, like global variables

Static Member Function

Definition:
"The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function"
• They are used to access static data members
• Access mechanism for static member functions is same as that of static data members
• They cannot access any non-static members

this Pointer and static member functions

• *this* pointer is passed implicitly to member functions
• *this* pointer is not passed to static member functions
• Reason is static member functions cannot access non static data members

Global Variable vs. Static Members

• Alternative to static member is to use global variable
• Global variables are accessible to all entities of the program
• User of Global variables is against the principle of information hiding.

Array of Objects

• Array of objects can only be created if an object can be created without supplying an explicit initializer
• There must always be a default constructor if we want to create array of objects

Lecture No 13

13.1.Pointer to Objects

• Pointer to objects are similar as pointer to built-in types
• They can also be used to dynamically allocate objects

Example

```
class Student{
…
public:
Student();
Student(char * aName);
void setRollNo(int aNo);
};
```

Example

```
int main(){
Student obj;
Student *ptr;
ptr = &obj;
ptr->setRollNo(10);
return 0;
}
```

Allocation with new Operator
• new operator can be used to create objects at runtime
Example

```
int main(){
Student *ptr;
ptr = new Student;
ptr->setRollNo(10);
return 0;
}
```

Example

```
int main(){
Student *ptr;
ptr = new Student("Ali");
ptr->setRollNo(10);
return 0;
}
```

Example

```
int main()
{
Student *ptr = new Student[100];
for(int i = 0; i < 100;i++)
{
ptr->setRollNo(10);
}
return 0;
}
```

13.2.Breakup of new Operation

new operator is decomposed as follows
• Allocating space in memory
• Calling the appropriate constructor

13.3.Case Study

Design a class date through which user must be able to perform following operations
• Get and set current day, month and year
• Increment by x number of days, months and year
• Set default date

Attributes

Attributes that can be seen in this problem statement are
• Day
• Month
• Year
• Default date

Attributes

The default date is a feature shared by all objects
• This attribute must be declared a static member
Attributes in Date.h

```
class Date
{
int day;
int month;
int year;
static Date defaultDate;
…
};
```

Interfaces

• getDay
• getMonth

• getYear
• setDay
• setMonth
• setYear
• addDay
• addMonth
• addYear
• setDefaultDate

Interfaces

As the default date is a static member the interface setDefaultDate should also be declared static

Interfaces in Date.h

```
class Date{
…
public:
void setDay(int aDay);
int getDay() const;
void addDay(int x);
…
};
```

Interfaces in Date.h

```
class Date{
…
public:
static void setDefaultDate(
int aDay,int aMonth, int aYear);
…
};
```

Constructors and Destructors in Date.h

```
Date(int aDay = 0,
int aMonth= 0, int aYear= 0);
~Date(); //Destructor
};
```

Implementation of Date Class

The static member variables must be initialized

```
Date Date::defaultDate (07,3,2005);
```

Constructors

```
Date::Date(int aDay, int aMonth,
int aYear) {
if(aDay==0) {
this->day = defaultDate.day;
}
else{
setDay(aDay);
}
//similarly for other members
}
```

Destructor

We are not required to do any house keeping chores in destructor

```
Date::~Date
{
}
```

Getter and Setter

```
void Date::setMonth(int a){
if(a > 0 && a <= 12){
month = a;
}
int getMonth() const{
return month;
}
```

addYear

```
void Date::addYear(int x){
year += x;
if(day == 29 && month == 2
&& !leapyear(year)){
day = 1;
month = 3;
}
}
```

Helper Function

```
class Date{
…
private:
bool leapYear(int x) const;
…
};
```

Helper Function

```
bool Date::leapYear(int x) const{
if((x%4 == 0 && x%100 != 0)
|| (x%400==0)){
return true;
}
return false;
}
```

setDefaultDate

```
void Date::setDefaultDate(
int d, int m, int y){
if(d >= 0 && d <= 31){
day = d;
}
…
}
```

Lecture No 14

Composition

If one object is part of another object (relationship of part and whole) in composition lifetime of one object depends upon the other. The part objects are essential components of the whole. For example person is **composed of** hands, eyes, feet so on.

In student class we assigning dynamic memory for variable name using new operator as shown,

```
class Student{
        private:
                float gpa;
        char * name;
                int rollNumber;
        public:
                Student(char * = NULL, int = 0, float =
        0.0);
                Student(const Student & st);
                const char * GetName() const;
        // never return handle of private data members or
        private member functions
                ~Student();
```

```
…
};
        Student::Student(char * _name, int roll, float g)
{
        cout << "Constructor::Student..\n";
if (!_name){
        name = new char[strlen(_name)+1];
        strcpy(name,_name);
}
        else name = NULL;
        rollNumber = roll;
        gpa = g;
}
        Student::Student(const Student & st){
        if(str.name != NULL){
        name = new char[strlen(st.name) + 1];
        strcpy(name, st.name);
}
        else name = NULL;
        rollNumber = st.roll;
        gpa = st.g;
}
        const char * Student::GetName(){
        return name;
}
// never return handle of private data members or private member functions const ensures that private data members will not be changed
        Student::~Student(){
        delete [] name; // deleting name array
}
```

In C++ "*it is all about code reuse"*

Composition is creating objects of one class inside another class

*"**Has a**"* relationship:

Bird has a beak

Student has a name

Composition

Now we change code slightly replacing name char * by String so that it is whole object of class String as it qualifies to be an object because we have to apply many operations on it like string dynamic creation and deletion, string copy using deep copy, searching a substring and so on….

Conceptual notation:

Composition: Now we see string class code to see how it simplifies original Student object and how we have used composition here,

```
        class String{
        private:
        char * ptr;
        public:
        String(); // default constructor
        String(const String &); // copy constructor
        void SetString(const char *); // setter function
        const char * GetString() const;
// getter function returning const pointer to data member ptr
~String()
…
};
        String::String(){
        cout << "Constructor::String..\n";
        ptr = NULL;
}
        String::String(const String & str){
        if(str.ptr != NULL){
        ptr = new char[strlen(str.ptr)+1];
        strcpy(ptr, str.ptr);
}
        else ptr = NULL;
}
        void String::SetString(const char * str){
        if(ptr != NULL){
        delete [] ptr;
        ptr = NULL;
}
        if(str != NULL){
        ptr = new char[strlen(str)+1];
        strcpy(ptr, str);
}}
```

issue of memory leakage (inaccessible memory) if we simply set pointer here memory will be outside of our object many cause problems later user still has pointer of passed value it can itself modify it We resolve these two issues in SetString by allocating new memory and deleting previous memory.

```
        const char * String::GetString()const{
        return ptr;
}
        String::~String(){
        delete [] ptr;
        cout <<"Destructor::String..\n";
}
```

Make pointer equal to NULL as well any where you delete dynamic memory. Now consider the code of Student class again, now by adding composed string object our code has been simplified very much, (we will use methods of composed object simply by calling them where needed)

```
class Student{
        private:
        float gpa;
        int rollNumber;
String name;
        public:
        Student(char* =NULL, int=0,float=0.0);
        Student(const Student &);
        void SetName(const char *);
        String GetName() const;
        const char * GetNamePtr() const;
        ~Student();
…
};
        Student ::Student(char * _name, int roll, float g){
        cout <<"Constructor::Student..\n";
        name.SetString(_name);
        rollNumber = roll;
        gpa = g;
}
```

```
Student::Student(const Student & s){
name.SetString(s.name.GetString());
gpa = s.gpa;
rollNumber = s.rollNumber;
}
```

Explanation:

1. **name.SetString**(s.name.GetString());
// setting composed name of newly created object
2. name.SetString(**s.name**.GetString());
//accessing the composed object string name of object to be copied
3. name.SetString(s.**name.GetString**());
//accessing the value of composed object string name by calling its member
function GetString
4. **name.SetString(s.name.GetString());**
//overall result : the value of composed object string of object to be copied will be copied to newly created object composed object string.

```
const char * Student::GetNamePtr() const{
return name.GetString();
}
void Student::SetName(const char * n){
name.SetString(n);
}
Student::~Student(){
cout <<"Destructor::Student..\n";
}
void main(){
Student *aStudent=new Student("Fakhir", 899, 3.1);
cout << endl;
cout << "Name:" << aStudent->GetNamePtr() << "\n";
}
```

Output:

```
Constructor::String..
Constructor::Student..
Name: Fakhir
Destructor::Student..
Destructor::String..
```

Important Points:

1. We can access methods of composed object in the same way as we can access methods of other objects.
Name of composed object.MemberFunction
2. Member functions of a class can access its private data members like,

```
Student::Student(const Student & s){
name.SetString(s.name.GetString());
```

// accessing private member **String name** of student using its object **s**
and then accessing **String name** member function **GetString** to access string value two methods calss in one line

```
gpa = s.gpa;
```

// accessing private members of student in student member function

```
rollNo = s.rollNo;
```

// accessing private members of student in student member function

```
}
```

Constructors & Composition

Constructors of the sub-objects are always executed before the constructors of the master class

Example:

As you see the example output of program given above,

Output:

```
Constructor::String..
Constructor::Student..
Name: Fakhir
Destructor::Student..
Destructor::String..
```

Constructor for the sub-object name is executed before the constructor of Student and destructor of sub-object is called after destructor of student. It is logical as composing object has to contain composed object so composed object should be created first and then composing object. Similarly while destructing objects we composing object is destructed first and then composed object as shown in diagram below,

Constructor calling: Constructors are called from composed objects to composing objects.
Destructor calling: Destructors are called from composing objects to composed objects.

Lecture No 15

Aggregation: In composition we made separate object of those concepts that we think were worthy to be implemented as an object within other object to make our code simpler and to make functionality modular (divided in parts) and understandable like we made String class in Student class, but in real life most situations are such that two distinct (different) objects and one object is using services of the other one like student and teacher, student and librarian, room and chair, passenger and bus, book and bookshelf, person and computer so on. In this case we cannot make one object as part of other object because they exist independently and only provide services to each other like in case of,

Student and Teacher: Student or Teacher object cannot be composed of other one yet they are taking services of each other.
Passenger and Bus: Passenger and Bus are taking services of each other but exist standalone also, bus includes passengers but passenger are not Part of Bus they can exist independently as well.

Composition vs. Aggregation

Aggregation is a *weak relationship* than composition because in this relationship two classes get services of each other but can exist independently as well, main difference is memory organization of two objects as shown below,

Aggregation C++ implementation:

In aggregation, a pointer or reference to an object is created inside a class. The subobject has a life that is **NOT** dependant on the life of its master class.

e.g
- Chairs can be moved inside or outside at anytime
- When Room is destroyed, the chairs may or **may not** be destroyed

Aggregation:

```
class Room{
private:
float area;
Chair * chairs[50];
Public:
Room();
void AddChair(Chair *, int chairNo);
Chair * GetChair(int chairNo);
bool FoldChair(int chairNo);
…
};
Room::Room(){
for(int i = 0; i < 50; i++)
chairs[i] = NULL;
void Room::AddChair(Chair * chair1, int chairNo){
if(chairNo >= 0 && chairNo < 50)
chairs[chairNo] = chair1;
}
Chair * Room::GetChair(int chairNo){
if(chairNo >= 0 && chairNo < 50)
return chairs[chairNo];
else
return NULL;
}
bool Room::FoldChair(int chairNo){
if(chairNo >= 0 && chairNo < 50)
return chairs[chairNo]->FoldChair();
else
return false;
}
int main(){
Chair ch1;
{
Room r1;
r1.AddChair(&ch1, 1);
r1.FoldChair(1);
}
ch1.UnFoldChair(1);
return 0;
}
```

Friend Functions: The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.

Why they are needed?: They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example, Suppose we wrote the code to compute a complex mathematical formulae in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

Are friend functions against the concept of Object Oriented Programming?:

It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

In order to access the member variables of the class, we must make function friend of that class

Friend Functions Prototypes of friend functions appear in the class definition. But friend functions are NOT member functions.

Friend functions can be placed anywhere in the class without any effect Access specifiers don't affect friend functions or classes

If keyword **friend** is used in the function definition, it's a syntax error

//Error… **friend void DoSomething(X obj){**

Similarly, one class can also be made friend of another class:

Lecture No 16

Operator overloading

Using operator overloading we can perform basic operations (like addition, subtraction, multiplication, division and so on…) on our own defined classes objects in the similar way as we perform them on basic built-in types(like int, float, long, double etc.).

C++ allows us to overload common operators like **+**, **-** or ***** etc… With operator overloading Mathematical statements don't have to be explicitly converted into function calls as we had to do to add two complex no objects using function call **Add**.

Assume that operator **+** has been overloaded then actual C++ code becomes:

c1+c2+c3+c4 The resultant code is very easy to read, write and maintain

C++ automatically overloads operators for pre-defined types as these have also been implemented as classes by c++.

Example of predefined types: **int** **float** **double** **char** **long**

The compiler probably calls the correct overloaded low level function for addition

Operator functions are not usually called directly, they are automatically invoked to evaluate the operations they implement by compiler.

List of operators that can be overloaded in C++:

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

List of operators that can't be overloaded:

```
.       .*      ::      ?:      #       ##
```

Reason: They take actual current object name, rather than value in their argument as you have seen previously in the use of dot (**'.'**) operator,

Student std;

int roll = std.getRollNo() // dot operator is performing on actual function (getRollNo) of class Student that will vary from program to program. **?:** is the only ternary operator in C++ and can't be overloaded.

The precedence of an operator:

The precedence of an operator is order of evaluation which operator will be evaluated first in expression. The precedence of an operator is **NOT** affected due to overloading.

Associativity: Associativity is **NOT** changed due to overloading Following arithmetic expression always is evaluated from left to right: **c1 + c2 + c3 + c4**

Unary operators and assignment operator are right associative, e.g: **a=b=c** is same as **a=(b=c)**

All other operators are left associative: **c1+c2+c3** is same as **(c1+c2)+c3**

Important things to consider: Always write code representing the operator for example adding subtraction code inside the + operator will create chaos. Creating a new operator is a syntax error (whether unary, binary or ternary), you cannot create $.

Arity of Operators: Arity (no of operands it works on) of an operator is NOT affected by overloading

Example: Division operator will take exactly two operands in any case: b = c / d

General syntax of Operators Overloading: In case of member functions of a class: return_type **class_name::operator** operator_symbol(parameters){ /*code*/

In case of non member functions of a class (in this case we will make overloaded operator function as friend function): return_type **operator** operator_symbol(parameters){ /*code*/

Binary Operators Overloading: Binary operators act on two quantities. Examples of binary operators

```
+     -     *     /     %     ^     &     |     ~
!     =     <     >     +=    -=    *=    /=    %=
^=    &=    |=    <<    >>    >>=   <<=   ==    !=
<=    >=    &&    ||    ,     ->*   ->
```

General syntax of Binary Operators Overloading:

In case of member function of a class: TYPE **class_name::operator** operator_symbol(TYPE rhs){ /*code*/

In case of non-member function of a class:

TYPE **class_name::operator** operator_symbol(TYPE rhs1, TYPE rhs2){ /*code*/

The "**operator OP**" must have at least one formal parameter of type class (user defined type)

Drawbacks of void return type:

Assignments and cascaded expressions are not possible

One of the existing objects is used to store result- Code is less readable-Debugging is tough- Code is very hard to maintain

Lecture No 17

Binary operators (cont.): The binary operator is always called with reference to the left hand argument.

The overloading code we discussed before for complex no. class can add two complex number objects but it cannot handle the following situation:

Complex c1;

c1 + 2.325 (as we are adding basic data type double to complex no.)

To do this, we have to modify the **Complex** class.

Modifying the complex class:

```
class Complex{
...
Complex operator+(const Complex & rhs);
Complex operator+(const double& rhs);
};
Complex operator + (const double& rhs){
Complex t;
t.real = real + rhs;
t.img = img;
return t;
}
```

Now we can write both forms of statements,

```
Complex c2, c3;
Complex c1 = c2 + c3;
Complex c4 = c2 + 235.01;
```

But problem arises if we do the following:

```
Complex c5 = 450.120 + c1;
```

The + operator is called with reference to **450.120**

No predefined overloaded + operator is there that takes **Complex** as an argument Now if we write the following two functions to the class, we can add a **Complex** to a **real** or vice versa :

```
friend Complex operator + (const Complex & lhs, const double & rhs);
friend Complex operator + (const double & lhs, const Complex & rhs);
};
```

We have made them as friend so that we can write them as nonmember functions and they are not called with respect to complex no. class object instead we pass both arguments (complex no. object and double value) to this function compiler invoke them according to arguments passed. Their implementation is similar,

```
Complex operator +(const Complex & lhs, const double& rhs){
Complex t;
t.real = lhs.real + rhs;
t.img = lhs.img;
return t;
}
```

```
Complex operator + (const double & lhs, const Complex & rhs){
Complex t;
t.real = lhs + rhs.real;
t.img = rhs.img;
return t;
}
```

So adding three overloaded versions for + operator as shown below allow us to write code to,
Add two complex objects
Add complex object and a double value.

```
Class Complex{
…
Complex operator + (const Complex &);
friend Complex operator + (const Complex &, const double &);
friend Complex operator + (const double &, const Complex &);
};
```

Nonmembers which are not friend call also achieve this functionality but in that case we need extra four functions two getters and two setters each for real and imag part. Compiler searches overloaded operator code in member function first and then in Nonmember functions.

Other binary operators are overloaded in similar to the + operator as demonstrated in the above examples As we know compiler can generate the following three functions for a class if required on its own,

- Default Constructor (in case we have not written any other constructor for a class)
- Copy Constructor
- Assignment Operator

However as we discussed previously if our class has any data member using dynamic memory then we had to write our own code for default constructor, copy constructor and similarly assignment operator as compiler generated version of these functions performs shallow copy that creates dangling pointer, and memory leakage issues in case of dynamic memory allocation. We have already seen code of default constructor and copy constructor and the code for overloaded assignment operator is similar,

```
String(); // default constructor
String(char *); // overloaded constructor
String(const String &); // copy constructor
```

Modified Assignment Operator Code:

```
class String{
…
public:
…
void operator =(const String &);
};
void String::operator = (const String & rhs){
size = rhs.size;
if(rhs.size != 0){
delete [] bufferPtr; // resolving inaccessible memory issue
bufferPtr = new char[rhs.size+1]; // creating new dynamic memory
strcpy(bufferPtr,rhs.bufferPtr); // deep copy
}
else
bufferPtr = NULL;
}
int main(){
String str1("ABC");
String str2("DE"), str3("FG");
str1 = str2; // Valid…
str1 = str2 = str3; // Error…
return 0;
}
```

The problem in statement

str1 = str2 = str3 is,
str1=str2=str3 is resolved as:
str1.operator=(str2.operator=(str3))
Assignment operator is beiing called two times one for part str2 = str3 and then for str1 = (str2 = str3) as assignment operator is right associate so first **str2=str3** will be executed, and str2 will become equal to str3, then first overloaded assignment operator execution result will be assigned to s1, str1.operator=(**str2.operator=(str3)**)
Problem is when compiler will try to invoke second assignment operator to assign value to str1 error will be returned becuase of void return type of overloaded assignment operator the reasn is explained below,
As we have studied before values are passed as parameters in operator overloadnig, str2 = str3 means str2.operator = (str3)
// str3 is being called as parameter and str1 = str2 = str3 means str3.operator(str2.operator = (str3))
// str2.operator = (str3) is being passed as parameter
This issue can be resolved by introducing minor change in our code of copy assignment operator to make it return String object instead of void as shown below,

```
…
public:
…
```

```
String & operator = (const String &);
};
String & String :: operator = (const String & rhs){
size = rhs.size;
delete [] bufferPtr;
if(rhs.size != 0){
bufferPtr = new char[rhs.size+1];
strcpy(bufferPtr,rhs.bufferPtr);
}
else bufferPtr = NULL;
return *this;
}
```

Now we are returning the value by reference of the object with respect to which this overloaded assignment operator will be called. It will be str2 in the case of str2 = str3, now when part **str1 = (**str2 = str3**)** will be executed, str2 will be passed as argument, that will be assigned to str1.

Lecture No 18

Self-assignment problem:

```
int main(){
String str1("Fakhir");
str1 = str1; // Self Assignment problem…
return 0;
}
```

Result of str1 = str1

We can resolve this issue by adding a simple if condition to ensure that both strings are not same

```
String & String :: operator = (const String & rhs){
if(this != &rhs){
size = rhs.size;
delete [] bufferPtr; // deleting memory of left hand side operand
if(rhs.bufferPtr != NULL){
bufferPtr = new char[rhs.size+1];
strcpy(bufferPtr,rhs.bufferPtr);
// memory access violation or incorrect data copy
}
else bufferPtr = NULL;
}
return *this;
}
```

Now self-assignment is properly handled:

```
int main(){
String str1("Fakhir");
str1 = str1;
return 0;
}
```

We can make return type String & as constant to avoid assignment to sub expressions, like (str1 = str2) = str3

```
…
public:
…
const String & operator=
(const String &);
};
int main(){
String s1(“ABC”),
s2(“DEF”),
s3(“GHI”);
// Error…
(s1 = s2) = s3;
return 0;
}
```

But as we can do that with primitive types so we can allow assignment to sub expressions by making return type as String & only as we have done before.

```
int main(){
int a, b, c;
(a = b) = c;
return 0;
}
```

Other Binary operators

Overloading += operator:

```
class Complex{
double real, img;
public:
Complex & operator+=(const Complex & rhs);
Complex & operator+=(const double & rhs);
...
};
Complex & Complex::operator += (const Complex & rhs){
real = real + rhs.real;
img = img + rhs.img;
return * this;
}
Complex & Complex::operator += (const double & rhs){
real = real + rhs;
return * this;
}
int main(){
Complex c1, c2, c3;
c1 += c2;
c3 += 0.087;
return 0;
}
```

Friend Functions and Operator overloading

Friend functions minimize encapsulation as we can access private data of any class using friend functions,

This can result in:

- Data vulnerability
- Programming bugs
- Tough debugging

Hence, use of friend functions must be limited we can overload operators without declaring them friend functions of a class, for example the + operator can be defined as a non-member, non-friend function as shown below,

(Three versions of overloaded + operator for Complex no class to handle three kinds of statements)

- obj1 + obj2

o + obj1

- obj1 + 3.78

```
Complex operator + (const Complex & a, const Complex & b){
Complex t = a; // creating temporary object t to store a+b
return t += b; // returning t by reference
}
```

```
Complex operator + (const double & a, const Complex & b){
Complex t = b;
return t += a;
}
Complex operator + (const Complex & a, const double & b){
Complex t = a;
return t += b;
}
```

Other Binary operators

The operators

-=, /=, *=, |=, %=, &=, ^=, <<=, >>=, !=

can be overloaded in a very similar fashion.

Lecture No 19

Stream Insertion operator

Often we need to display the data on the screen c++ provides us insertion operator ('<<')to put data on output stream default is console but it can be any file or network socket as well to send data on network from our program.

Example:

```
int i=1, j=2;
cout << "i= "<< i << "\n";
cout << "j= "<< j << "\n";
```

Stream Extraction operator

We also need to get data from the console or from file or network this is achieved through c++ provided stream extraction operator ('>>') that is used to get data from input stream, again default input stream is from console.

Example:

```
int i,j;          cin >> i >> j; // getting value of i and j from user
```

Explanation:

cin and **cout** are objects of istream and ostream classes used for input and output respectively, the insertion and extractions operators have been overloaded in istream and ostream classes to do these tasks. When we write lines like,

```
int i;
cin>> i;
cout << i;
```

Actually we are using istream and ostream class objects and using these objects we are calling these classes overloaded (>> and <<) operators that have been overloaded for all basic types like integer, float , long , double and char *. We have seen previously that actual call of overloaded operators for a class takes place by passing the objects to overloaded function as parameter like shown below,

```
cin>>i;
```

*istream & operator >> **(istream & in, int & i)***

Here cin will be passed as istream object along with int i to be displayed and code of this functions is returning istream object by reference & to accommodate multiple input statement in a single line like,

```
int i , j;
cin>>i >> j;
```

as we did in our string class before.

Same goes for insertion operator <<

```
cout << i;
```

ostream & operator >> (ostream & os, const int & i)

Stream insertion and extraction operator have been overloaded for basic data types but if we try to use them for user defined data types like our Complex no. class compiler will generate error as it will not find any overloaded operator code for our complex no class

```
Complex c1;
cout << c1; // Error
cout << c1 << 2; // Error cascaded statement
```

// Compiler error: binary '<<' : no operator // defined which takes a right-hand operand of type 'class Complex'

Same error will be for stream extraction operator so will need to overload these two operators (<< and >>) for our Complex no. class.

Stream Insertion Operator

First we try to overload insertion << operator as member function as we did before,

```
class Complex{
…
public:
…
void operator << (const Complex & rhs);
};
```

But when we will call this overloaded function is main compiler will generate errors as shown below,

```
int main(){
Complex c1;
cout << c1; // Error
c1 << cout;
c1 << cout << 2; // Error
return 0;
};
class Complex{
…
public:
…
```

```
void operator << (ostream &);
};
void Complex::operator << (ostream & os){
os << '(' << real
<< ',' << img << ')';
}
```

Now the statement c1 << cout will work but it has two limitations, Difficult to understand and remember statement syntax (c1 << cout ;) Cascaded statements not possible (cout << c1 << 2 ;) Better syntax is given below to resolve these two issues,

```
class Complex{
...
friend ostream & operator << (ostream & os, const
Complex & c);
};
Stream Insertion operator
// we want the output as: (real, img)
ostream & operator << (ostream & os, const Complex
& c){
os << '(' << c.real
<< ','
<< c.img << ')';
return os;
}
```

ostream reference can not be const as it store the data in its buffer to insert on output stream, however Complex reference will be constant as we are only getting data from Complex object and inserting it to output stream.

```
Complex c1(1.01, 20.1), c2(0.01, 12.0);
cout << c1 << endl << c2;
```

Stream Insertion operator

Output:

(1.01 , 20.1)

(0.01 , 12.0)

Now cascading statements are also possible as given below,

```
cout << c1 << c2;
```

is equivalent to

operator<<(operator<<(cout,c1),c2);

Because insertion operator is Left to right associative so first left part **cout << c1** << c2 will be executed and then the next part as opposed to copy assignment operator that will right associative. Same thing can be done with stream extraction operator,

Overloading Stream Extraction Operator:

```
class Complex{
...
friend istream & operator >> (istream & i, Complex & c);
};
```

istream can not be cont and istream buffer will change as we will get data from it and assign it to complex reference similarly Complex object can not be const for stream extraction operator as well because we will add data to it and hence its state will change.

Stream Extraction Operator Code:

```
istream & operator << (istream & in, Complex & c){
in >> c.real;
in >> c.img;
return in;
}
Main Program:
Complex c1(1.01, 20.1);
cin >> c1;
// suppose we entered // 1.0025 for c1.real and // 0.0241 for
c1.img
cout << c1;
Output:
( 1.0025 , 0.0241 )
```

Other Binary operators:

Overloading comparison operators (Equality and Inequality operators)

```
class Complex{
public:
bool operator == (const Complex & c);
//friend bool operator == (const //Complex & c1, const
Complex & c2);
bool operator != (const Complex & c);
//friend bool operator != (const //Complex & c1, const
Complex & c2);
…
};
```

Equality operator:

```
bool Complex::operator ==(const Complex & c){
if((real == c.real) &&
(img == c.img)){
return true;
}
else
return false;
}
```

As non member friend function:

```
bool operator ==(const Complex& lhs, const Complex&
rhs){
if((lhs.real == rhs.real) &&
(lhs.img == rhs.img)){
return true;
}
else
return false;
}
```

Inequality Operator:

```
bool Complex::operator !=(const Complex & c){
if((real != c.real) ||
(img != c.img)){
return true;
}
else
```

return false;

Lecture No 20

Modified String Class:

We have seen the following string class till now,

```
class String{
private:
char * bufferPtr;
int size;
public:
String();
String(char * ptr);
void SetString(char * ptr);
const char * GetString();
...
};
int main(){
String str1("Test");
String str2;
str2.SetString("Ping");
return 0;
}
```

What if we want to change the string from "*Ping*" to "*Pong*"?? *{ONLY 1 character to be changed…}*

Possible solution:

- Call: **str2.SetString("Pong");**
- This will delete the current buffer and allocate a new one
- Too much overhead if string is too big

Or, we can add a function which changes a character at nth location

```
class String{
...
public:
void SetChar(char c, int pos);
...
};
void SetChar(char c, int pos){
if(bufferPtr != NULL){
if(pos>0 && pos<=size)
bufferPtr[pos] = c;
}
}
```

Other Binary Operators

Now we can efficiently change a single character:

```
String str1("Ping");
str1.SetChar('o', 2);
// str1 is now changed to "Pong"
```

Subscript [] Operator

There is another elegant solution present for this problem. It is subscript operator that is used on basic char [] data type as shown below,

```
char array[5] = "Ping";
array[1] = 'o';
```

We want to get same functionality for our own defined String class for this we overload the subscript **"[]"** operator. We want function like given below for subscript operator in our String class,

```
int main(){
String str2;
str2.SetString("Ping");
}

str[2] = 'o';
// acting as l-value (left value so that we can assign it some value)
cout << str[2];
// acting as r-value (we are reading value using subscript operator)
return 0;
}
```

Overloading Subscript [] Operator

Subscript operator must be overloaded as member function of the class with one parameter of integer type,

```
class String{
...
public:
char & operator[](int);
...
};
char & String::operator[]( int pos){
assert(pos>0 && pos<=size);
return stringPtr[pos-1];
}
int main() {
String s1("Ping");
cout <<str.GetString()<< endl;
s1[2] = 'o';
cout << str.GetString();
return 0;
}
```

Output: **Ping** **Pong**

Overloading Function () operator

Must be a member function

Any number of parameters can be specified

Any return type can be specified

Operator() can perform any generic operation

Function Operator

```
class String{
...
public:
char & operator()(int);
...
};
char & String::operator()
(int pos){
assert(pos>0 && pos<=size);
return bufferPtr[pos-1];
}
int main(){
String s1("Ping");
char g = s1(2); // g = 'i'
s1(2) = 'o';
cout << g << "\n";
cout << str.GetString();
return 0;
}
```

Output: **I** **Pong**

Function Operator performing Sub String operation,

```
class String{
```

```
...
public:
String operator()(int, int);
...
};
String String::operator()(int index, int subLength){
assert(index>0 && index+subLength-1<=size);
char * ptr = new char[subLength+1];
for (int i=0; i < subLength; ++i)
ptr[i] = bufferPtr[i+index-1];
ptr[subLength] = '\0';
String str(ptr);
delete [] ptr;
return str;
}
int main(){
String s("Hello World");
// "<<" is overloaded
cout << s(1, 5);
return 0;
}
```

Function Operator

Output: **Hello**

Unary Operators

Unary operators take one operand, they act on the object with reference to which they have been called as shown below,

& * + - ++ -- ! ~

Examples:

- **--x**
- **-(x++)**
- **!(*ptr ++)**

Lecture No 21

Unary Operators

Behavior of ++ and -- for pre-defined types:

- Post-increment **++:**

Post-increment operator ++ increments the current value and then returns the previous value

- Post-decrement --: Works exactly like post ++

Example:

```
int x = 1, y = 2;
cout << y++ << endl;
cout << y;
```

Output:

2
3

Example:

```
int y = 2;
y++++; // Error
y++ = x; // Error
```

Behavior of ++ and -- for pre-defined types:

- Pre-increment ++:

Pre-increment operator ++ increments the current value and then returns it's reference

- Pre-decrement --:

Works exactly like Pre-increment ++

Example:

```
int y = 2;
cout << ++y << endl;
cout << y << endl;
```

Unary Operators

Unary operators are usually prefix, except for ++ and --

++ and -- both act as prefix and postfix

Example:

```
h++;
g-- + ++h - --i;
```

General syntax for unary operators

As Member Functions:

```
TYPE & operator OP (); // no argument the object with respect to which it is called is taken as one operand
```

As Non-member Functions:

```
Friend TYPE & operator OP (TYPE & t);
// one argument object with respect to which it is called.
```

Overloading unary '-':

```
class Complex{
...
Complex operator - ( );
// friend Complex operator -(Complex &);
};
Complex Complex::operator -(){
Complex temp;
temp.real = -real;
temp.img = -img;
return temp;
}
Complex c1(1.0 , 2.0), c2;
c2 = -c1;
// c2.real = -1.0
// c2.img = -2.0
```

Unary '+' is overloaded in the same way.

Output:

3
3

Example:

```
int x = 2, y = 2;
++++y;
cout << y;
++y = x;
cout << y;
```

Output:

4
2

Example (Pre-increment):

```
class Complex{
double real, img;
public:
...
Complex & operator ++ ();
// friend Complex & operator ++(Complex &);
};
Complex & Complex::operator++(){ // member function
real = real + 1;
return * this;
}
Complex & operator ++ (Complex & h){ // non member function
```

```
h.real += 1;
return h;
}
```

How does a compiler know whether it is a pre-increment or a post-increment?

A post-fix unary operator is implemented using:
Member function with 1 dummy int argument
OR
Non-member function with two arguments
In post increment, current value of the object is stored in a temporary variable Current object is incremented
Value of the temporary variable is returned

Post-increment operator:

```
class Complex{
...
Complex operator ++ (int);
// friend Complex operator ++(const Complex &, int);
};
Complex Complex::operator ++ (int){
complex t = *this;
real += 1;
return t;
}
Complex operator ++ (const
Complex & h, int){
complex t = h;
h.real += 1;
return t;
}
```

How does a compiler know whether it is a pre-increment or a post-increment?

The dummy parameter in the operator function tells compiler that it is postincrement
Example:

```
Complex h1, h2, h3;
h1++;
h3++ = h2 + h3++; // Error…
```

The *pre* and *post* decrement operator -- is implemented in exactly the same way

21.3.Type Conversion

The compiler automatically performs a type coercion of compatible types
e.g:

```
int f = 0.021;
double g = 34;
```

// type **float** *is automatically converted into* **int**. *Compiler only issues a warning…*

Type Conversion

The user can also explicitly convert between types:

```
int g = (int)0.0210;
double h = double(35);
```

// type **float** *is explicitly converted (casted) into* **int**. *Not even a warning*
// is issued now…

Type Conversion

For user defined classes, there are two types of conversions
From any other type to current type
From current type to any other type
We can do the following type conversion by our self,

Conversion from any other type to current type:
Requires a constructor with a single parameter
For example,

```
String str = 135;
```

Conversion from current type to any other type:
Requires an overloaded operator
For example,

```
String str;
char * ptr = str;
```

Conversion from other type to current type (int to String):

```
class String{
...
public:
String(int a);
char * GetStringPtr()const;
};
String::String(int a){
cout << "String(int) called..." << endl;
char array[15];
itoa(a, array, 10);
size = strlen(array);
bufferPtr = new char [size + 1];
strcpy(bufferPtr, array);
}
char * String::GetStringPtr() const{
return bufferPtr;
}
int main(){
String s = 345;
cout << s.GetStringPtr() << endl;
return 0;
}
```

Output:

```
String(int) called…
345
```

Automatic conversion like shown above using constructor has drawbacks conversion takes place transparently even if the user didn't wanted the conversion for example see the code below, in it user can write the following code to initialize the string with a single character:

```
int main(){
String s = 'A';
// ASCII value of A that is 65 is being taken automatically
it may be what the user // wants, perhaps user intent was
String s = "A" to store A in the string object
// but instead 65 is being stored
cout << s.GetStringPtr()<< endl << s.GetSize() << endl;
return 0;
}
Output:
String(int) called…
65
2
```

Keyword explicit

There is a mechanism in C++ to restrict automatic conversions like this using constructor code it is to use keyword **explicit** with such constructors if we have used

this keyword then casting must be explicitly performed by the user. Keyword **explicit** only works with constructors.
Example:

```
class String{
…
public:
…
explicit String(int);
};
int main(){
String s;
s = 'A'; // Error…
return 0;
}
int main(){
String s1, s2;
s1 = String(101); // valid, explicit casting…
// OR
s2 = (String)204;
return 0;
}
```

Type Conversion

There is another method for type conversion:
"Operator overloading"
It is used for converting from current type (user defined) to any other basic type or user defined type.
General Syntax:
TYPE$_1$::operator TYPE$_2$();
Like,
String::operator char * (); // TYPE1 = String , TYPE2 = char *
It will convert our string object to char *.
We write such functions as member function of the class.
NO return type and arguments are specified
Return type is implicitly taken to be TYPE2 by compiler

Type Conversion

Overloading pre-defined types:

```
class String{
…
public:
…
operator int();
operator char *();
};
String::operator int(){
if(size > 0)
return atoi(bufferPtr);
else
return -1;
}
String::operator char *(){
return bufferPtr;
}
int main(){
String s("2324");
cout << (int)s << endl << (char *)s;
// int a = (int)s;
// int a = int (s);
return 0;
}
```

Output:
2324
2324

User Defined types:

User-defined types can be overloaded in exactly the same way
Only prototype is shown below:

```
class String{
…
operator Complex();
operator HugeInt();
operator IntVector();
```

Drawbacks of Type Conversion Operator:

```
class String{
…
public:
…
String(char *);
operator int();
};
int main(){
String s("Fakhir");
// << is NOT overloaded
cout << s; // compiler is automatically converting s to int
return 0;
}
```

Output:

Junk Returned…

To avoid this problem DO NOT use *type conversion operators* instead use separate member function for such type conversion as shown below,
Modifying String class:

```
class String{
…
public:
…
String(char *);
int AsInt();
};
int String::AsInt(){
if(size > 0)
return atoi(bufferPtr);
else
return -1;
}
int main(){
String s("434");
// << is NOT overloaded
cout << s; //error
cout << s.AsInt();
return 0;
}
```

Lecture No 22

Practical implementation of Inheritance in c++:

Inheritance in Classes: If a class B inherits from class A, then B contains all the characteristics (information structure and behavior) of class A. The class whose behavior is being inherited is called base class and the class who inherits the behavior of base class is called derived class. Base class is also called parent class and child class is called also derived class Besides inherited characteristics, derived class may have its own unique characteristics

UML Notation

We use arrow from derived class to the parent class to show inheritance

Inheritance in C++

In c++ we can inherit a class from another class in three ways,

• Public • Private • Protected

"IS A" Relationship

Inheritance represents "IS A" relationship for example "a student IS A person". In general words we can say that inheritance represents,

"Derived class IS A kind of Parent class"

C++ Syntax of Inheritance

```
class ChildClass
: public BaseClass{
...
};
```

Example

```
class Person{
...
};
class Student: public Person{
...
};
```

Accessing Members: Public members of base class become public member of derived class. Private members of base class are not accessible from outside of base class, even in the derived class (Information Hiding)

Example

In the code given below Student and Teacher classes has been derived from single Person class,

In C++ char arrays (char []) are handled in two ways one way is statically using
statements like,

char name[30]; // static array of length 30 characters or dynamically as shown below,

char * name;

name = new char[30];

In dynamic creation of arrays we simply store char * in class and assign it a dynamic memory according to our need using new operator.

Constructors

• The anonymous object of base class must be initialized using constructor of base class

• When a derived class object is created the constructor of base class is executed before the constructor of derived class

Example

```
class Parent{
public:
Parent(){ cout <<
"Parent Constructor...";}
};
class Child : public Parent{
public:
Child(){ cout <<
"Child Constructor...";}
};
int main(){
Child cobj;
return 0;
}
```

Output:

Parent Constructor...

Child Constructor...

Constructor

• If default constructor of base class does not exist then the compiler will try to generate a default constructor for base class and execute it before executing constructor of derived class

• If the user has given only an overloaded constructor for base class, the compiler will not generate default constructor for base class

Example

```
class Parent{
public:
Parent(int i){}
};
class Child : public Parent{
public:
Child(){}
} Child_Object; //ERROR
```

Definition of Some Terms:

Default constructor: Default constructor is such constructor which either has no parameter or if it has some parameters these have default values. The benefit of default constructor is that it can be used to create class object without passing any argument.

Implicit Default constructor:

Compiler generates implicit default constructor for any class in case we have not given any constructor for the class.

Explicit Default constructor:

If user has given constructor for any class without any arguments or with all arguments with default values then it is also default constructor according to definition but it is explicit (user defined) default constructor. Now if a base class has only **non-default constructor** (constructor with parameters without default values), then when we will create object of any class derived from this base class compiler will not be able to call base class constructor as base class has no default constructor (constructor that can be called without giving any parameters) so compiler will generate error. We can avoid this error by calling base class non-default constructor in derived class constructor initializer list by ourself.

Base Class Initializer

• C++ has provided a mechanism to explicitly call a constructor of base class from derived class
• The syntax is similar to member initializer and is referred as base-class initialization

Example

```
class Parent{
public:
Parent(int i){…};
};
class Child : public Parent{
public:
Child(int i): Parent(i)
{…}
};
```

Example

```
class Parent{
public:
Parent(){cout <<
“Parent Constructor...”;}
...
};
class Child : public Parent{
public:
Child():Parent()
{cout << “Child Constructor...”;}
...
};
```

Base Class Initializer

• User can provide base class initializer and member initializer simultaneously

Example

```
class Parent{
public:
Parent(){…}
};
class Child : public Parent{
int member;
public:
Child():member(0), Parent()
{…}
};
```

Base Class Initializer

• The base class initializer can be written after member initializer for derived class
• The base class constructor is executed before the initialization of data members of derived class.

Initializing Members

• Derived class can only initialize members of base class using overloaded constructors
o Derived class cannot initialize the public data member of base class using member initialization list

Example

```
class Person{
public:
int age;
char *name;
...
public:
Person();
};
```

Example

```
class Student: public Person{
private:
int semester;
...
public:
Student(int a):age(a)
{ //error
}
};
```

Reason

• It will be an assignment not an initialization

Destructors

• Destructors are called in reverse order of constructor called
• Derived class destructor is called before the base class destructor is called

Example

```
class Parent{
public:
Parent(){cout <<“Parent Constructor”;}
~Parent(){cout<<“Parent Destructor”;}
};
class Child : public Parent{
public:
Child(){cout << “Child Constructor”;}
~Child(){cout << “Child Destructo”;}
}; Example
```

Output:

```
Parent Constructor
Child Constructor
Child Destructor
Parent Destructor
```