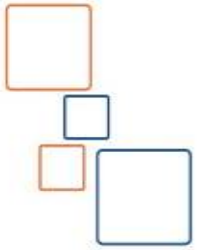
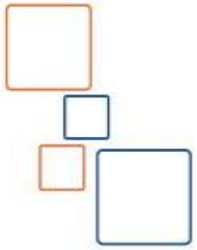
C++ Programming Language

Object-Oriented Programming Using C++



Invest In Yourself,
Develop Your Career

Introduction to Object-Oriented Programming Using C++



Invest In Yourself,
Develop Your Career

Course Duration and Evaluation

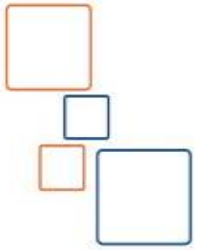
- Duration: 48 hours
  - 8 Lectures (24 hours)
  - 8 Labs (24 hours)

- Evaluation Criteria:
  - 40% on labs activities and assignments
  - 60% on written exam after 7 days of the last lectures.

Course Content

- D1: Object-Oriented Concepts and Terminologies
- D2: Polymorphism: Function Overloading
- D3: Friend function, Dynamic Area Problem and Copy Constructor
- D4: Polymorphism: Operator Overloading
- D5: Association among Classes and Collections of Objects
- D6: Weak Association among Classes, and Inheritance I
- D7: Inheritance II
- D8: Multiple Inheritance, Template Class, and Introduction to UML

Object-Oriented Concepts and Terminologies



Invest In Yourself,
Develop Your Career

Content

- Introduction to Object-Oriented programing
- Object-Oriented Terminologies
  - Class, Object, Instance
  - Members:
    - Attributes (Data),
    - Behaviors (Functions or Methods)
  - Messages
  - Encapsulation: private, public
  - Abstraction
  - Polymorphism:
    - Overloading:
      - Function overloading
      - Operator overloading
    - Overriding

Content

- Inheritance (is-a relationship):
  - Multi-Level Inheritance
  - Multiple Inheritance
- Constructors and destructor
- Static members and Instance (non-static) members
- Virtual function and dynamic binding
- Abstract methods (Pure virtual function), Abstract class, and Concrete class
- Association
- Strong (Composition)
- Weak (Aggregation)
- Object-Oriented programming characteristics
- Advantages of Object-Oriented programing
- Complex Example (with class keyword)
- Constructors and destructor

1.1 Introduction to Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields known as *attributes* or *properties*, and code, in the form of function known as *methods*.
- 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.
- Object Oriented programming (OOP) is a programming paradigm that relies on the concept of *classes* and *objects*. It is used to structure a software program into simple, *reusable pieces of code* blueprints (usually called classes), which are used to create individual instances of objects.

1.1 Introduction to Object-Oriented Programming

- OOP has become a fundamental part of software development. Thanks to the ubiquity of languages like Java and C++, you can't develop software for *mobile* unless you understand the object-oriented approach. The same goes for serious *web* development, given the popularity of OOP languages like Python, PHP and Ruby.
- The basic concept is that instead of writing a program, you create a class, which is a kind of template containing variables and functions. Objects are self-contained instances of that class, and you can get them to interact in fun and exciting ways.

1.1.1 Brief History about OOP

- The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.
  - The first object–oriented language was ***Simula*** (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
  - In 1970, *Alan Kay* and his research group at Xerox PARK created a personal computer named *Dynabook* and the first pure object-oriented programming language (OOPL) - ***Smalltalk***, for programming the *Dynabook*.
  - In the 1980s, *Grady Booch* published a paper titled Object Oriented Design that mainly presented a design for the programming language, ***Ada***. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
  - In the 1990s, *Coad* incorporated behavioral ideas to object-oriented methods.
- The other significant innovations were Object Modelling Techniques (OMT) by *James Rumbaugh* and Object-Oriented Software Engineering (OOSE) by *Ivar Jacobson*.

1.1.2 OOP Definition

• Grady Booch has defined object–oriented programming as:

*"A method of implementation in which programs are organized as **cooperative collections of objects**, each of which represents an **instance of some class**, and whose classes are all members of a **hierarchy** of classes united via **inheritance relationships**".*

1.1.3 What are the differences between Procedural and Object Oriented Programming?

- **Procedural Programming** can be defined as a programming model which is based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself. Example of programming languages: Pascal and C.
- **Object oriented programming** can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types. Example of programming and scripting languages: Java, C++, C#, Python, PHP, Ruby, Perl, Objective-C, Swift, Scala.

1.1.3 What are the are the differences between Procedural and Object Oriented Programming?

PROCEDURAL ORIENTED PROGRAMMING	OBJECT ORIENTED PROGRAMMING
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is ***no access specifier*** in procedural programming.	Object oriented programming have access specifiers like ***private***, ***public***, ***protected*** etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming have simple way for hiding data so it is ***less secure***.	Object oriented programming provides good ways data hiding so it is ***more secure***.
In procedural programming, *overloading is not possible*.	***Overloading*** is possible in object oriented programming.
In procedural programming, ***function*** is more important than data.	In object oriented programming, ***data*** is more important than function.
Procedural programming is based on unreal world.	Object oriented programming is based on ***real world***.

1.1.4 Features of OOP

- The important features of object–oriented programming are:
  - Bottom–up approach in program design
  - Programs organized around objects, grouped in classes
  - Focus on data with methods to operate upon object's data
  - Interaction between objects through functions
  - Reusability of design through creation of new classes by adding features to existing classes

1.2 OOP Basic Terminologies

- **Class**: A class is a category of objects, classified according to the members that they have. It is *Pattern* or *blueprint* for creating an object. A class contains all attributes and behaviors that describe or make up the object.
- **Object and Instances**: An instance of a particular class; an Object is the logical view and called Instance when be exist in memory.
- **Members**: Objects can have their own data, including variables and constants, and their own methods. The variables, constants, and methods associated with an object are collectively referred to as its *members* or *features*.
- **Attributes or properties**: which represent some of the features of a class datatype, it is may be primitive datatypes or objects from other classes
- **Behaviors**: it is represents the other part of features of the class and represent the operations which work on the attributes of the same class; i.e. it is the functions belongs to the class and called *methods*.

1.2 OOP Basic Terminologies

- **Message**: All the objects from the same class have the same methods and attributes, when you want to call a method you have to decide which object will call the method; i.e. you send a message to the object by calling a method to work on its attributes. The general form of message is: ***object_Name.member_Name()***
- **Encapsulation**: Encapsulation refers to mechanisms that allow each object to have its own data and methods. OOP has mechanisms for restricting interactions between components. So, some members may be ***private*** - it is accessible only by the methods of its class - and some other members may be ***public*** - it is accessible outside and inside the class through any object of that class.
- **Abstraction**: Refers to hiding the internal details of an object from the user, and its class is independent on the application which that object is used in.
- **Polymorphism**: Generally, the ability of different classes of object to respond to the same message in different, class-specific ways. Polymorphic methods are used which have one name but different implementations for in the same class or different classes. There are two types of polymorphism: ***Overloading***, and ***Overriding***.

1.2 OOP Basic Terminologies

- **Overloading**: Allowing the same method name to be used for more than one implementation in the same class, with different inputs. It has two types: ***Operator*** Overloading, and ***Function(Method)*** Overloading.
- **Function(method) Overloading**: Two or more methods with the same name defined within a class are said to be overloaded. This applies to both *constructors* and other *methods*.
- **Operator Overloading**: It is the mechanism to enrich the functionality of an operator defined in a programming language to deals among objects of a class, by define functions as members in that class to overload operator's functionalities.
- **Inheritance (*is-a*)**: Refers to the capability of creating a *new class* from an existing *class*. It is a relationship between classes where one class is a *parent (super or base)* of another *(Child, subclass, derived).* It implements *"is-a"* relationships between objects. Inheritance takes advantage of the commonality among objects to reduce complexity.

1.2 OOP Basic Terminologies

- **Inheritance Hierarchy**: The relationship between *super classes* and *subclasses* is known as an inheritance hierarchy.
- **Multi-Level Inheritance**: In Multilevel Inheritance a derived class can also inherited by another class.
- **Multiple Inheritance**: The ability of a class to extend more than one class; i.e. A *class* can inherit characteristics and features from more than one parent class.
- **Overriding**: A *method* defined in a *superclass* may be overridden by a *method* of the same name defined in a *subclass*. The two *methods* must have the same name and number and types of formal input parameters.
- **Constructor**: A constructor is an *instance method* that has the following Characteristics:
  - It has the same name as the class
  - It is auto calling when an object is created to be the initializing behavior of its life.
  - It can be overloaded, i.e. any class may have more than one constructor.
  - It can not return any datatype.

1.2 OOP Basic Terminologies

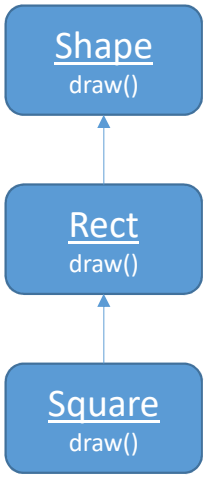
- **Destructor**: A destructor is an *instance method* that has the following Characteristics:
  - It has the same name as the class with *tilde* **"~"** character at the beginning of its name.
  - It is auto calling before during the destruction of an object, i.e. it is the *last behavior* of object's life. Actions executed in the destructor include the following:
    - *Recovering the heap space allocated during the lifetime of an object*
    - *Closing file or database connections*
    - *Releasing network resources*
  - It can not be overloaded, i.e. any class has only one destructor.
  - It can not return any datatype.
- **Static Members (Class members)**: A class can also hold data *variable* and *constants* that are *shared by* all of its objects and can handle *methods* that deal with an *entire class rather than an individual object.* These members are called ***class members*** or, in some languages (C++ and Java, for example), ***static members***. The members that are associated with objects are called ***instance members (Non-static members)***.

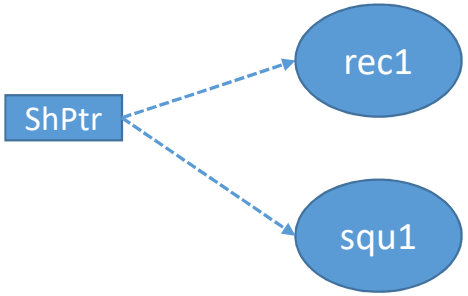
1.2 OOP Basic Terminologies

- **Association (*has-a*)**: Association establish relationship between two separate *classes* through their *objects*. The relationship can be *one to one*, *one to many, many to one and many to many*. Association is a relationship among classes which is used to show that instances of classes could be either ***linked to each other*** or combined *logically* or *physically* into some *aggregation*.
- **Weak Association (*Aggregation*)**: It is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed.
- **Strong Association (*Composition*)**: Composition is a specialized form of association in which if the container object is destroyed, the included objects would *cease to exist*. It is actually a ***strong*** type of association and is also referred to as a "***death***" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.

1.2 OOP Basic Terminologies

- **Dynamic (Late) Binding**: Dynamic binding also called dynamic dispatch is the process of linking method call to a specific sequence of code at run-time. It means that the code to be executed for a specific method call is not known until run-time.





1.2 OOP Basic Terminologies

- **Virtual Function**: A virtual function is a member function that you expect to be redefined in derived classes. 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. It is the mechanism of ***Dynamic Binding*** concept.
- **Abstract Method (Pure Virtual Function)**: An abstract method is a *method* that is declared, but contains no implementation, without body. It is needed to be overridden by the subclasses for standardization or generalization. (Ex: Event Handling).
- **Abstract Class**: Abstract classes are distinguished by the fact that you may not directly construct objects from them. An abstract class may have one or more *abstract methods*. Abstract classes may not be instantiated, and require *subclasses* to provide implementations for the abstract methods. It is needed for standardization or generalization.
- **Concrete Class**: Any class can be initiating objects directly from it, it is not abstract class.

1.3 OOP Fundamentals Characteristics

- Alan Kay, considered by some to be the father of object-oriented programming, identified the following such as fundamental to OOP:

1. Everything is an object
2. Each object has its own memory, which consists of other objects.
3. Every object is an instance of a class, i.e. any object has a particular type.
4. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary, to complete the task.
5. The class is the repository for behavior associated with an object. That is, all object that are instances of the same class can perform the same action, i.e. can receive the same messages.
6. Classes are organized into a singly rooted tree structure, called the ***inheritance hierarchy***.

1.4 Advantages of OOP

- Re-usability: It means reusing some facilities rather than building it again and again. This is done with the use of a class. We can use it ‘n’ number of times as per our need.
- Simplicity: Object oriented programming is based on ***real world***.
- Modularity: Parts of a program can be stopped without made a problem with other parts.
- Extensibility: Adding new functionalities to the program is easy because of modularity.
- Modifiability: Any module can be modified easy.
- Maintainability: Any problem can be cached easy and solved because of modularity.

1.5 Complex Class Example – Using struct

- We will write a class represent the complex number:

```
struct Complex
{     private:
            float real;
            float imag;
      public:
            Complex();       // constructor without parameters default constructor
            Complex(float r);     // constructor with one input parameter
            Complex(float r, float i);// constructor with two input parameters
            ~Complex();      // Destructor; only one for each class
            void setReal(float r) ;    // Setter for the real attribute
            void setImag(float i) ;    // Setter for the imag attribute
            float getReal() ;          // getter for the real attribute
            float getImag() ;          // getter for the imag attribute
            void print();              // a member to perform printing behavior
};
```

1.5 Complex Class Example – Using struct

```
Complex::Complex()
{
    real = image = 0 ;
    cout<<"\n Default Constructor is calling";
}
    Complex::Complex(float r, float i)
{
    real =r;        imag = i ;
    cout<<"\n Constructor with two parameters is calling";
}
    Complex::Complex(float r)
{
    real = imag = r;
    cout<<"\n Constructor with one parameters is calling";
}
    Complex::~Complex()
{
    cout<<"\n Destructor is calling";
}
```

1.5 Complex Class Example – Using struct

```
void Complex::setReal(float r)
{
     real = r ;
}
void Complex::setImag(float i)
{
     imag = i ;
}
float Complex::getReal()
{
     return real ;
}
float Complex::getImag()
{
     return imag ;
}
```

1.5 Complex Class Example – Using struct

```
void Complex::print()
{
      if(imag<0)
      {
            cout<<real<<" - "<<fabs(imag)<<"i"<<endl;
      }
      else
      {
            cout<<real<<" + "<<imag<<"i"<<endl;
      }
}
```

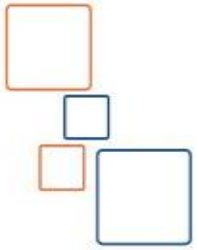
1.5 Complex Class Example – Using struct

```
Complex add (Complex c1, Complex c2)
{
     Complex temp;
     temp.setReal(c1.getReal() + C2.getReal());
     temp.setImag(c1.getImag() + c2.getImag());
     return temp;
}
Complex sub (Complex c1, Complex c2)
{
     Complex temp;
     temp.setReal(c1.getReal() - C2.getReal());
     temp.setImag(c1.getImag() - c2.getImag());
     return temp;
}
```

1.5 Complex Class Example – Using struct

```
int main()
{
    clrscr() ;
     Complex myComp1, myComp2(3, 4), resultComp(5) ;
     myComp1.setReal(7) ;
     myComp1.setImag(2) ;
     resultComp = add(myComp1,myComp2) ;
     resultComp.print() ;
     resultComp = sub(myComp1,myComp2) ;
     resultComp.print() ;
     return 0 ;
}
```

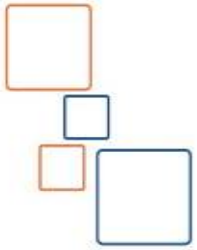
Lab Exercise



Assignments

- Complex Example with "add" and "subtract" stand alone functions.
- Note: the following points will cover in the lab (***ask your lab assistant***):
  - Moving from .C to .CPP compiler:
    - Flexible Deceleration of variables anywhere in the program.
    - Eliminating "void" keyword from brackets in function headers.
    - cin>> and cout<< streams.
    - Dynamic allocation with "new" and "delete" operators.
    - Call By Reference, with the Swap example.
  - Emphasize on Member Functions VS. Stand Alone Functions, and scope resolution operator.
  - Explain the Inline Function.
  - Explain that the default for struct is public

Polymorphism
Function Overloading



Content

- Class Vs Struct
- Change add and sub to be member functions, and make the needed changes
- Function overloading: setAll function with 3 implementation in class complex
- "***this***" pointer
- Stack Example
- Static variables and static methods
- Friend function
- Passing an object as function parameter: Stack Example on passing objects, "viewContent( )" friend function.

2.1 Using Class instead of Struct

- Actually the structure in C++ with added features like private and public sections, and including functions, was made for helping C programmers to move from structure programming to object-oriented programming.
- The only difference between a struct and class in C++ is the default accessibility of member variables and methods. In a struct they are public; in a class they are private.

2.2 Changing the add and sub in Complex example

```
struct Complex
{     private:
            float real;
            float imag;
      public:
            Complex();       // constructor without parameters default constructor
            Complex(float r);     // constructor with one input parameter
            Complex(float r, float i);// constructor with two input parameters
            ~Complex();      // Destructor; only one for each class
            void setReal(float r) ;    // Setter for the real attribute
            void setImag(float i) ;    // Setter for the imag attribute
            float getReal() ;          // getter for the real attribute
            float getImag() ;          // getter for the imag attribute
            Complex add(Complex c);    // a member to perform addition behavior
            Complex sub(Complex c);    // a member to perform subtraction behavior
            void print();              // a member to perform printing behavior
};
```

2.2 Changing the add and sub in Complex example

```
Complex Complex::add(Complex c)
{
    Complex temp;
    temp.real = this->real + C.real;
    temp.imag = this->imag + c.imag;
    return temp;
}
Complex Complex::sub(Complex c)
{
    Complex temp;
    temp.real = real - C.real;
    temp.imag = imag - c.imag;
    return temp;
}
```

2.2 Changing the add and sub in Complex example

```
int main()
{
    clrscr() ;
    Complex myComp1, myComp2(3, 4), resultComp(5) ;
    myComp1.setReal(7) ;
    myComp1.setImag(2) ;
    resultComp = myComp1.add(myComp2) ;
    resultComp.print() ;
    resultComp = myComp1.sub(myComp2) ;
    resultComp.print() ;
    return 0 ;
}
```

2.3 Default arguments values in C++

- Default arguments are those which are provided to the called function in case the caller statement does provide any value for them.
- Example:

```
int sum(int x=0, int y=5, int z=3, int w=7){
    return (x + y + z + w);
}
int main(){
    cout << sum() << endl;          // output = 15
    cout << sum(10) << endl;        // output = 25
    cout << sum(10, 15) << endl;    // output = 35
    cout << sum(10, 15, 25) << endl; // output = 57
    cout << sum(10, 15, 25, 30) << endl; // output = 80
    return 0;
}
```

2.4 Polymorphism: Function Overloading

- As we discussed in the example of Complex example, there was three constructors, default constructor (without parameters), constructor with one parameter, and the third one with two parameters. It is one of the function overloading example.
- This may be done for any of the other behaviors.
- We have a behavior for setting the real and imag attributes when we call it, called setALL(). We may make three methods with the same name but with different parameters, as follow

```
void setAll();
void setAll(float f);
void setAll(float r, float i);
```

- That means the same behavior is performed but with different methodologies.
- Rewrite the Complex example with using class instead od struct and add the setAll().

2.4 Polymorphism: Function Overloading

```
class Complex
{          // The default accessibility in class is private so if we do not
           // label the section with accessibility type, it will be private.
           float real;
           float imag;
     public:
           Complex();      // constructor without parameters default constructor
           Complex(float r);              // constructor with one input parameter
           Complex(float r, float i);     // constructor with two input parameters
           ~Complex();                 // Destructor; only one for each class
           void setReal(float r) ;   // Setter for the real attribute
           void setImag(float i) ;   // Setter for the imag attribute
```

2.4 Polymorphism: Function Overloading

```
class Complex
{           …
            …
     float getReal() ;          // getter for the real attribute
     float getImag() ;          // getter for the imag attribute
     void setAll();             // setting both of real and imag with 0
     void setAll(float f);      // setting both of real and imag with f
     void setAll(float r, float i); // setting real and imag with r & i
     Complex add(Complex c);    // a member to perform addition behavior
     Complex sub(Complex c);    // a member to perform subtraction behavior
     void print();              // a member to perform printing behavior
};
```

2.4 Polymorphism: Function Overloading

```
void Complex::setAll()
{
      real = imag = 0;
}
void Complex::setAll(float f)
{
      real = f;
      imag = f;
}
void Complex::setAll(float r, float i)
{
      real = r;
      imag = i;
}
```

2.5 The "*this*" pointer

- Every object in C++ has access to its own address through an important pointer called ***this*** pointer.
- To understand 'this' pointer, it is important to know how objects look at functions and data members of a class:
  1. Each object gets its own copy of the data member.
  2. All objects access the same function definition as present in the code segment.
- Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
- So, the "***this"*** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object by address.
- The stand alone and friend (will be explained later) functions do not have a ***this*** pointer, because they are not members of a class. Only member functions have a ***this*** pointer.

2.5 The "*this*" pointer

```
        Complex::Complex(float real, flaot imag)
{
        this->real = real;
        this->imag = imag;
}
void    Complex::setAll(float f)
{
    this->real = f;
    this->imag = f;
}
```

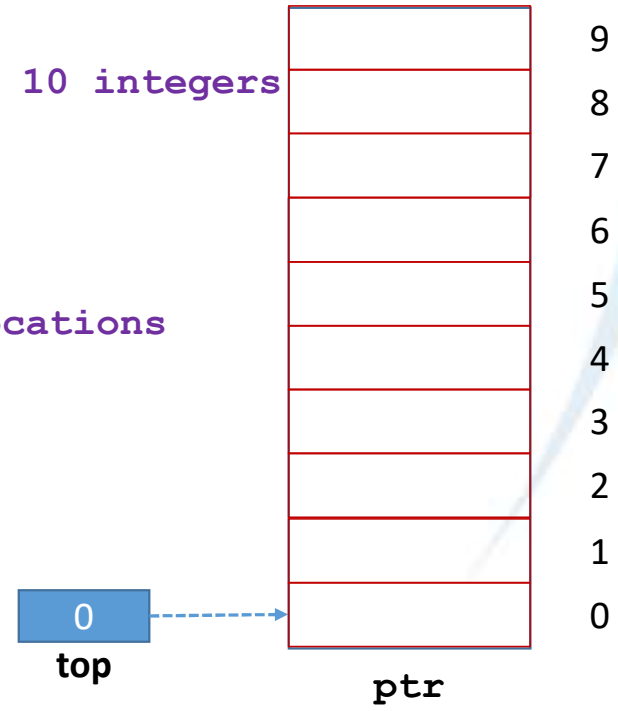
2.6 Stack Example

- We will make a class to represent a stack data structure (array based) with dynamic allocation to the array places.

```
class Stack
{
  private:
     int top ;       // indicator to the top of stack
     int size;       // the max size of the stack
     int *ptr;       // the pointer to create and access the stack elements
```

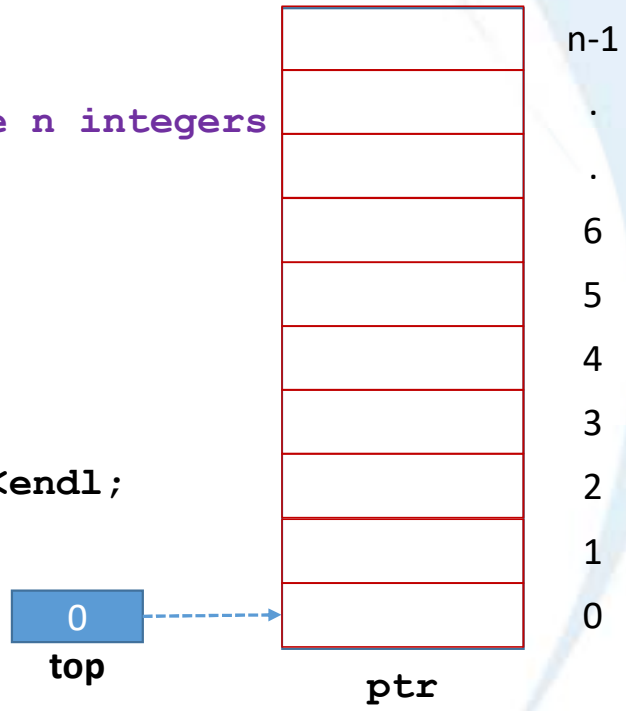
2.6 Stack Example

```
public:
    int isFull() ;  //functions using to check, they offer no service
    int isEmpty(); //to the outside world.
    Stack()  // The default constructor with stack size is 10 integers
    {
          top = 0 ;       // initialize stack state
          size = 10;
          ptr = new int[size];      // Allocate the stack locations

          cout<<"This is the default constructor"<<endl;
    }
```



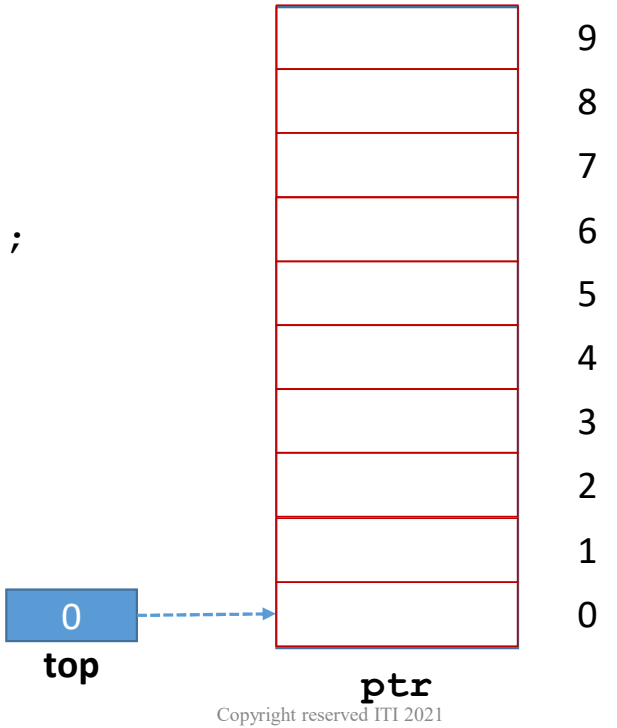
2.6 Stack Example

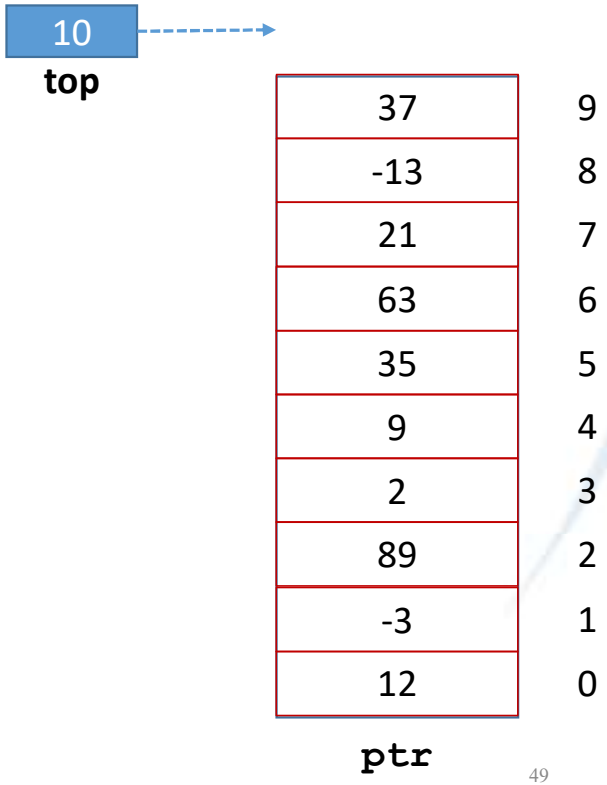
```
Stack(int n) // The default constructor with stack size n integers
{
      top = 0;
      size = n;
      ptr = new int[size];

      cout<<"This is a constructor with one parameter"<<endl;
}
~Stack()
{
      delete[] ptr;  size=0;

      cout<<"This is the destructor"<<endl;
}
```



2.6 Stack Example

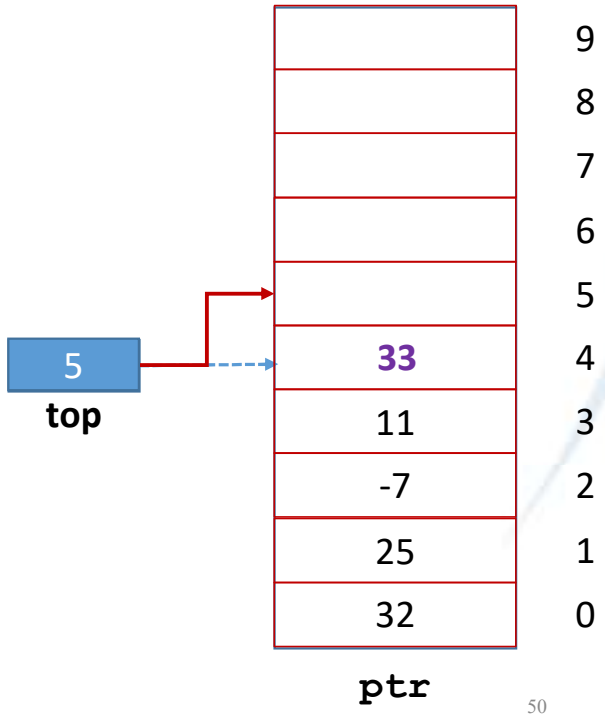
```
        int push(int n);
        int pop(int & n);
};

int Stack::isFull()
{
        return (top==size) ;
}

int Stack::isEmpty()
{
        return (top==0) ;
}
```





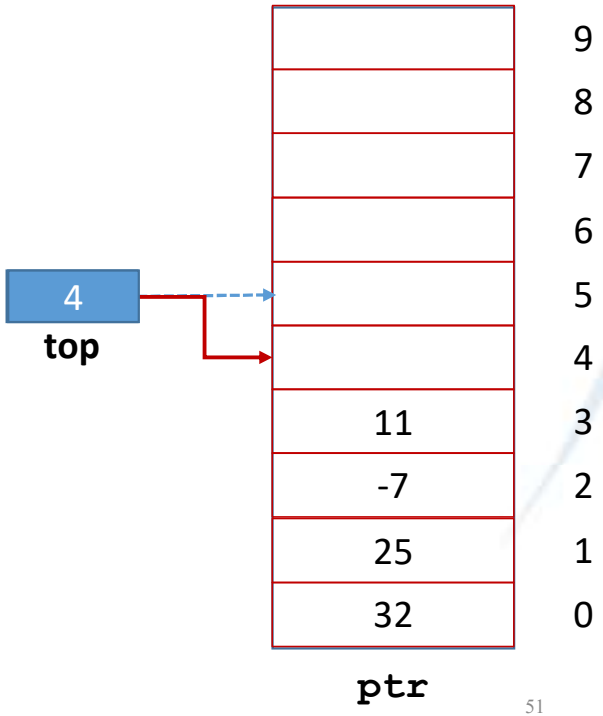
2.6 Stack Example

```
int Stack::push(int n)
{
      if (isFull())
            return 0;
      ptr[top] = n;
      top++;
      return 1
}
```



2.6 Stack Example

```
int Stack::pop(int & n)
{
      if (isEmpty())
            return 0;
      top--;
      n = ptr[top];
      return 1;
}
```



2.7 Static Variables and Static Methods

- We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by re-declaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.
- Let's apply that on the Stack example, we will define a static private **int** variable identified as **counter**, to count the exist objects currently in memory, for accessing it we have to make a method called **getCounter** in public section, and it will be static for accessing at any time *before/after* creating any object.
- The changing of the counter static variable will be done by the constructors (increment it) and destructor (decrements it).

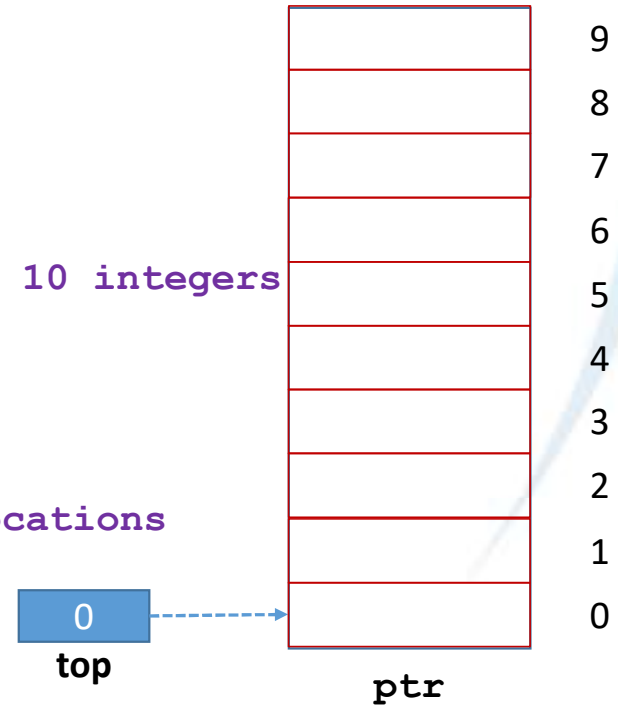
2.7 Static Variables and Static Methods

- We will make a class to represent a stack data structure (array based) with dynamic allocation to the array places.

```
class Stack
{
  private:
     int top ;       // indicator to the top of stack
     int size;       // the max size of the stack
     int *ptr;       // the pointer to create and access the stack elements
     static int counter; // it is not allowed to initialize it here in C++
```

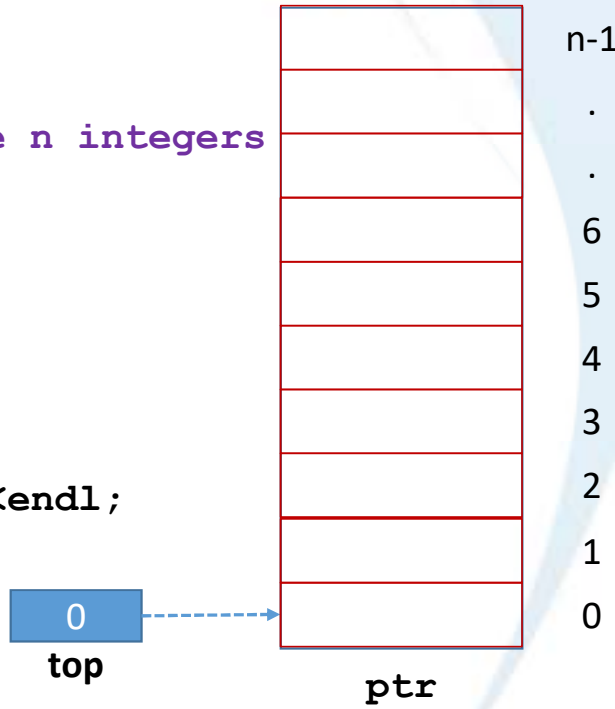
2.7 Static Variables and Static Methods

```
public:
    int isFull() ;  //functions using to check, they offer no service
    int isEmpty(); //to the outside world.
    static int getCounter()
    {
         return counter;
    }
    Stack()  // The default constructor with stack size is 10 integers
    {
         top = 0 ;       // initialize stack state
         size = 10;
         ptr = new int[size];      // Allocate the stack locations
         counter ++;
         cout<<"This is the default constructor"<<endl;
    }
```



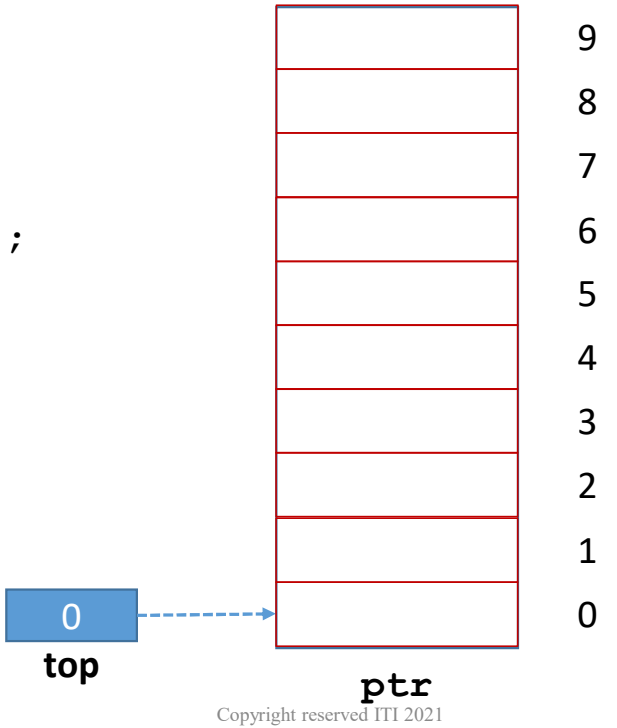
2.7 Static Variables and Static Methods

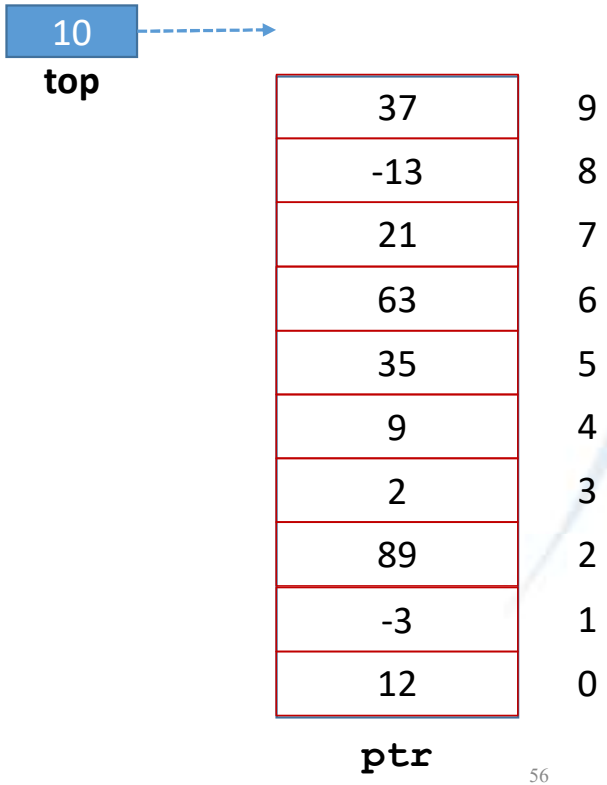
```
Stack(int n) // The default constructor with stack size n integers
{
      top = 0;
      size = n;
      ptr = new int[size];
      counter ++;
      cout<<"This is a constructor with one parameter"<<endl;
}
~Stack()
{
      delete[] ptr;  size=0;
      counter --;
      cout<<"This is the destructor"<<endl;
}
```



2.7 Static Variables and Static Methods

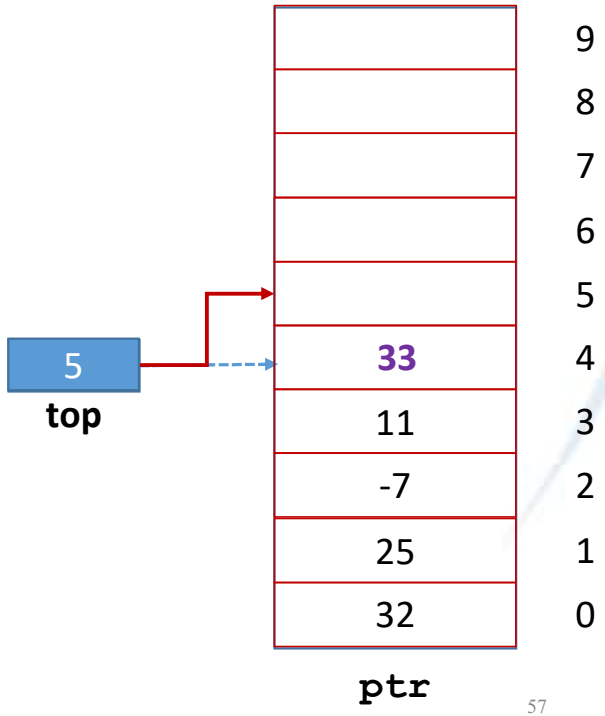
```
      int push(int n);
      int pop(int & n);
};

int Stack::isFull()
{
      return (top==size) ;
}

int Stack::isEmpty()
{
      return (top==0) ;
}
```





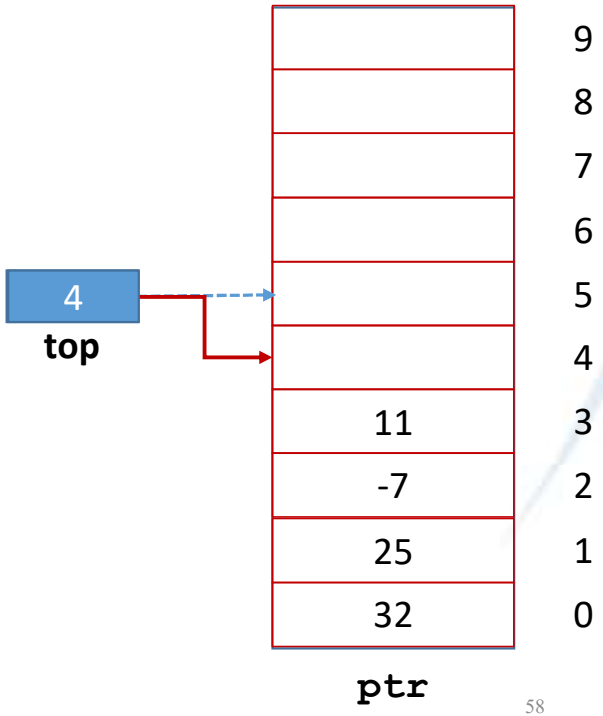
2.7 Static Variables and Static Methods

```
int Stack::push(int n)
{
      if (isFull())
            return 0;
      ptr[top] = n;
      top++;
      return 1
}
//static variable initialization
int Stack::counter = 0;
```



2.7 Static Variables and Static Methods

```
int Stack::pop(int & n)
{
      if (isEmpty())
            return 0;
      top--;
      n = ptr[top];
      return 1;
}
```



2.7 Static Variables and Static Methods

```
int main()
{
     clrscr() ;
     int num ;
     Stack s1(2) ;
     s1.push(5);
     s1.push(14);
     s1.push(20) ;      // Stack is full so the third one will not in
     if(s1.pop(num))
     {
          cout<<num<<endl ; // the result will be 14
     }
```

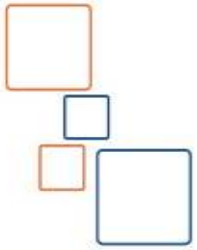
2.7 Static Variables and Static Methods

```
    if(s1.pop(num))
    {
        cout<<num<<endl ; // It will print 5 last element
    }
    if(s1.pop(num))
    {
        cout<<num<<endl ; // The stack is empty
    }
    else cout<< "\n Stack is empty …";
    getch() ;
    return 0;
}
```

2.7 Static Variables and Static Methods

```
int main()
{
     cout<<"\nThe number of objects created = "<<Stack::getCounter();
     Stack s1, s2(5);
     cout<<"\nThe number of objects created = "<<Stack::getCounter();
     {
          Stack s3(10);
        cout<<"\nThe number of objects created = "<<Stack::getCounter();
     }
     cout<<"\n the number of objects created = "<<Stack::getCounter();
     getch();
     return 0;
}
```

Lab Exercise

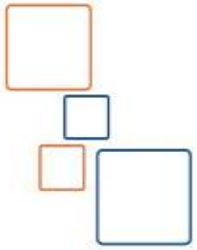


Assignments

- Continue Complex (setters overloading and make add sub as members).
- Stack Example:
  - Constructors.
  - Destructor.
  - Accessing members
  - Static counter and static getCounter
  - main( ) test.

- Note: You should Trace the code using F7 and F8

Friend function, Dynamic Area Problem and Copy Constructor



Invest In Yourself,
Develop Your Career

Content

- Friend function
- Passing an object as function parameter: Stack Example on passing objects, "viewContent( )" friend function.
- Dynamic Area Problem and its solution
  - Using Call by Reference.
  - Using the Copy Constructor.
- Copy constructor
  - Example on Stack

3.1 Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a stand alone function, or member function, in which case the entire class and all of its members are friends.
- The friend function ***violates*** an important concept in Object-Oriented programming; "Encapsulation".
- To declare all member functions of class *ClassTwo* as friends of class *ClassOne*, place the following declaration in the definition of class *ClassOne*

```
friend class ClassTwo;
```

- Let's make a friend function to class Stack, for viewing the contents of the stack without pop its data.

3.1 Friend Function

```
Class Stack{
      private:
      int top ;        // indicator to the top of stack
      int size;        // the max size of the stack
      int *ptr;        // the pointer to create and access the stack elements
      static int counter;  // it is not allowed to initialize it here in C++
public:
      int isFull() ;  //functions using to check, they offer no service
      int isEmpty();  //to the outside world.
      static int getCounter();
      Stack();
      Stack(int n);
      ~Stack();
      int push(int n);
      int pop(int &n);
      friend void viewContent(Stack s);
};
```

3.1 Friend Function

```
void viewContent (Stack s)
{
     for(int i = 0; i<s.top ; i++)
          cout<<"\n Element no ("<<i+1<<") ="<< s.ptr[i];
}
```

- As we show, there is a non-member function can access the private members of a class, just it is declared as a friend function inside that class.

3.1 Friend Function

```
int main()
{     clrscr() ;
      int num ;
      Stack s1(5) ;
      s1.push(5);
      s1.push(14);
      s1.push(20) ;
      viewContent(s1); // All the contents are displayed
      if(s1.pop(num))
      {
            cout<<num<<endl ; // the result will be 20
      }
      getch();
      return 0;
}
```

3.2 Dynamic Area Problem

- When we pass an object to a function as call by value, a copy of the object is passed to the function. How this copy is created?
- Actually, the C++ compiler make another constructor by default– if we do not write it- called *Copy Constructor*, it is calling when a copy of an object is needed – like pass the object by value or return it by value -, it makes a shallow copy, i.e. byte wise copy.
- If the object creation make dynamic memory allocation, it causes a problem when pass the object by value or return it by value, Why? …
- When the function return, all the local variables and arguments are removed from memory, so the shallow copy of the passed object will removed then the destructor is calling for that shallow copy, here the shallow destructing will deallocate the memory locations which is shared with the original object.

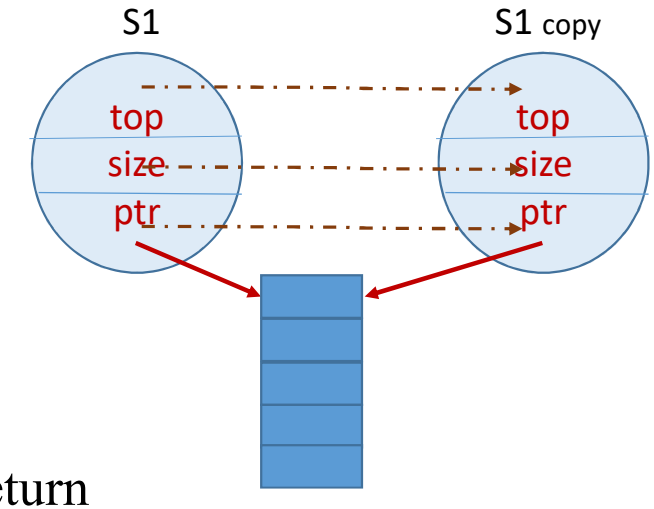
3.2 Dynamic Area Problem

- For example, when we made the add and sub functions as standalone in the Complex example, we observe that there are number of destructors grater than the number of constructor and it is not sense!!
- But in the Complex example there was no data losing or no logical problem appear because Complex members are embedded and primitive.
- If there is a member is a pointer and it make a dynamic allocation, it will be logical problem and data losing may be happened.
- Let’s see the example of Stack class in the following slides….

3.2 Dynamic Area Problem

- Lets see what happen when the object of a Stack class are passed to the viewContent function:

  viewContent(S1)

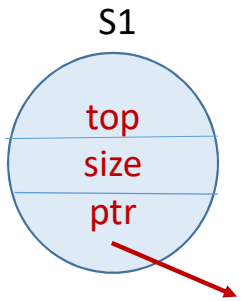


- When the function return

3.2 Dynamic Area Problem

- Lets see what happen when the object of a Stack class are passed to the viewContent function:

  viewContent(S1)



- When the function return

3.2 The solution of Dynamic Area Problem

- There are two ways to solve that problem:
  - Make the passing to an object by reference instead of call by value, or
  - Write a copy constructor to make a concrete (deep) copy of the object to shallow copy.
- The side effect of send an object by reference, the object attributes may be changed by the function – specially if the function is friendly function like Stack example-.
- The side effect of copy constructor solution that the numbers of objects created for calling by value and return by value is overhead on program execution time, i.e. affect on the performance.

3.2.1 Make the viewContent call by reference

```
void viewContent (Stack& s)
{
    for(int i = 0; i<s.top ; i++)
        cout<<"\n Element no ("<<i+1<<") ="<< s.ptr[i];
}
```

3.3 Copy Constructor

- A copy constructor is a member function which initializes an object using another object of the same class.
- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- If a copy constructor if not defined in a class, the compiler itself defines one, which make a shallow copy of object by make the byte wise equality.
- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
- The general form of the copy constructor as follow:

```
ClassName (ClassName & obj){}
```

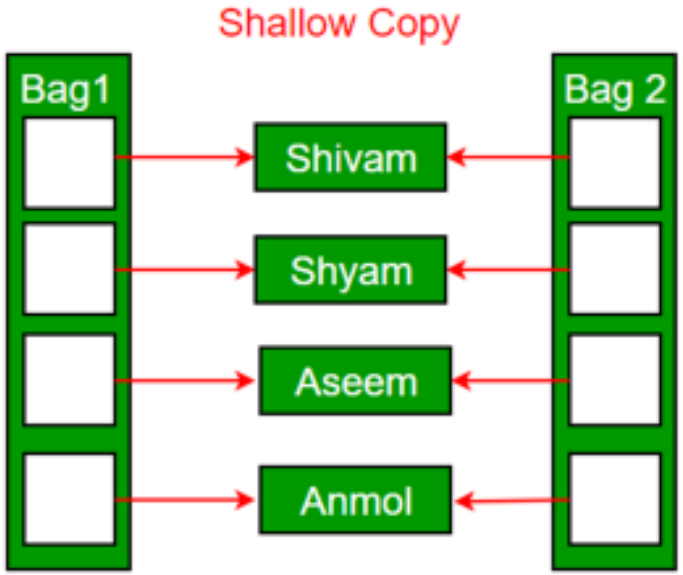
- That means, the copy constructor has only one parameter which is a reference to an object from the same class.

3.3 Copy Constructor

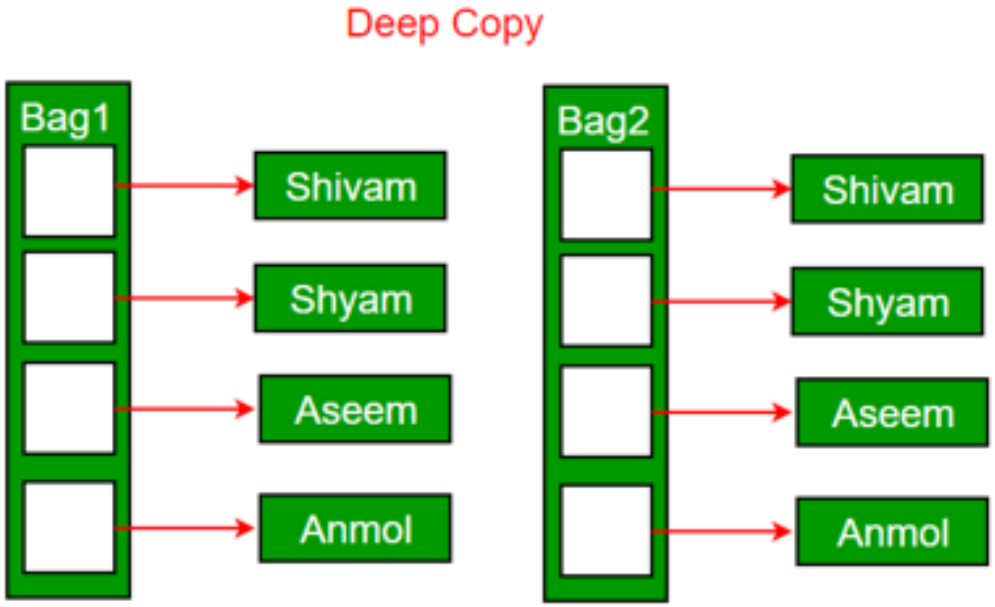
- The Copy Constructor is auto calling in the following cases:
  1. When an object of the class is returned by value.
  2. When an object of the class is passed (to a function) by value as an argument.
  3. When an object is constructed based on another object of the same class.
- Some times the compiler needs to generate a temporary object, so it may call the copy constructor.
- Default copy constructor does only shallow copy of the object, so at case – like stack class- we need to implement the copy constructor in our class to describe how make a deep copy of an object form that class.

3.3 Copy Constructor

Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor.



3.3 Copy Constructor: Stack Example

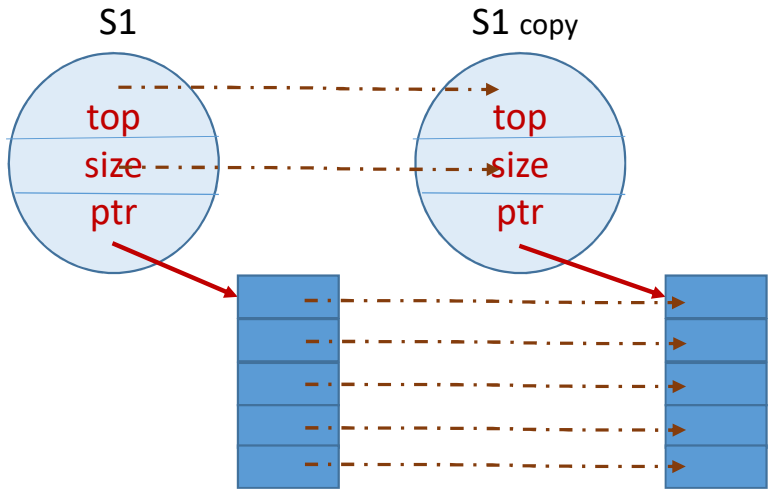
```
Class Stack{
     private:
     int top ;      // indicator to the top of stack
     int size;      // the max size of the stack
     int *ptr;      // the pointer to create and access the stack elements
     static int counter;  // it is not allowed to initialize it here in C++
public:
     int isFull() ;  //functions using to check, they offer no service
```

3.3 Copy Constructor: Stack Example

```
    int isEmpty(); //to the outside world.
    static int getCounter();
    Stack();
    Stack(Stack & s);
    Stack(int n);
    ~Stack();
    int push(int n);
    int pop(int &n);
    friend void viewContent(Stack s);
};
```

3.3 Copy Constructor: Stack Example

```
Stack(Stack & s)
{
    top = s.top
    size = s.size;
    ptr = new int[size];
    for(int i=0;i<top;i++)
        ptr[i] = s.ptr[i];
}
```

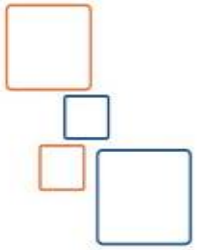


3.3 Copy Constructor

- **How to create an object which is initialized by another object of the same type:**

```
int main()
{    int num
     Stack s1(10);
     s1.push(7);
     s1.push(13);
     s1.push(-5);
     Stack s2(s1);
     s2.pop(num);
     cout<<num<<endl;
     s1.pop(num);
     cout<<num<<endl;
     getch();
     return 0;
}
```

Lab Exercise

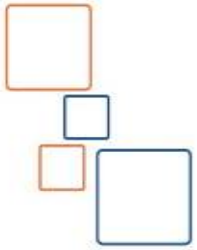


Assignments

- Continue the Stack Example:
  - Write the "viewContent( )" friend function.
  - Try without copy constructor. (to see the extra destructor call).
  - Try to send the Complex object by reference to viewContent.
  - Write the copy constructor, and then try again (back to call b).

- Note: You should Trace the code using F7 and F8

Polymorphism: Operator Overloading



Invest In Yourself,
Develop Your Career

Content

- Operator Overloading
  - Example on Class Complex: +, ++, =, +=, ==, casting
- Using operator overloading on shift operators, << and >>, for **cin**, and **cout**

4.1 Operator Overloading

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**.
- A feature in C++ that enables the redefinition of operators. This feature operates on user defined objects. All overloaded operators provides syntactic sugar for function calls that are equivalent. Without adding to / changing the fundamental language changes, operator overloading provides a pleasant façade.
- It is a type of polymorphism in which an operator is overloaded to give user-defined meaning to it. Overloaded operators is used to perform operation in user-defined datatype.
- Operator Overloading means providing multiple definition for the same operator.
- Operator overloading is a specific case of polymorphism in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. It can easily be emulated using function calls.

4.1.1 Rules for operator overloading in C++

- In C++, following are the general rules for operator overloading.
- Only built-in operators can be overloaded. New operators can not be created.
- Arity (arguments or operands) of the operators cannot be changed.
- Precedence and associativity of the operators cannot be changed.
- Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
- Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
- 6) Assignment (=), subscript ([]), and function call ("()") 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 (stand alone) functions.

4.1.2 Operators that cannot be overloaded in C++

- In C++ we can overload some operators like +, -, [], = etc. But we cannot overload any operators in it. Some of the operators cannot be overloaded. These operators are like below:
  - “.” Member access or dot operator
  - “? : ” Ternary or conditional operator
  - “::” Scope resolution operator
  - “.*” Pointer to member operator
  - “sizeof” The object size operator
- Note: The short-circuit operators && and || is not affected by the overloading.
- Now we will make operator overloading on Complex class

4.2.1 Mathematical Operator: +

```
class Complex
{     ...
      public:
            Complex(Complex & c);    // Copy constructor
            Complex operator + (Complex & c) // using in form: Com1 + com2
            {
                  Complex temp(real+c.real, imag+c.imag);
                  return temp;
                  // return Complex(real+c.real, imag+c.imag);
            }
            Complex operator + (float f) // using in form: Com1 + f
            {
                  Complex temp(real+f, imag);
                  return temp;
                  // return Complex(real+f, imag);
            }

};
```

4.2.1 Mathematical Operator: +

- What we do if we want to add a complex object to a float number?
- In the above cases there is an object which call the operator overloaded method, but in this case the float variable needs to call an overloaded function; which has to be stand alone to inform the compiler how to use the operator + between float and complex.
- If we need this stand alone function to deal directly with the private members of the complex class, make it friend function.

4.2.1 Mathematical Operator: +

```
class Complex
{       ...
        public:
                Complex(Complex & c);        // Copy constructor
                Complex operator + (Complex & c) // using in form: Com1 + com2
                {
                        Complex temp(real+c.real, imag+c.imag);
                        return temp;
                        // return Complex(real+c.real, imag+c.imag);
                }
                Complex operator + (float f) // using in form: Com1 + f
                {
                        Complex temp(real+f, imag);
                        return temp;
                        // return Complex(real+f, imag);
                }
                friend Complex operator +(float f, Complex & C);

};
```

4.2.1 Mathematical Operator: +

```
Complex operator+ (float f, Complex & c)
{
    Complex temp(f+c.real, c.imag);
    return temp;
    // return c+f;
}
```

4.2.1 Mathematical Operator: +

```
int main()
{
    Complex c1(12, 7),c2(10, -5);
    Complex c3;
    c3=c1+c2;
    c3.print();
    c3=c1+13.65;
    c3.print();
    c3=6.2+c2;
    c3.print();
    getch();
    return 0;
}
```

4.2.2 Unary Operator: ++

```
class Complex
{     ...
      public:
            Complex& operator++ ()// prefix
            {     real++;
                  imag++;
                  return *this;
            }
            Complex operator++ (int dumy)// postfix
            {     Complex temp(*this);
                  real++;
                  imag++;
                  return temp;
            }
};
```

4.2.2 Unary Operator: ++

```
int main()
{
    Complex c1(12, 7),c2(10, -5),c3;
    c3=++c1;
    c1.print();
    c3.print();
    c3=c2++;
    c2.print()
    c3.print();
    getch();
    return 0;
}
```

4.2.3 Assignment Operators: =

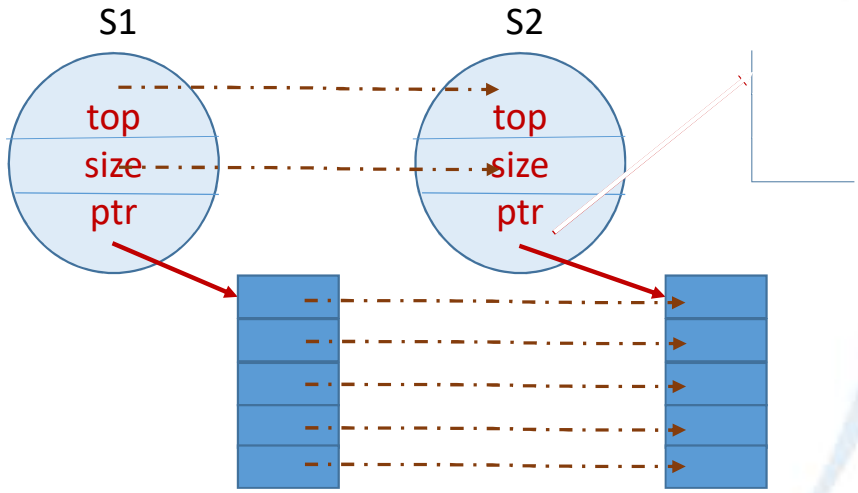
```
class Complex
{   ...
    public:
        Complex& operator= (Complex & c)// for cascading =
        {
            real=c.real;
            imag=c.image;
            return *this;
        }

};
```

4.2.3 Assignment Operators: =

```
int main()
{
    Complex c1(9, 7), c2(13), c3;
    c3=c1;
    c1.print();
    c3.print();
    c2=c1=c3;
    c1.print();
    c2.print();
    c3.print();
    getch();
    return 0;
}
```

4.2.3 Assignment Operators: = , For Stack Example

```
class Stack
{     ...
   public:
     Stack& operator= (Stack & c)
     {
          top = c.top;
          size = c.size;
          delete [] ptr;
          ptr = new int[size];
          for(int i; i<top; i++)
               ptr[i] = c.ptr[i];
     }

};
```



4.2.3 Assignment Operators: =

```
int main()
{    int n;
     Stack s1(10),s2(7);
     s1.push(37);
     s1.push(-12);
     s1.push(26);
     s2=s1;
     s2.pop(n);
     cout<<n<<endl;     // print 26
     s1.pop(n);
     cout<<n<<endl;     // print 26 too
     getch();
     return 0;
}
```

4.2.3 Assignment Operators: +=

```
class Complex
{   ...
    public:
        Complex& operator+= (Complex & c)
        {
            real+=c.real;
            imag+=c.image;
            return *this;
        }

};
```

4.2.3 Assignment Operators: =, +=

```
int main()
{
    Complex c1(9, 7), c2(13), c3;
    c2+=c1;
    c1.print();
    c2.print();
    c2=c1+=c3;
    c1.print();
    c2.print();
    c3.print();
    getch();
    return 0;
}
```

4.2.4 Comparison Operators: ==

```
class Complex
{    ...
     public:
          int operator== (Complex & c)
          {
               if((real==c.real) && (imag==c.image))
                         reurn 1;
               return 0;
          // return ((real==c.real) && (imag==c.image));
          }

};
```

4.2.4 Comparison Operators: ==

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cout<<"c1 equals to c3 ? "<<(c1==c3)<<endl;
    if(!(c1==c2))
        cout<<"c1 is not equal to c2"<<endl;
    getch();
    return 0;
}
```

4.2.5 Unary Operator: casting to float

```
class Complex
{   ...
    public:
        operator float ()
        {
            return real;
        // return sqrt(real*real+imag*imag);
        }

};
```

4.2.5 Unary Operator: casting to float

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cout<<"The float of c1 "<<float(c1)<<endl;
    float f = (float)c2;
    cout<<"The casting of c2 is: "<<f<<endl;
    getch();
    return 0;
}
```

4.2.6 Shift operators with istream and ostream: cin, cout

```
class Complex
{   ...
    public:
      friend istream& operator>>(istream &in, Complex& c);
      friend ostream& operator<<(ostream &out, Complex& c);

};
```

4.2.6 Shift operators with istream and ostream: cin, cout

```
istream& operator>>(istream& in, Complex & c)
{
    cout<<"\n Enter real part:";
    in>>c.real;
    cout<<"Enter imag part";
    in>>c.imag;
    return in;
}
```

4.2.6 Shift operators with istream and ostream: cin, cout

```
ostream& operator<<(ostream& out, Complex & c)
{
    if(imag<0)
            out<<real<<" - "<<fabs(imag)<<"i"<<endl;
    else
            out<<real<<" + "<<imag<<"i"<<endl;
    return out;
}
```

4.2.6 Shift operators with istream and ostream: cin, cout

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cin>>c2;
    cout<<c1<<c2<<c3;

    getch();
    return 0;
}
```

4.2.7 Array operator [ ]

```
class MyArray
{
    private:
        int size;
        int * data;
    public:
        MyArray(int size)
        {
            this->size = size;
            data= new int[size];
        }
```

4.2.7 Array operator [ ]

```
    ~MyArray()
    {
        delete [] data;
    }
    int & operator [] (int index)
    {
        if(index<size)
                return data[index];
        else
        {
            cout<<"Array Out of Boundaries Exception";
            exit(1);
        }
    }
};
```

4.2.7 Array operator [ ]

```
int main()
{
    MyArray arr(13);
    for(int i=0; i<12; i++)
        cin>>arr[i];
    for (i=0; i<12; i++)
        cout<<arr[i];
    getch();
    return 0;
}
```

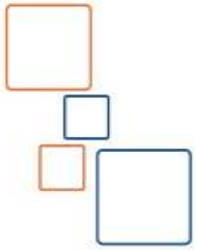
4.2.8 Function operator

```
class Point
{   private:
        int x,y;
    public:
        Point();
        Point(int a,int b);
        void operator()(int a, int b)
        {   x=a;
            y=b;
        }
    void print(){cout<<"\n Point Data: x="<<x<<" y="<<y<<endl;}
};
```

4.2.8 Function operator

```
int main ()
{
    Point p1(12, 20);
    p1.print();
    p1(-5, 132);
    p1.print();
    getch();
    return 0;
}
```

Lab Exercise

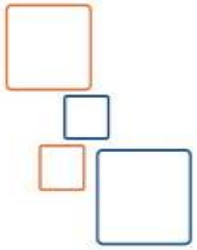


Assignments

- Continue Complex Example: Overloading of all the other operators, and create a main function to test all the operations.
- Continue Stack Example: Overloading the operator =.
- Make the overloading on shift operator of cin and cout to print complex objects
- Make the array operator overloading in class MyArray
- Make the function operator overloading on class point, to set the XY coordinates of the point.

- Note: You should Trace the code using F7 and F8

Association among Classes and Collections of Objects



Content

- Association Class Relations (has-a relationship)
  - Strong: Example of Point, Rect, Circle.
  - Embedded Objects
  - Constructors chaining
- Collections and Temporary Objects:
  1) Static allocation: single objects and array of objects. (how to specify desired constructors for each object in the array).
  2) dynamic allocation: pointers to single object and array of objects.

5.1 Association Relationship (has-a relation)

- Association establish relationship between two separate *classes* through their *objects*. The relationship can be *one to one*, *one to many, many to one and many to many*. Association is a relationship among classes which is used to show that instances of classes could be either ***linked to each other*** or combined *logically* or *physically* into some *aggregation*.
- In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.
- It represent ***has-a*** relationship among classes.
- For example, if there are two classes, Institute and Employee, the Institute class has a collections of Employee objects.

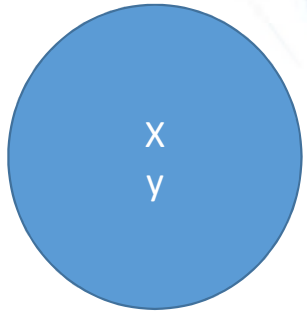
5.1.2 Strong Association (Composition)

- Composition is a specialized form of association in which if the container object is destroyed, the included objects would *cease to exist*. It is actually a ***strong*** type of association and is also referred to as a "***death***" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.
- It represents *part-of* relationship.
- In composition, both the entities are dependent on each other. When there is a composition between two entities, the composed object cannot exist without the other entity.
- Embedded object is a form of strong association

5.1.3 Embedded Objects as Strong Assocaiation

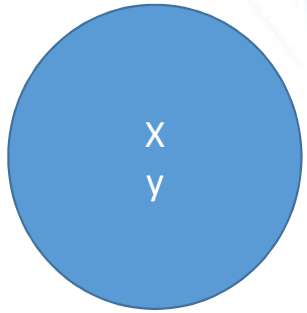
- An embedded object is physically stored in the container object, In other words, the embedded object is actually a part of the container object in which it resides.
- Let's take an example of classes Point, Line, Rect, and Circle
- Any line can be described by two points; start and end.
- And so, any rectangle can be described by two points; upper left and lower right
- Circle is described by central point and radius.

5.1.3 Embedded Objects as Strong Association




```
class Point
{  private:
     int x ;
     int y ;
  public:
     Point()
     {    x = y = 0 ;
          cout<<"Point default constructor is calling"<<endl;
     }
     Point(int m)
     {    x = y = m ;
          cout<<"Point one parameter constructor is calling"<<endl;
     }
```

5.1.3 Embedded Objects as Strong Association

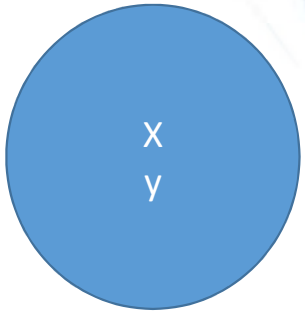



```
    Point(int m, int n)
    {   x = m ;
        y = n ;
        cout<<"Point two parameter constructor is
calling"<<endl;
    }
    ~Point(){cout<<"Point destructor is calling"<<endl;}
```

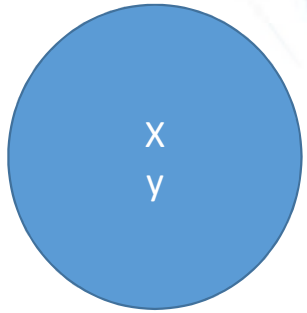
5.1.3 Embedded Objects as Strong Association

```
void setX(int m)
{    x = m ;
}
void setY(int n)
{    y = n ;
}
void setXY(int m, int n)
{    x=m;
     y=n;
}
```

Class Point



5.1.3 Embedded Objects as Strong Association

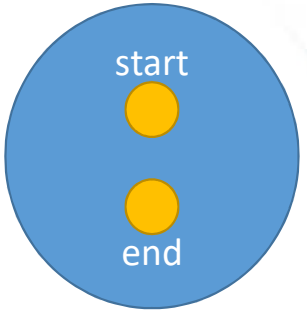



```
    int getX()
    {    return x ;
    }
    int getY()
    {    return y ;
    }
    void print(){cout<<"\n Point Data: x="<<x<<" y="<<y<<endl;}
};
```

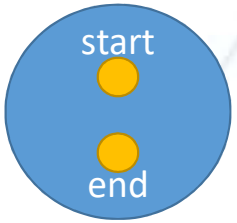
5.1.3 Embedded Objects as Strong Association

```
class Line
{  private:
     Point start;
     Point end;
  public:
     Line()
     {    start.setXY(0,0); end.setXY(0,0);
          cout<<"Line default constructor is calling"<<endl;
     }
```

Class Line



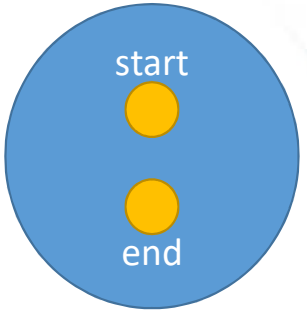
L1



5.1.3 Embedded Objects as Strong Association

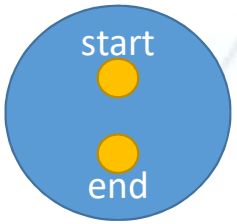
Java™ Education and Technology Services

Class Line




```
    Line(int x1, int y1, int x2, int y2)
    {
        start.setXY(x1,y1);    end.setXY(x2,y2);
        cout<<"Line with 4 parameter constructor is calling"<<endl;
    }
    ~Line(){cout<<"Line destructor is calling"<<endl;}
    void print()
    {   cout<<"\nStart:"; start.print();
        cout<<"\nEnd:";   end.print();
    }
};
```

L1



5.1.3.1 Constructor and destructor Chaining in case of embedded objects

- In fact, to create an object from class Line, all the instance attributes must be created and allocated in the memory before any constructor of the Line constructors could be called. So, the two objects from class Point, start and end, are completely created and allocated as apart of Line object, then the constructors for the two Point objects are executed before the constructor of Line is executing.
- And vise versa, when trying to remove a Line object from the memory, the destructor of Line is the last behavior of the lifetime of the Line object, after it is executed, the Line object is starting to remove from the memory; i.e. its components will be removing, so the two Point objects will be removing, then the destructor for these objects is calling for each one after destructor of Line destructor.

5.1.3.1 Constructor and destructor Chaining in case of embedded objects

- The constructor chaining of the an object which has an embedded object, the constructor of the embedded object is execute first and then the constructor of the container object.
- The destructor chaining of the an object has embedded another object, the destructor of the embedded object is execute after the destructor of the container object.

5.1.3.1 Constructor and destructor Chaining in case of embedded objects

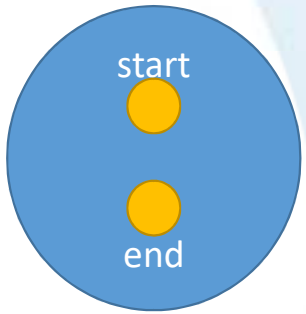
- The question is, which constructor of the embedded object is calling? Where it is not allowed to initialize any attributes inside the class. The answer is: the default constructor of the embedded objects which will be called.
- What if there is no default constructor for its class? Or if I want to let another constructor to be called instead of the default constructor.
- Yes we can, by making redirection to another constructor through the header of the constructor of the container object. In the above example, we make the redirection of the Point objects constructors through the header of the Line constructor, by butting ":" at the end of the header and write the name of embedded object with the parameters to select which constructor you want. As we show below:

```
Line(int x1, int y1, int x2, int y2) : start(x1, y1) , end(x2, y2)
{}
```

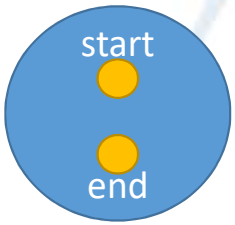
5.1.3 Embedded Objects as Strong Association

```
class Line
{
  private:
     Point start;
     Point end;
  public:
     Line() : start() , end()
     {
          //start.setXY(0,0);    end.setXY(0,0);
          cout<<"Line default constructor is calling"<<endl;
     }
```

Class Line

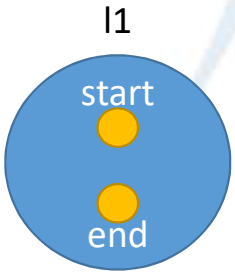


l1



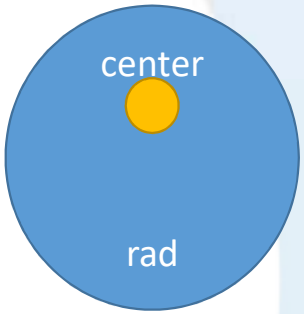
5.1.3 Embedded Objects as Strong Association

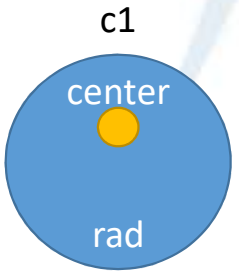
```
Line(int x1, int y1, int x2, int y2) : start(x1, y1), end(x2, y2)
{
     // start.setXY(x1,y1); end.setXY(x2,y2);
     cout<<"Line constructor with 4 parameters is calling"<<endl;
}
~Line(){cout<<"Line destructor is calling"<<endl;}
void print()
{
     cout<<"\nStart:"; start.print();
     cout<<"\nEnd:";   end.print();
}
};
```



5.1.3 Embedded Objects as Strong Association

```
class Circle
{
  private:
     Point center;
     int rad;
  public:
      Circle() : center(0)  {
          //center.setXY(0,0);
          rad = 0;
          cout<<"Circle default constructor is calling"<<endl;
     }
```

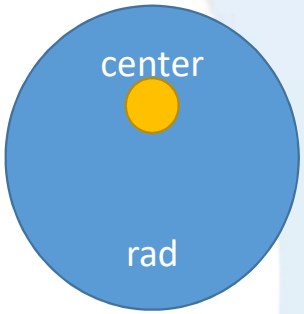




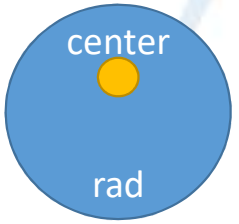
5.1.3 Embedded Objects as Strong Association

```
    Circle(int x1, int y1, int r) : center(x1, y1)
    {
        //  center.setXY(x1,y1);   rad = 0;
        cout<<"Circle constructor with 3 parameters is calling"<<endl;
    }
    ~Circle(){cout<<"Circle destructor is calling"<<endl;}
    void print()
    {
        cout<<"\ncenter:";     center.print();
        cout<<"\nRadius = "<<rad<<endl;
    }
};
```

Class Circle



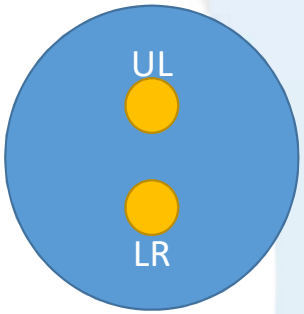
c1



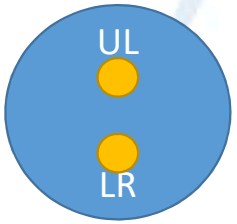
5.1.3 Embedded Objects as Strong Association

```
class Rect
{
  private:
     Point UL;
     Point LR;
  public:
     Rect() : UL() , LR()
     {
          //UL.setXY(0,0);  LR.setXY(0,0);
          cout<<"Rect default constructor is calling"<<endl;
     }
```

Class Rect



r1

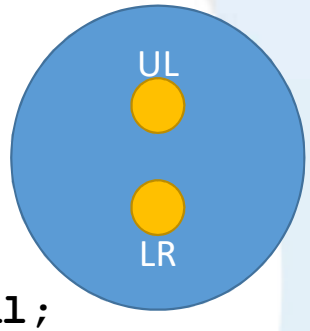


5.1.3 Embedded Objects as Strong Association

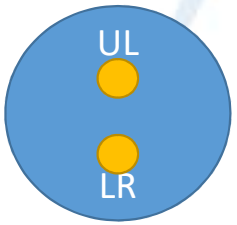
Java™ Education and Technology Services

```
Rect(int x1, int y1, int x2, int y2) : UL(x1, y1), LR(x2, y2)
{
     // UL.setXY(x1,y1);    LR.setXY(x2,y2);
     cout<<"Rect constructor with 4 parameter is calling"<<endl;
}
~Rect(){cout<<"Rect destructor is calling"<<endl;}
void print()
{
     cout<<"\nUpper Left:"; UL.print();
     cout<<"\nLower Right:";    LR.print();
}
};
```

Class Rect

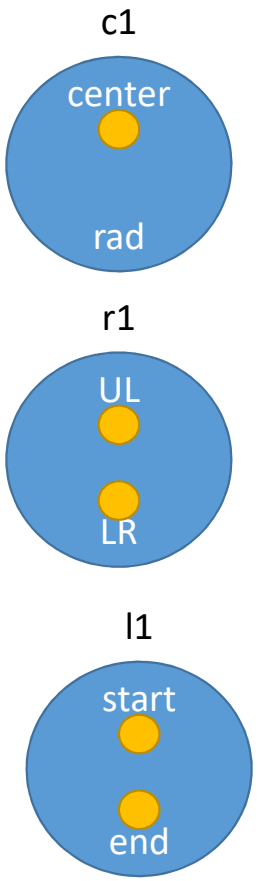


r1



5.1.3 Embedded Objects as Strong Association

```
int main()
{
     Circle c1(250,150,100) ;
     Rect r1(10,100,90,350) ;
     Line l1(30,100, 350, 400) ;

     c1.print() ;
     r1.print() ;
     l1.print() ;
     getch() ;
     return 0 ;
}
```



5.2 Collections of objects

- We can deal with objects like the structure, we may crate an array of objects, and we may make dynamic memory allocation with objects.
- We may initialize the array of objects. How?

5.2.1 Static Allocations : array of objects

- We may declare an array of objects exactly like declare array of structures.

```
Complex carr[10];
```

- The above line is declare an array of 10 Complex objects and each object call the default constructor
- We may call a different constructor for each object as we need:

```
Complex carr[3] = {Complex(2, 4), Complex(),
                                  Complex(8)};
```

- If we write constructors to some objects and not to all, the remaining will call the default constructor
- Like any array in C, we may not write the size in array declaration if we make initialization with constructors.

5.2.1 Static Allocations : array of objects

```
int main()
{
    Complex arr[3] = {Complex(2), Complex(), Complex(5,7)};
    for(int i = 0 , i<3 ; i++)
        arr[i].print();
    getch();
    return 0;
}
```

5.2.2 Dynamic Allocation: pointer to object

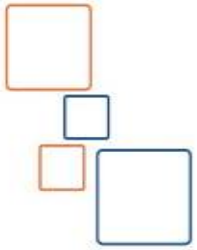
- We can use a pointer to object to make dynamic memory allocation with number of objects allocated in the heap memory and deal with them like array. The easy of using new operator in C++ make the dynamic memory allocation as piece of cake.

```
Complex * cptr;
cptr = new Complex(2.1, 7.3); // just allocate one Complex

Cptr = new Complex[12]; // allocate 12 complex contiguous
                        // but with default constructor
                        // only
```

5.2.2 Dynamic Allocation: pointer to object

```
int main()
{
    int n;
    Complex * cptr;
    cin>>n;
    cptr= new Complex[n]
    for(int i = 0 , i<n ; i++)
        cin>>cptr[i];
    for(i=0;i<n;i++)
        cptr[i].print();
    getch();
    return 0;
}
```

Lab Exercise

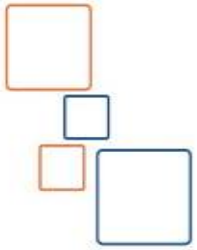


Invest In Yourself,
Develop Your Career

Assignments

- Example of Point, Rect, Circle, Line as strong association.
- Collections: Simple trials in the main( ) function to create object and array of objects from class Complex

Weak Association among Classes and Inheritance I



Invest In Yourself,
Develop Your Career

Content

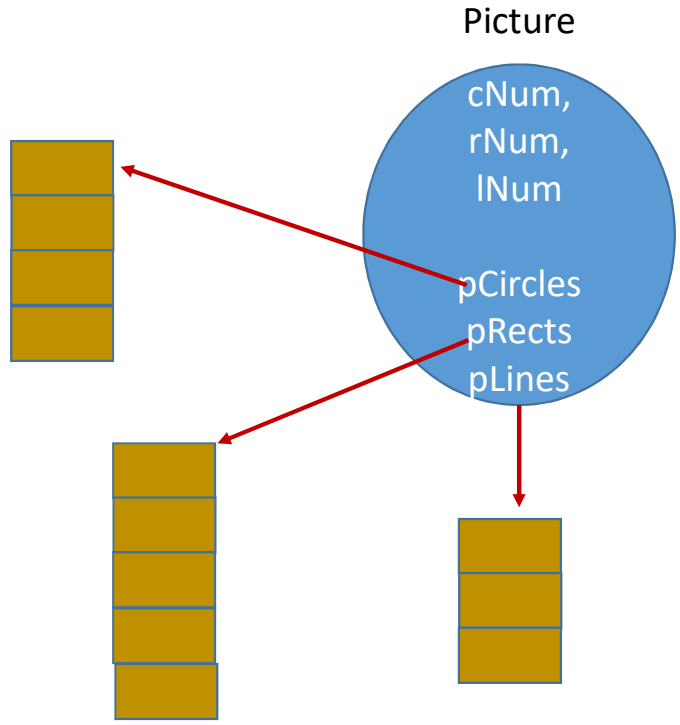
- Weak Association:
  - Example of: class Picture associated with Rect, Circle.
- Aggregation Vs Composition
- Inheritance
  - Inheritance is an "is-a" relationship.
  - Basic concepts of inheriting variables and methods.
  - Overriding.
  - The "protected" access specifier.
  - Simple Hierarchy Example on overriding and protected: the "calculateSum( )" program.

6.1 Weak Association: (Aggregation)

- It is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed.
- A type of whole-part relationship in which the component parts also exist as individual objects apart from the aggregate.

6.1.1 Example of Picture class with lines circles, and rect

```
class Picture
{
  private :
       int cNum ;
       int rNum ;
       int lNum ;
       Circle *pCircles;
       Rect *pRects;
       Line *pLines;
  public :
       Picture()
       {
              cNum=0;
              rNum=0;
              lNum=0;
              pCircles = NULL ;
              pRects = NULL ;
              pLines = NULL ;
       }
```



6.1.1 Example of Picture class with lines circles, and rect

```
    Picture(int cn, int rn, int ln, Circle *pC, Rect *pR, Line *pL)
    {
        cNum = cn;
        rNum = rn;
        lNum = ln;
        pCircles = pC ;
        pRects = pR ;
        pLines = pL ;
    }

    void setCircles(int, Circle *);
    void setRects(int, Rect *);
    void setLines(int, Line *);
    void print();
};
```

6.1.1 Example of Picture class with lines circles, and rect

```
void Picture::setCircles(int cn, Circle * cptr)
{
      cNum = cn ;
      pCircles = cptr ;
}

void Picture::setRects(int rn, Rect * rptr)
{
      rNum = rn ;
      pRects = rptr ;
}

void Picture::setLines(int ln, Line * lptr)
{
      lNum = ln ;
      pLines = lptr ;
}
```

6.1.1 Example of Picture class with lines circles, and rect

```
void Picture::print()
{
	int i;
	for(i=0; i<cNum ; i++)
	{
		pCircles[i].print();
	}

	for(i=0 ; i<rNum ; i++)
	{
		pRects[i].print();
	}

	for(i=0 ; i<lNum; i++)
	{
		pLines[i].print();
	}
}
```

6.1.1 Example of Picture class with lines circles, and rect

```
int main()
{
     Picture  myPic;
     Circle cArr[3]={Circle(50,50,50), Circle(200,100,100),
                                                Circle(420,50,30)};
     Rect rArr[2]={Rect(30,40,170,100), Rect(420,50,500,300)};
     Line lArr[2]={Line(420,50,300,300), Line(40,500,500,400)};
     myPic.setCircles(3,cArr) ;
     myPic.setRects(2,rArr) ;
     myPic.setLines(2,lArr) ;
     myPic.print() ;
     getch();
     return 0;
}
```

6.1.1 Example of Picture class with lines circles, and rect

```
int main()
{
     Circle cArr[3]={Circle(50,50,50), Circle(200,100,100),
                                                Circle(420,50,30)};
     Rect rArr[2]={Rect(30,40,170,100), Rect(420,50,500,300)};
     Line lArr[2]={Line(420,50,300,300), Line(40,500,500,400)};
     {
          Picture p2(3, 2, 2, cArr, rArr, lArr);
          p2.print();
     }
     getch();
     return 0;
}
```

6.1.1 Example of Picture class with lines circles, and rect

```
int main()
{     Picture   myPic;
      //example on static allocation
      Circle cArr[3]={Circle(50,50,50), Circle(200,100,100),
                                                Circle(420,50,30)};
      //example on static allocation, using temporary objects (on the fly)
      Rect rArr[2] ;
      rArr[0] = Rect(30,40,170,100) ;
      Point myP1(420,50) ;
      Point myP2(500,300) ;
      rArr[1] = Rect(myP1, myP2) ;
```

6.1.1 Example of Picture class with lines circles, and rect

```
    //example on dynamic allocation, using temporary objects (on the fly)
    Line * lArr ;
    lArr = new Line[2] ;
    lArr[0] = Line(Point(420,50) , Point(300,300)) ;
    lArr[1] = Line(40,500,500,400) ;
    myPic.setCircles(3,cArr) ;
    myPic.setRects(2,rArr) ;
    myPic.setLines(2,lArr) ;
    myPic.print() ;
    delete[] lArr ;
    return 0;
}
```

6.2 Aggregation vs Composition

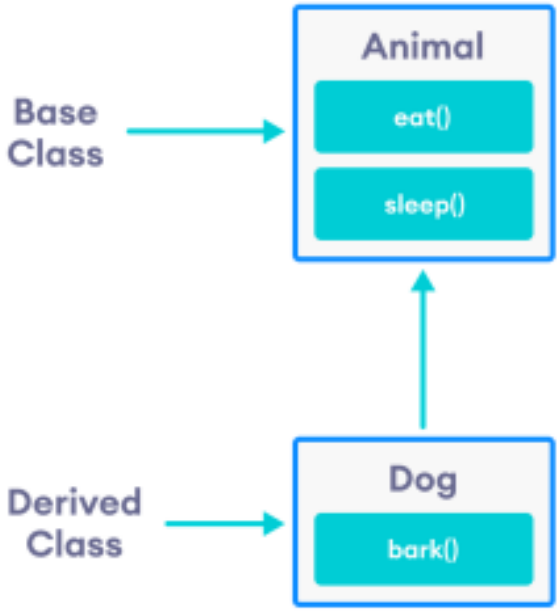
- **Dependency:** Aggregation implies a relationship where the included object **can exist independently** of the container object. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the included object **cannot exist independent** of the container object. Example: Human and heart, heart don't exist separate to a Human
- **Type of Relationship:** Aggregation relation is **"has-a"** and composition is **"part-of"** relation.
- **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

6.3 Inheritance: is-a

- *Inheritance* is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).
- The derived class inherits the features from the base class and can have additional features of its own.
- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
  Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.
- It is a relationship between classes where one class is a *parent (super or base)* of another *(Child, subclass, derived).* It implements *"is-a"* relationships between objects. Inheritance takes advantage of the commonality among objects to reduce complexity.

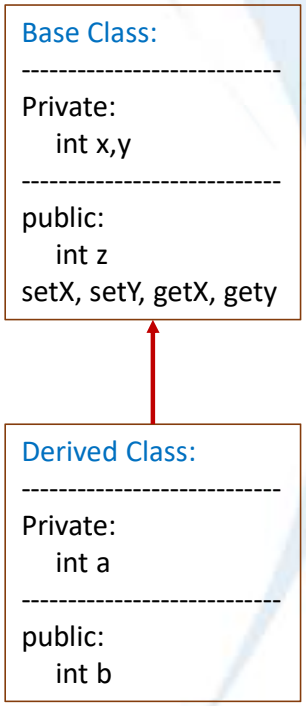
6.3.1 Advantages of Inheritance

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



6.3.2 Basic concepts of inheriting variables and methods.

```
int main()
{
     Derived d;
     d.b=5;
     d.a=7;              // Not Accessible
     d.z=12;             // It is ok
     d.x=-13;            // Not Accessible
     d.setX(12);         // It is ok
     cout<<d.getY()      // It is ok
}
```



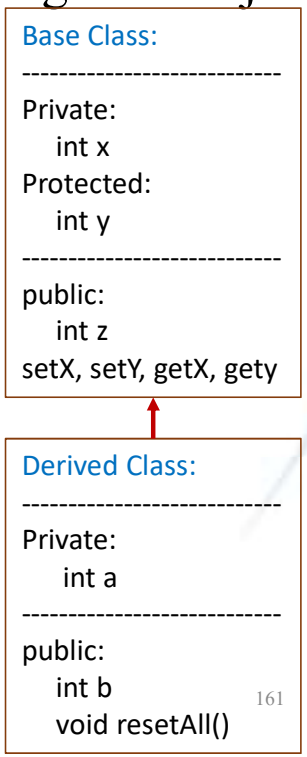
- If a derived class need to access an inherited member which was private in a parent class, it is not allowed, it has to use the inherited public members which access the private members. If a base class want to let its derived classes to access its private members directly, it has to put those members in ***protected*** section.

6.3.3 Protected Section in Class

- The protected members, which declared in protected section, it is like private members in the same class, the different in the case of inheritance where the child class can deal with it directly –by members of the child class not through the objects of the child class- without need to use the public members.

```
void resetAll(){
    x=0;    // Still not accessible use setX()
    y=z=0;  // it is ok
    a=b=0;
}
```

- But still x,y, and a not accessible though using Derived class object.



6.3.4 The Syntax of Inheritance to make a Child Class

```
class Child_class_name : access-mode Parent_class_name
{
    // body of the derived class.
}
```

- Where,
  - **Child_class_name:** It is the name of the derived class.
  - **Access mode:** The access mode specifies whether the features of the base class are publicly inherited, protected inherited, or privately inherited. It can be public, protected, or private.
  - **Parent_class_name:** It is the name of the base class.

6.4 Modes of Inheritance

Parent Sections / Inheritance Mode	Private	Protected	Public
Private	Not Accessible	Private	Private
Protected	Not Accessible	Protected	Protected
Public	Not Accessible	Protected	Public

- The most cases of inheritance is public mode (more than 95%)
- The protected mode is using little.
- The using of private mode is rare, because it is closed the accessibility of the base class after the direct children level –including the public members-

6.5 Polymorphism: Overriding

- If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.
- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.
- **Overriding**: A *method* defined in a *superclass* may be overridden by a *method* of the same name defined in a *subclass*. The two *methods* must have the **same name** and **number and types of formal input parameters**.

6.5 Polymorphism: Overriding

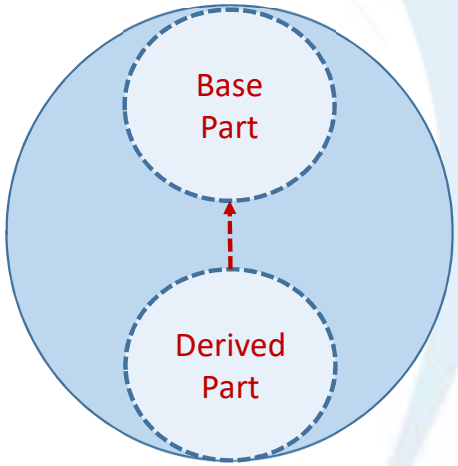
```
class Base {
public:
   void disp(){
      cout<<"Function of Parent Class";
   }
};
class Derived : public Base{
public:
   void disp() {
      cout<<"Function of Child Class";
   }
};
int main() {
   Derived obj;              // It execute the Base version except if you make
   obj.disp();               // overriding it will print the Derived version.
   obj.Base::disp();         // You may call the function of the base using
                             // the name of the base class with scope operator
}
```

6.6 Multi-Level Inheritance

- In Multilevel Inheritance a derived class can also inherited by another class.
- In object-Oriented Paradigm, classes are organized into a singly rooted tree structure, called the *inheritance hierarchy*.
- **Inheritance Hierarchy**: The relationship between *super classes* and *subclasses* is known as an inheritance hierarchy.
- The words Super Class, Parent Class, and Base Class are represent the same thing.
- The words Sub Class, Child Class, and Derived Class are represent the same thing.

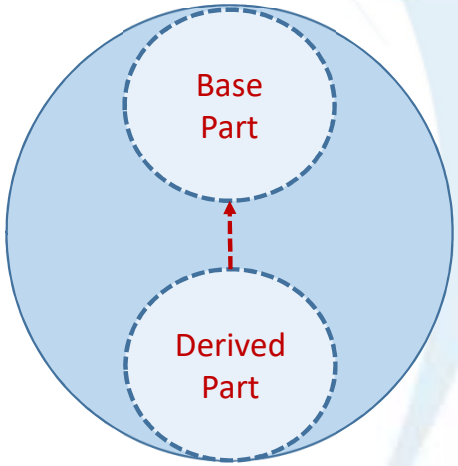
6.7 Constructor and Destructor Chaining in case of Inheritance

- Any object from the derived class consists of two parts: the inherited part from the base class, and the second part is the added members by the derived class.
- Each part has its initial behavior – constructor- the inherited part call the constructor of the base class after it is created, and the other part call the derived constructor after it is created.
- In the inheritance, the constructors of the base and of the derived are calling in sequential, but in which order?
- The base part is created first, then the constructor of the base class will calling first, and then the constructor of the derived class is calling after that.



6.7 Constructor and Destructor Chaining in case of Inheritance

- You may redirect which constructor of the base class is calling, by write as extension to the header of the constructor of the derived class using ":" and select which constructor you want from the base class using the name of the base class.
- The order of the destructor chaining will be the reverse order of the constructor chaining, i.e. the destructor of the derived class will be calling first and then the destructor of the base class will be calling after that



6.8 Simple Example of Inheritance

```
class Base
{
  protected: //first, the student should try it as private
      int a ;
      int b ;
  public:
      Base()
      { a=b=0 ; }

      Base(int n)
      { a=b=n ; }

      Base(int x, int y)
      { a = x ; b = y ; }

      void setA(int x)
      { a = x ; }
```

6.8 Simple Example of Inheritance

```
    void setB(int y)
    { b = y ; }

    int getA()
    { return a ; }

    int getB()
    { return b ; }

    int calculateSum()
    {
        return a + b ;
    }
};
```

6.8 Simple Example of Inheritance

```
class Derived : public Base
{
  private:
    int c ;
  public:
    Derived() : Base()
    { c = 0 ; }

    Derived(int n) : Base(n)
    { c = n ; }

    Derived(int x, int y, int z) : Base(x,y)
    { c = z ; }
```

6.8 Simple Example of Inheritance

```
    void setC(int z)
    { c = z ; }

    int getC()
    { return c ; }

    int calculateSum()  //overriding
    {
        return a + b + c ;  // only if "a" and "b" were protected,
                            // or use getters if it is private
        //  other implementation ideas:
        //  return Base::calculateSum() + c ;
    }
};
```

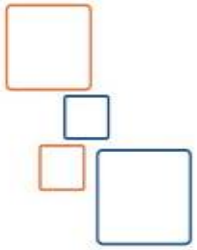
6.8 Simple Example of Inheritance

```
int main()
{
    clrscr() ;
    Base b(5,4) ;
    cout<<b.calculateSum()<<endl ;
    Derived obj1 ;
    obj1.setA(3) ;
    obj1.setB(7) ;
    obj1.setC(1) ;
    Derived obj2(20) ;
```

6.8 Simple Example of Inheritance

```
    Derived obj3(4,5,6) ;
    cout<<"obj1: "<<obj1.calculateSum()<<endl ; // =11
    cout<<"obj2: "<<obj2.calculateSum()<<endl ; // =60
    cout<<"obj3: "<<obj3.calculateSum()<<endl ; // =15
cout<<"obj1: "<<obj1.Base::calculateSum()<<endl ; //only =10
    getch() ;

return 0 ;
}
```

Lab Exercise

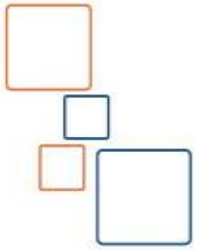


Assignments

- Weak Association Example: Picture, Point, Rect, Circle, Line
- Example of: Base, Derived, and calculateSum( ).

- Note: Students should Trace the code by F7 or F8 to see the sequence of calling of constructors of Base and Derived.

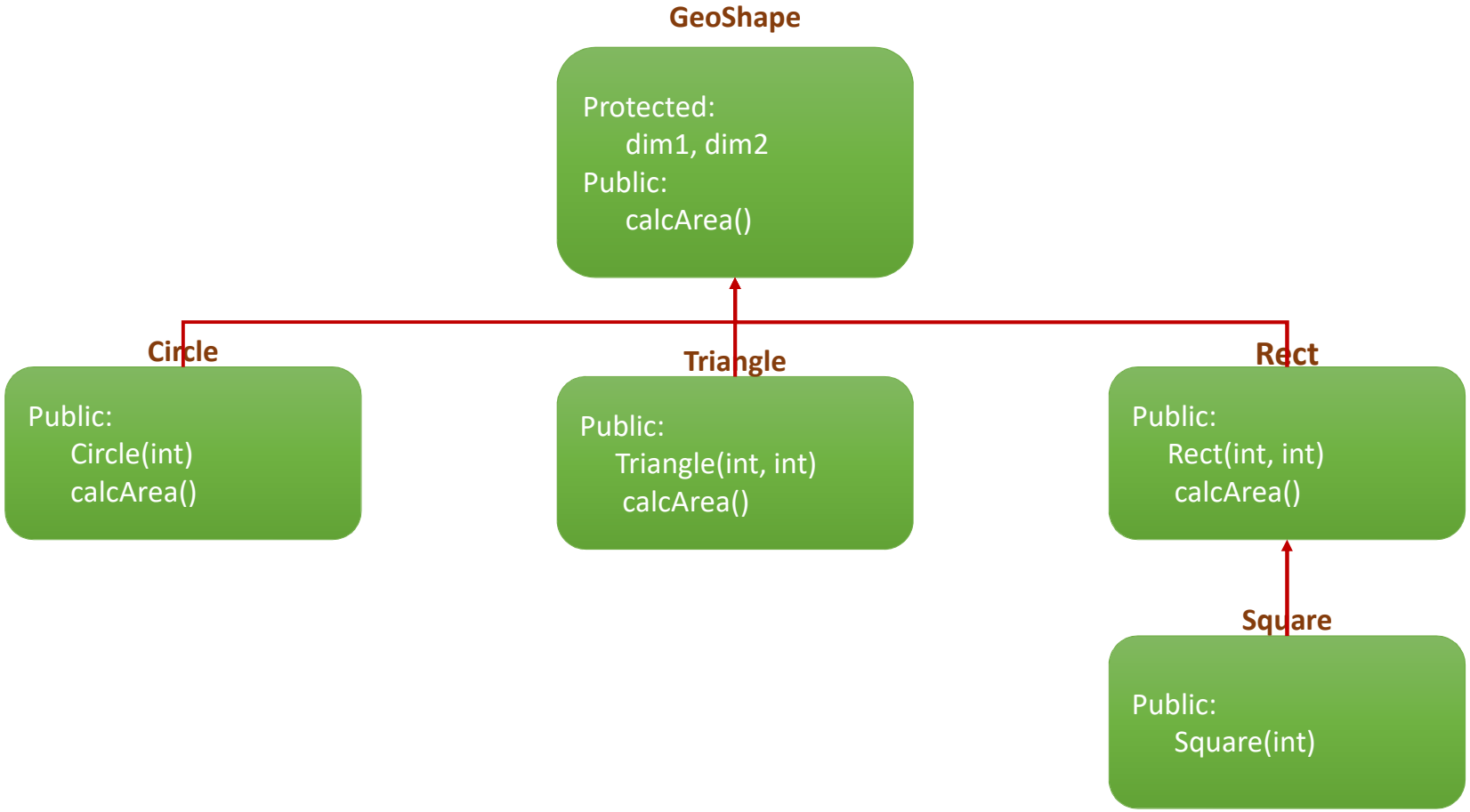
Inheritance II



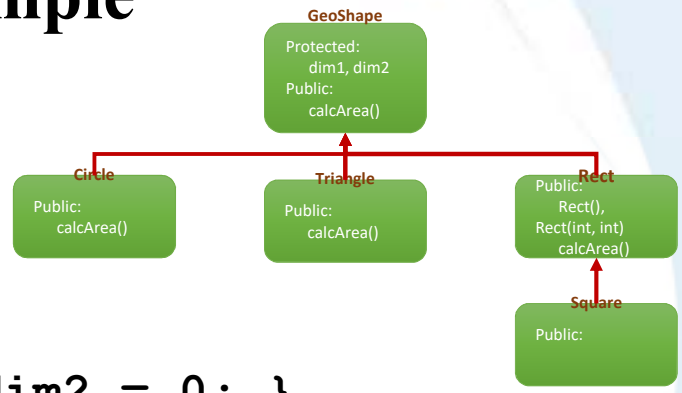
Content

- The GeoShape class hierarchy: GeoShape, Circle, Triangle, Rectangle, Square).
  - The problem of violating the square constraint.
  - Modification of GeoShape class hierarchy (protected inheritance).
- Dynamic Binding and Virtual Function
  - Example of Dynamic Binding: sumOfAreas(, ,) as stand alone function

7.1 The GeoShape Class Hierarchy Example

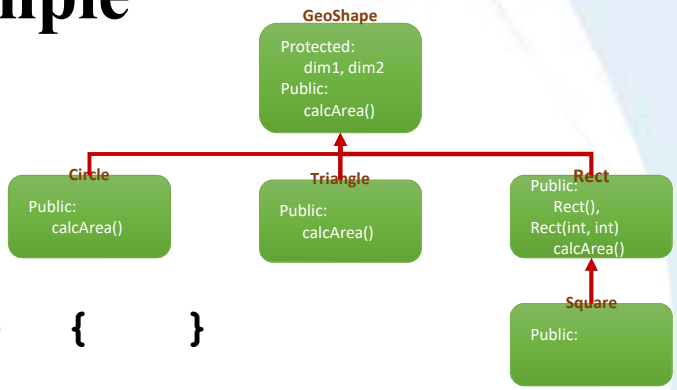


7.1 The GeoShape Class Hierarchy Example



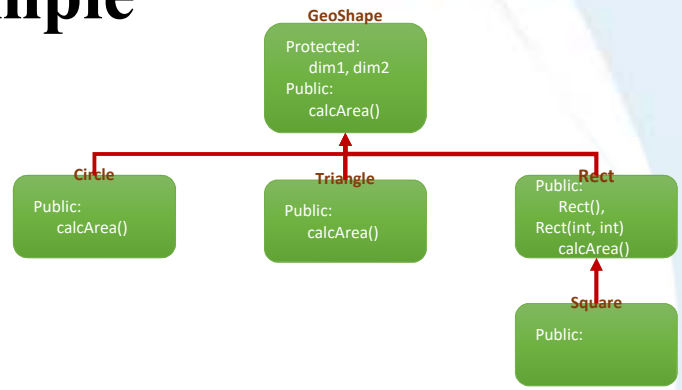

```
class GeoShape
{  protected:
     float dim1, dim2;
   public:
     GeoShape()                      { dim1 = dim2 = 0; }
     GeoShape(float x)               { dim1 = dim2 = x; }
     GeoShape(float x, float y)      { dim1 = x;   dim2 = y;  }
     void setDim1(float x)           { dim1 = x; }
     void setDim2(float x)           { dim2 = x; }
     float getDim1()                 { return dim1; }
     float getDim2()                 { return dim2; }
     float calcArea()                { return 0.0;  }
};
```

7.1 The GeoShape Class Hierarchy Example



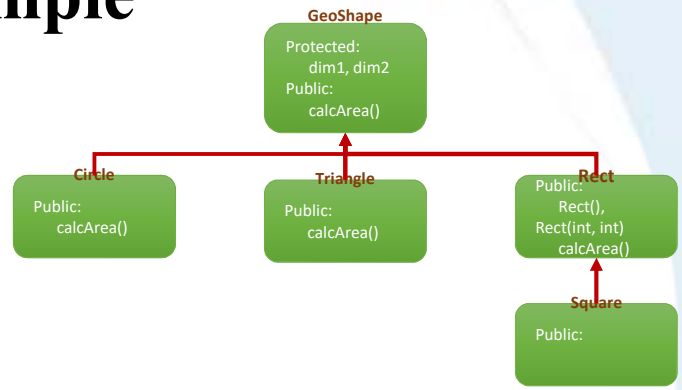

```
class Rect: public GeoShape
{  public:

    Rect(float x, float y) : GeoShape(x, y)  {    }
    float calcArea()
    {
        return dim1 * dim2;
    }
};
class Square: public Rect
{  public:
    Square(float x) : Rect(x, x) {    }
};
```

7.1 The GeoShape Class Hierarchy Example




```
class Triangle : public GeoShape
{  public:
     Triangle(float b, float h):GeoShape(b, h){ }
     float calcArea()
     {
          return 0.5 * dim1 * dim2;
     }
};
class Circle : public GeoShape
{  public:
     Circle(float r) : GeoShape(r)   {    }
     float calcArea()
     {                                        //it's ok since dim1 equals to dim2.
          return 22.0/7 * dim1 * dim2; //you may write:  22.0/7*dim1*dim1.
     }
};
```

7.1 The GeoShape Class Hierarchy Example



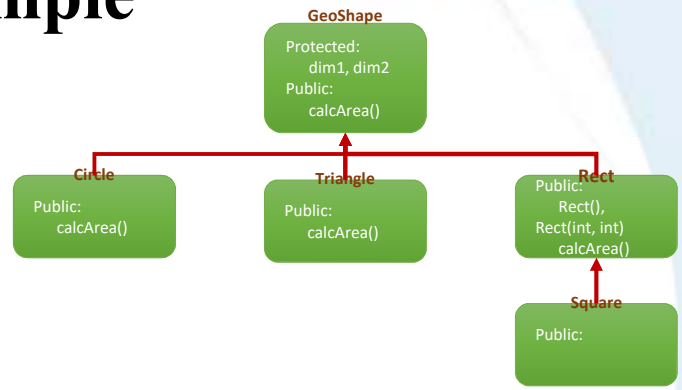

```
int main()
{    Triangle myT(20, 10);
     cout << myT.calcArea() << endl;
     Circle myC(7);
     cout << myC.calcArea() << endl;
     Rect myR(2, 5);
     cout << myR.calcArea() << endl;
     Square myS(5);
     cout << myS.calcArea() << endl;
     //What happened if you try:
     myS.setDim2(4) ; //Violating the Square Constraint
     myC.setDim2(3) ; //Violating the Circle Constraint
     getch() ;
     return 0;
}
```

7.1 The GeoShape Class Hierarchy Example

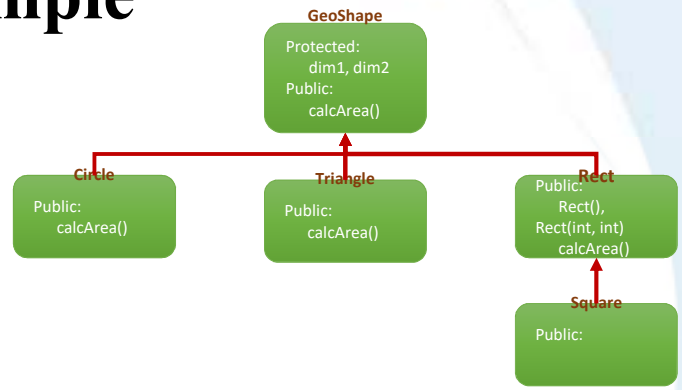
- In the above example there are cases *violate* the constraint of the Square logic; the two dimensions are the same equal to the *side length*. And so the constraint of the circle logic; where the two dimensions are the same equal to the *radius*.
- This happened because the public inheritance of Square class from Rect class, where the setters *setDim1* and *setDim2* is accessible inside the Square class and so through an object from the Square class. How this logical problem can be solving?
- You may ***override*** the two setters *setDim1* and *setDim2* inside the Square class to do the same thing: change in the two dimensions together. But this solution may confusing the user of the class.
- Other better solution, make the inheritance mode to be ***protected*** mode when you inherit the Square class from the Rect class, and make a public members with name *setSide(int)* which set the two dimensions with the same value. But you have to override the *calcArea()* in class Square.
- Do the same thing with the Circle class.

7.1 The GeoShape Class Hierarchy Example

```
class Square: protected Rect
{  public:
    Square(float x) : Rect(x, x)   {    }
    void setSide(int length)
    {
        dim1 = dim2 = length;
    }
    float calcArea()
    {
        return dim1 * dim2;
    }
};
```



7.1 The GeoShape Class Hierarchy Example

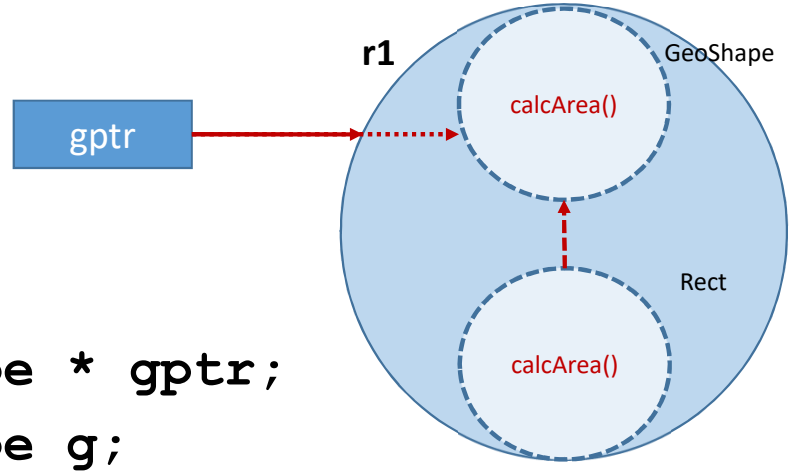


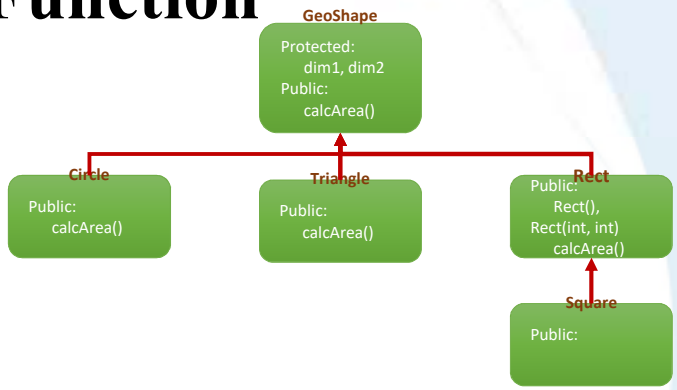

```
class Circle : protected GeoShape
{  public:
     Circle(float r) : GeoShape(r)
     {    }
     void setRadius(float rad)
     {
          dim1 = dim2 = rad;
     }
     float calcArea()
     {                                   //it's ok since dim1 equals to dim2.
          return 22.0/7*dim1*dim2;//you may write:  22.0/7*dim1*dim1.
     }
};
```

7.2 Dynamic Binding and Virtual Function

- Dynamic binding also called dynamic dispatch is the process of linking method call to a specific sequence of code at run-time. It means that the code to be executed for a specific method call is not known until run-time.
- The way of perform the dynamic binding using the ***Virtual Function***.
- A virtual function is a member function that you expect to be redefined in derived classes. 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. It is the mechanism of ***Dynamic Binding*** concept.
- For example: the function calcArea() which is defined as a members in class GeoShape, the expectation to be overridden is near to be 100%, so we have to make it virtual, by add the reserved word ***"virtual"*** before the returned data type of the header of the function as a function modifier.

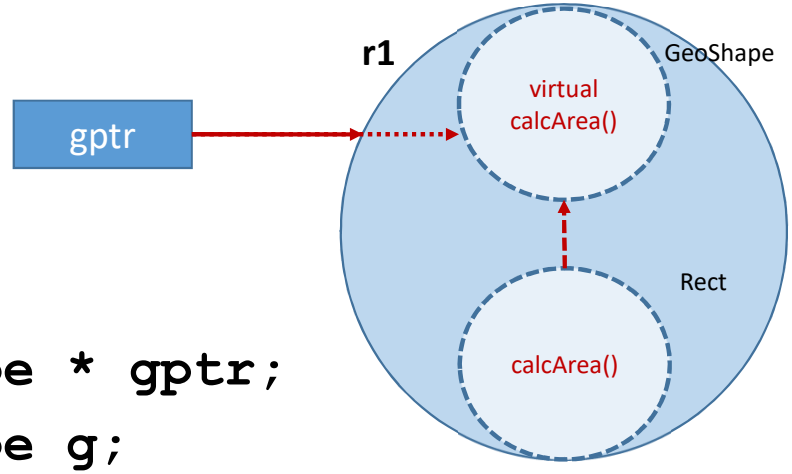
7.2 Dynamic (late) Binding and Virtual Function

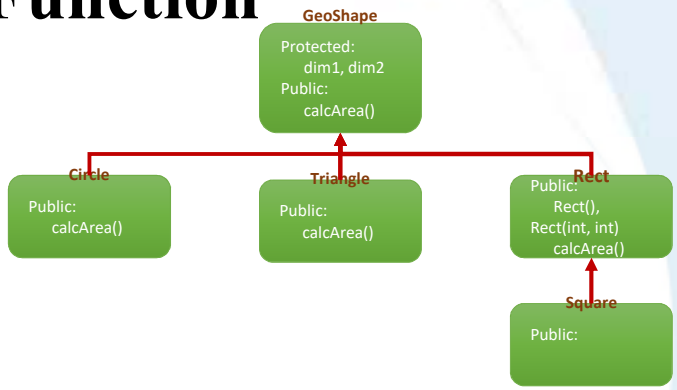





```
Geoshape * gptr;
Geoshape g;
Rect r1;
gptr = & g;
cout<<gptr->calcArea();
gptr = & r1;
cout<<gptr->calcArea(); //the GeoShape version is called
```

7.2 Dynamic (late) Binding and Virtual Function






```
Geoshape * gptr;
Geoshape g;
Rect r1;
gptr = & g;
cout<<gptr->calcArea();
gptr = & r1;
cout<<gptr->calcArea(); //the Rect version if it is virtual
```

7.2 Dynamic (late) Binding and Virtual Function

```
class GeoShape
{  public:

    virtual float calcArea()
    {
        return 0.0;
    }
};
```

7.3 Example of Dynamic Binding: sumAreas(, ,)

- If we need a function to make the sum of any three objects from type of GeoShape – from all of its children-, it is an example of using of the dynamic binding.
- This function will take three object by their address, the calling it will be by address and it will send to pointer to GeoShape.

```
float sumAreas(GeoShape * p1, GeoShape * p1, GeoShape * p3)
{
    return p1->calcArea()+p2->calcArea()+p3->calcArea();
}
```

- This function will execute on any three objects from type GeoShape; i.e. any objects from GeoShape or any objects from children of GeoShape.
- When you call this function, you may send three object from Rect, or send 1 from Circle and 2 from Square, and so on, the version pf the calcArea will be dedicated the run time according to the type of the object.
- Note: Make this function as stand alone function.

7.3 Example of Dynamic Binding: sumAreas(, ,)

```
int main()
{
    Rect r1(1,2,3,4), r2(5,10,15,20);
    Square s1(10), s2(20), s3(30);
    Circle c1(30);
    Triangle t1(10,20), t2(20, 40);
    cout<<sumAreas(&s1, &s2, &s3);
    cout<<sumAreas(&t1, &c2, &r2);
    cout<<sumAreas(&r1, &s2, &t2);
    getch();
    return 0;
}
```

7.3 Example of Dynamic Binding: sumAreas(, ,)

• The version of the function using references:

```
float sumAreas(GeoShape & r1, GeoShape & r1, GeoShape & r3)
{
    return r1.calcArea()+r2.calcArea()+r3.calcArea();
}
```

7.3 Example of Dynamic Binding: sumAreas(, ,)

```
int main()
{
    Rect r1(1,2,3,4), r2(5,10,15,20);
    Square s1(10), s2(20), s3(30);
    Circle c1(30);
    Triangle t1(10,20), t2(20, 40);
    cout<<sumAreas(s1, s2, s3);
    cout<<sumAreas(t1, c2, r2);
    cout<<sumAreas(r1, s2, t2);
    getch();
    return 0;
}
```

7.4 Pure Virtual Function and Abstract Class

- **Abstract Method (Pure Virtual Function)** is a *method* that is declared, but contains no implementation, without body. It is needed to be overridden by the subclasses for standardization or generalization.
- **Abstract Class** is distinguished by the fact that you may not directly construct objects from it. An abstract class may have one or more *abstract methods*. Abstract classes may not be instantiated, and require *subclasses* to provide implementations for the abstract methods. It is needed for standardization or generalization.
- **Concrete Class**: Any class can be initiating objects directly from it, it is not abstract class.
- For Example: the calcArea method of the GeoShape class it must be abstract method, so the class GeoShape will be an abstract class.
- The Abstract method (or pure virtual function in C++) conists of header only without body, the header start with word "***virtual***" and end with "***= 0***".

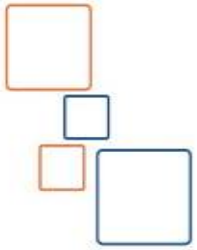
7.4 Pure Virtual Function and Abstract Class

```
class GeoShape
{
    public:
    …
    virtual float calcArea() = 0; // pure virtual function
    …


};
```

7.4 Pure Virtual Function and Abstract Class

```
int main(){
    GeoShape gobj;      // Compilation error can not create
                        // an object from abstract class
    GeoShape * gptr;    // It is allowed to point to any
                        // object from any children classes
```

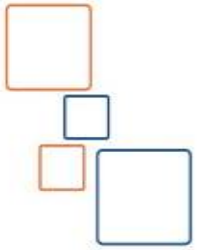
Lab Exercise



Assignments

- GeoShape Example:
  - Try with public inheritance (try to violate the square and circle constraints by calling the inherited setter methods).
  - Try with protected inheritance (not able to violate the square and circle constraints).
  - Add the new special setter functions to class square and circle.
- Write the standalone function: "sumOfAreas(~, ~, ~), which takes 3 parameters as pointers from type GeoShape which return the summation of objects' areas.
- Make the function: "calculateArea( )", a pure virtual function, and make necessary changes to other classes.
  - Not able to create objects of the abstract class: GeoShape, but it is allowed to create pointers to GeoShape.

Multiple Inheritance, Template Class and Introduction to UML



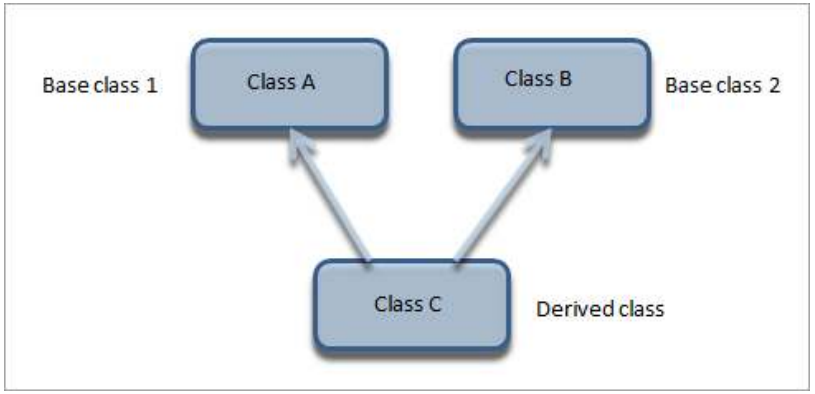
Invest In Yourself,
Develop Your Career

Content

- Multiple Inheritance
  - Duplicate in members names from the direct parents' classes problem
  - Diamond problem
- Template in C++ (Generic)
  - Template Function
  - Template Class

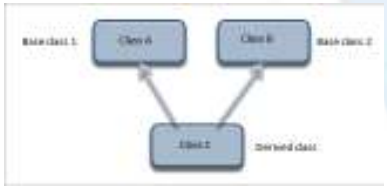
8.1 Multiple Inheritance

- The ability of a class to extend more than one class; i.e. A *class* can inherit characteristics and features from more than one parent class.



- Multiple inheritance is a type of inheritance in which a class derives from more than one classes. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.
- In a real-life scenario, a child inherits from its father and mother. This can be considered as an example of multiple inheritance.

8.1 Multiple Inheritance

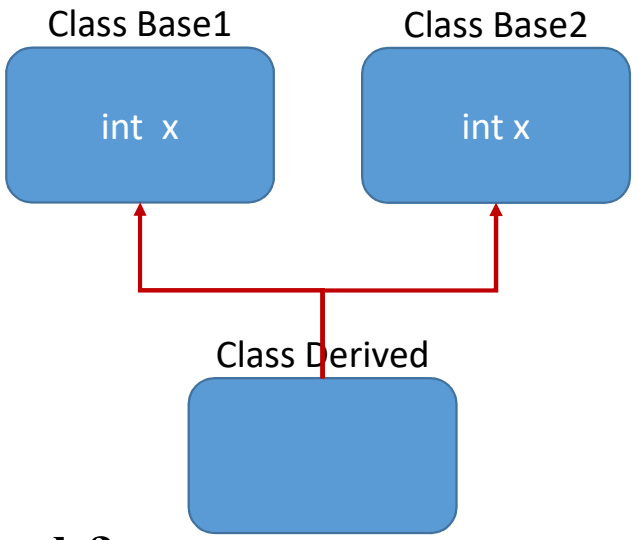


```
class Base1
{}
class Base2
{}
class Derived: public Base1, Base2
{}
```

- All the rules of the inheritance is applied.
- But there are some problems may be faced with multiple inheritance, like if we have two members in the two base classes have the same name, it is solved by using the scope operator (::) with the name of one of the base classes to select which one you want to access. But what if this variable was inherited before to the two base classes from common parent for them?. It is called the ***Diamond Problem***.

8.1.1 Duplicate members name from different base classes

```
Derived d;
d.X = 10; // ambiguity which x?
          // to solve that using ::
d.Base1::x = 10;
d.Base2::x = 20;
…
```



- But what if this member in the two base classes is the same one; has the same original from above levels?
- That what we call, “Diamond Problem”

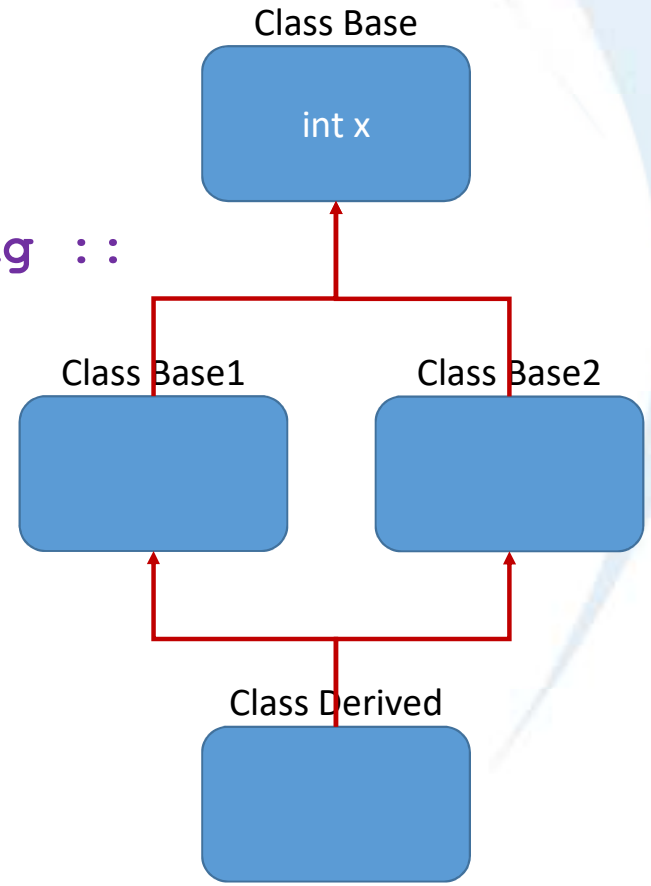
8.1.2 Diamond Problem

```
Derived d;
d.X = 10;              // ambiguity which x?
                       // to solve that using ::
d.Base1::x = 10;       // Not Meaningful
d.Base2::x = 20;       // Not Meaningful
```

…

- The solve by make Base1 and Base2 inherit from class base virtual, how?

```
class Base1: virtual public Base {}
class Base2: public virtual Base {}
```



8.2 Templates in C++ (Generic)

- Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.
- Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.
- The concept of templates can be used in two different ways:
  - Function Templates
  - Class Templates
- C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

8.2.1 Function Template

- We write a generic function that can be used for different data types. It is similar to a normal function, with one key difference; a single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.
- However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.
- The general form of a template function definition is shown here:

```
template <class type>
return-type function-name(parameter list) {
        // body of function
}
```

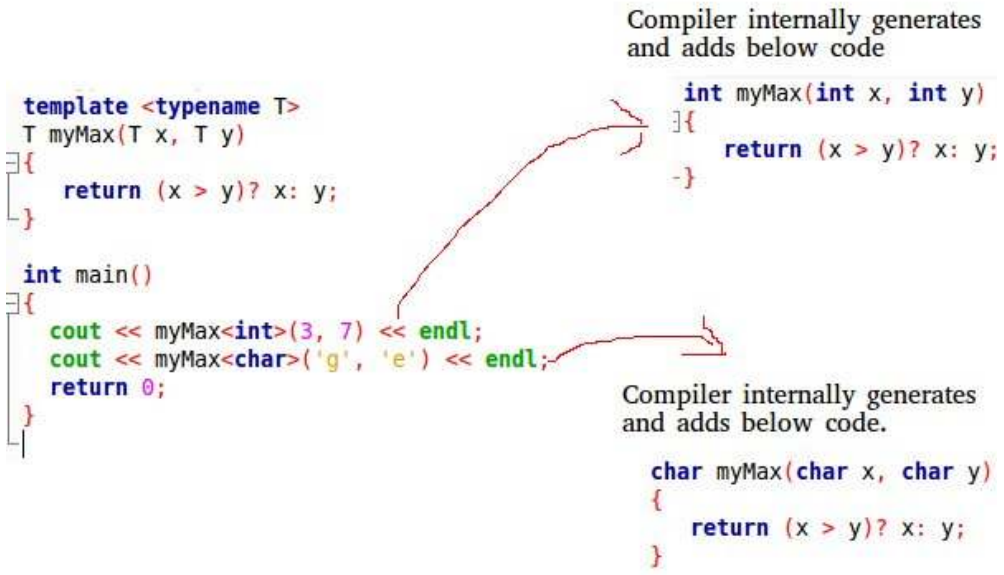
8.2.2 Class Template

- You can create class templates for generic class operations. Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.
- This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
- However, class templates make it easy to reuse the same code for all data types.
- The general form of a generic class declaration is shown here:

```
template <class type>
class class-name {
    .
    .
}
```

8.2.3 How templates work?

- Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



8.2.4 Stack Class as Generic Class

```
template <class T>
class Stack
{  private:
      int top, size;
       T *ptr;
      static int counter ;
  public:
      Stack()
      {     top = 0 ;
            size = 10;
            ptr = new T[size];
            counter++ ;
      }
      Stack(int n)
      {     top = 0;
            size = n;
            ptr = new T[size];
            counter++ ;
      }
```

8.2.4 Stack Class as Generic Class

```
    ~Stack()
    {
        delete[] ptr;
        counter-- ;
    }
    static int getCounter()
    {
        return counter ;
    }
    Stack(Stack & s) ;
    int push(T);
    int pop(T & n);
    Stack& operator= (Stack& s);
    friend void viewContent(Stack s) ;
};
```

8.2.4 Stack Class as Generic Class

```
//static variable initialization
template <class T>
int Stack< T >::counter = 0 ;

template <class T >
Stack<T>::Stack(Stack<T> & myStk)
{
     top = myStk.top;
     size = myStk.size;
     ptr = new T[size];
     for (int i=0 ; i<top ; i++)
     {
          ptr[i] = myStk.ptr[i];
     }
     counter++ ;
}
```

8.2.4 Stack Class as Generic Class

```
template <class T>
int Stack<T>::push(T n)
{
    if (isFull())
    {
        return 0;
    }
    else
    {
        ptr[top] = n;
        top++;
        return 1;
    }
}
```

8.2.4 Stack Class as Generic Class

```
template <class T>
int Stack<T>::pop(T & data)
{
    if (isEmpty())
    {
        return 0;
    }
    else
    {
        top--;
        data = ptr[top];
        return 1 ;
    }
}
```

8.2.4 Stack Class as Generic Class

```
template <class T>
Stack<T>& Stack<T>::operator= (Stack<T>& myS)
{
    if(ptr)     delete[] ptr;
    top = myS.top;
    size = myS.size;
    ptr = new T[size];
    for (int i = 0; i < top; i++)
    {
        ptr[i] = myS.ptr[i];
    }
    return *this;
}
```

8.2.4 Stack Class as Generic Class

```
template <class T>
void viewContent(Stack<T> myS)
{
    for(int i=0 ; i<myS.top ; i++)
    {
        cout<<myS.ptr[i]<<endl ;
    }
}
```

8.2.4 Stack Class as Generic Class

```
int main()
{   int n;
    clrscr();
    Stack<int> s1(5);
cout<<"\nNo of Int Stacks is:"<<Stack<int>::getCounter();
    s1.push(10);
    s1.push(3);
    s1.push(2);
    s1.pop(n);
    cout << "\n1st integer: " <<n;
    s1.pop(n);
    cout << "\n2nd integer: " <<n;
```

8.2.4 Stack Class as Generic Class

```
    Stack<char> s2;
    char nc;
cout <<"\nNo of Char Stacks is:"<<Stack<char>::getCounter();
    s2.push('q');
    s2.push('r');
    s2.push('s');
    viewContent(s2) ;
    s2.pop(cn);
    cout << "\n1st character: " <<nc;
    s2.pop(nc);
    cout << "\n2nd character: " <<nc;
    getch() ;
return 0;
}
```

8.3 Introduction to UML

- **What is UML?**
  - The Unified Modeling Language (UML) is the standard modeling language for software and systems development.
  - Modeling helps you to focus on, capture, document, and communicate the important aspects of your system's design.
  - A modeling language can be made up of pseudo-code, actual code, pictures, diagrams, or long passages of description; in fact, it's pretty much anything that helps you describe your system.
  - The elements that make up a modeling language are called its *notation* (a way of expressing the model).
  - A modeling language can be anything that contains a notation and a description of what that notation means.

8.3 Introduction to UML

- **UML Advantages**
  - **It's a formal language**: Each element of the language has a strongly defined meaning, so you can be confident that when you model a particular facet of your system it will not be misunderstood.
  - **It's concise**: The entire language is made up of simple and straightforward notation.
  - **It's comprehensive**: It describes all important aspects of a system.
  - **It's scalable**: Where needed, the language is formal enough to handle massive system modeling projects, but it also scales down to small projects.
  - **It's built on lessons learned**: UML is the result of best practices in the object-oriented community during the decades.
  - **It's the standard**: UML is controlled by an open standards group with active contributions from a worldwide group of vendors and academics. (*http://www.omg.org*)

8.3 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
Use Case	Interactions between your system and users or other external systems. Also helpful in mapping requirements to your systems.	UML 1.x
Activity	Sequential and parallel activities within your system.	UML 1.x
Class	Classes, types, interfaces, and the relationships between them.	UML 1.x
Object	Object instances of the classes defined in class diagrams in configurations that are important to your system.	Informally UML 1.x
Sequence	Interactions between objects where the order of the interactions is important.	UML 1.x
Communication	The ways in which objects interact and the connections that are needed to support that interaction.	Renamed from UML 1.x's collaboration diagrams

8.3 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
Timing	Interactions between objects where timing is an important concern.	UML 2.0
Interaction Overview	Used to collect sequence, communication, and timing diagrams together to capture an important interaction that occurs within your system.	UML 2.0
Composite Structure	The internals of a class or component, and can describe class relationships within a given context.	UML 2.0
Component	Important components within your system and the interfaces they use to interact with each other.	UML 1.x, but takes on a new meaning in UML 2.0
Package	The hierarchical organization of groups of classes and components.	UML 2.0

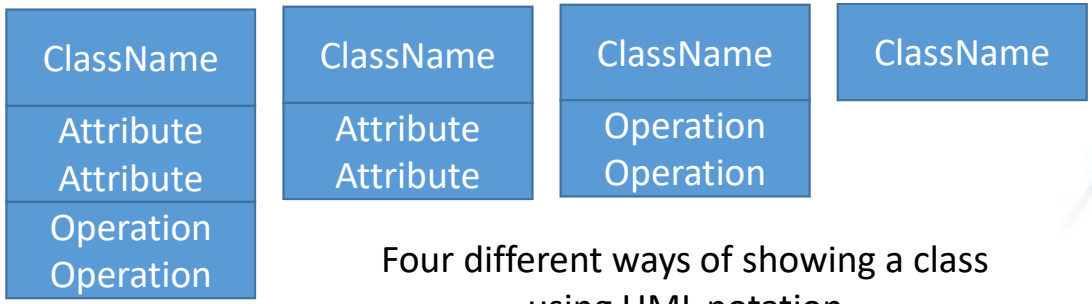
8.3 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
State Machine	The state of an object throughout its lifetime and the events that can change that state.	UML 1.x
Deployment	How your system is finally deployed in a given real world situation.	UML 1.x

8.3 Introduction to UML

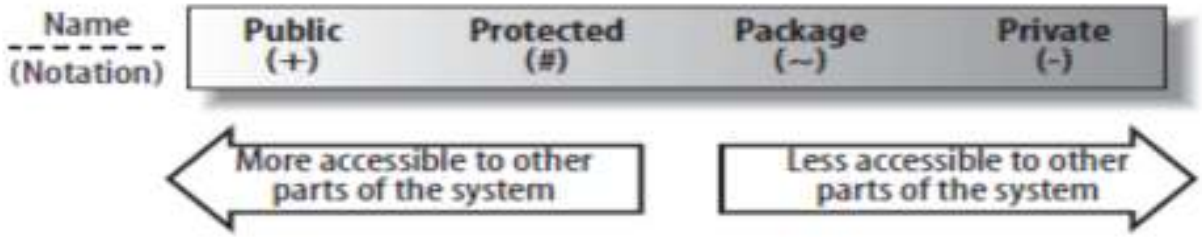
- **Classes in UML**
  - A class in UML is drawn as a rectangle split into up to three sections.
  - The top section contains the name of the class.
  - The middle section contains the attributes or information that the class contains.
  - The final section contains the operations that represent the behavior that the class exhibits.



Four different ways of showing a class using UML notation

8.3 Introduction to UML

- **Visibility**
  - Visibility describes how a class reveals its operations and data to other classes.
  - Once visibility characteristics are applied, you can control access to attributes, operations, and even entire classes to effectively enforce encapsulation.
  - There are four different types of visibility that can be applied to the elements of a UML model:



8.3 Introduction to UML

- **Public Visibility**
  - Public visibility is specified using the plus (+) symbol before the associated attribute or operation.
  - Declare an attribute or operation public if you want it to be accessible directly by any other class.
  - The collection of attributes and operations that are declared public on a class create that class's public interface.
  - The public interface of a class consists of the attributes and operations that can be accessed and used by other classes.
  - This means the public interface is the part of your class that other classes will depend on the most.

8.3 Introduction to UML

- **Protected Visibility**
  - Protected attributes and operations are specified using the hash (#) symbol and are more visible to the rest of your system than private attributes and operations, but are less visible than public.
  - Declared protected elements on classes can be accessed by methods that are part of your class and also by methods that are declared on any class that inherits from your class.

8.3 Introduction to UML

- **Package Visibility**
  - Package visibility, specified with a tilde (~), when applied to attributes and operations, sits in between protected and private.
  - Packages are the key factor in determining which classes can see an attribute or operation that is declared with package visibility.
  - The rule is fairly simple:

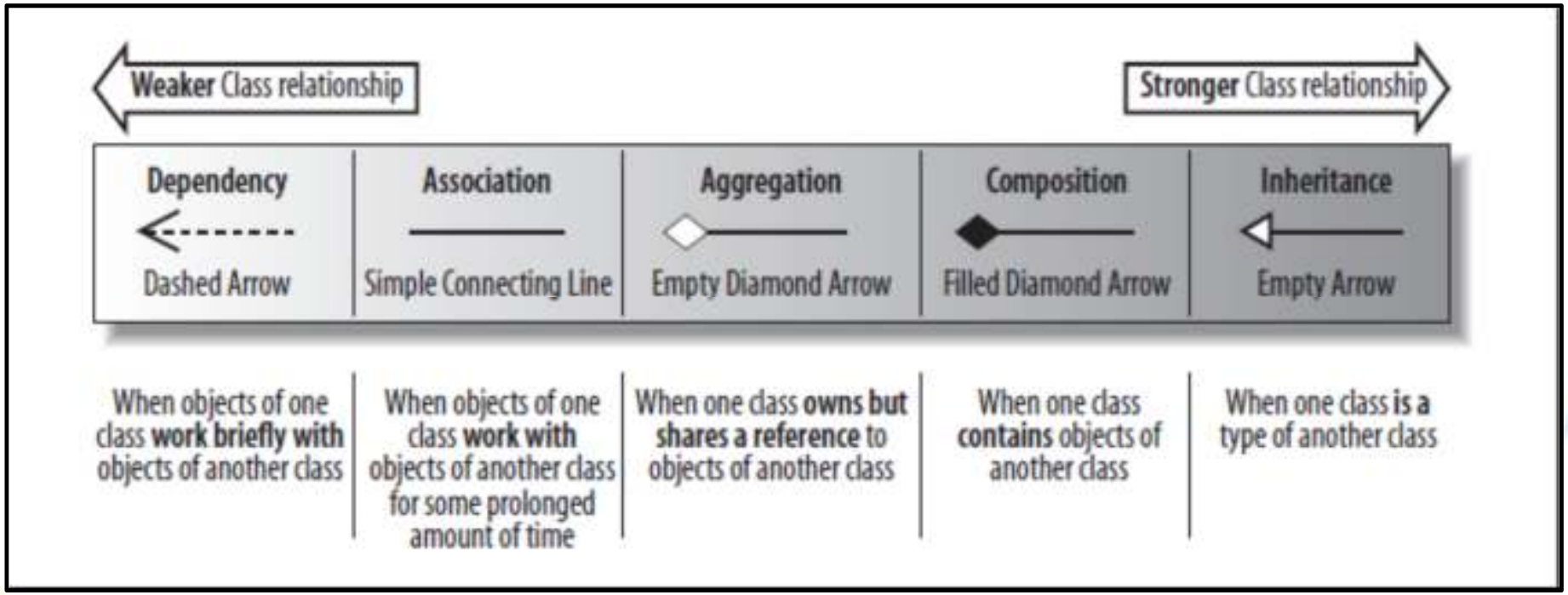
if you add an attribute or operation that is declared with package visibility to your class, then any class in the same package can directly access that attribute or operation

8.3 Introduction to UML

- **Private Visibility**
  - Private visibility is the most tightly constrained type of visibility classification, and it is shown by adding a minus (-) symbol before the attribute or operation.
  - Only the class that contains the private element can see or work with the data stored in a private attribute or make a call to a private operation.

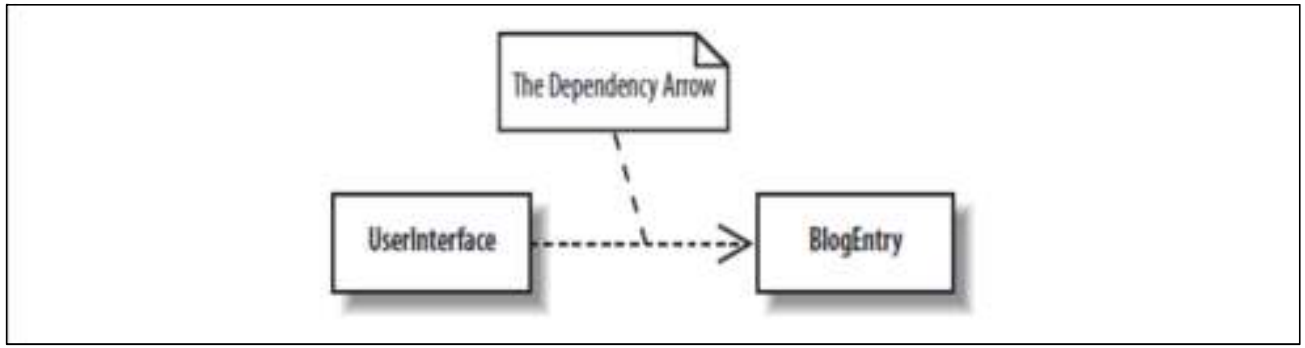
8.3 Introduction to UML

- **Class Relationships**



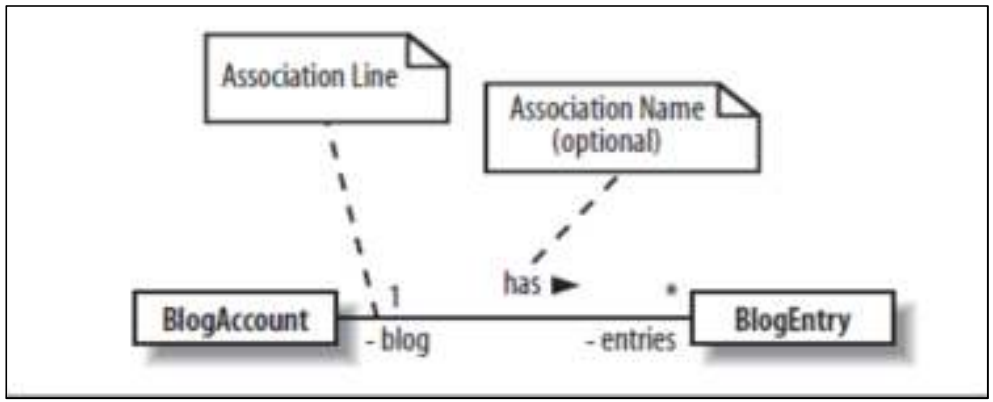
8.3 Introduction to UML

- **Dependency**
  - The dependency relationship is often used when you have a class that is providing a set of general-purpose utility functions, such as in Java's regular expression (java.util.regex) and mathematics (java.math) packages.
  - A dependency between two classes declares that a class needs to know about another class to use objects of that class.



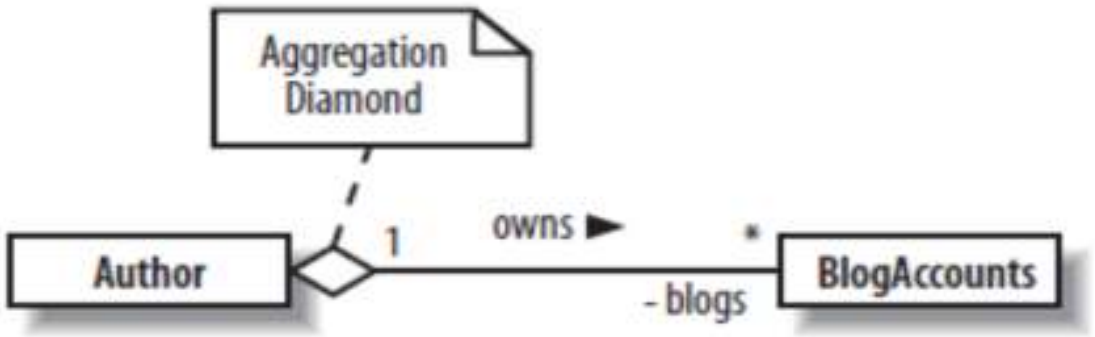
8.3 Introduction to UML

- **Association**
  - Although dependency simply allows one class to use objects of another class, association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute.
  - If you find yourself saying that a class works with an object of another class, then the relationship between those classes is a great candidate for association rather than just a dependency.



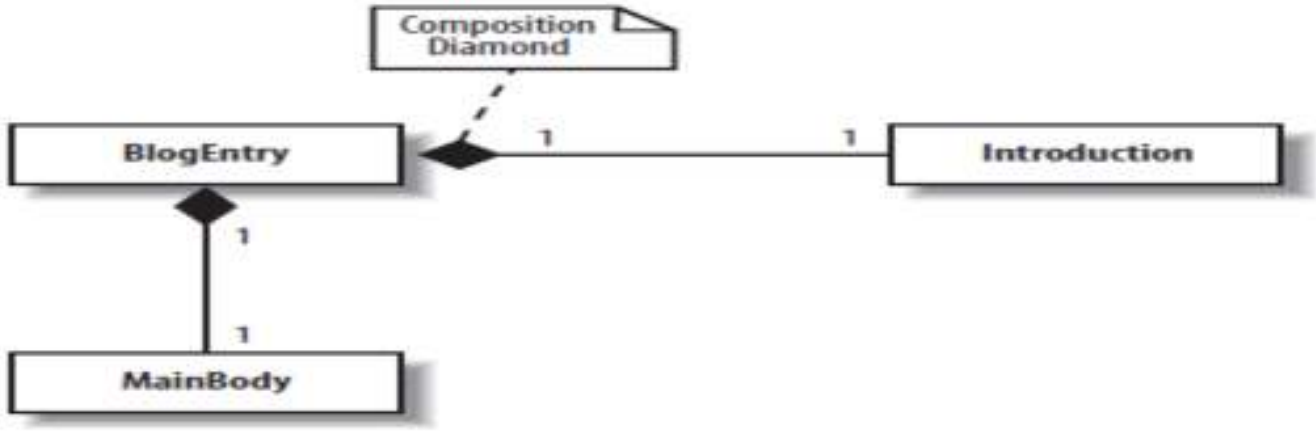
8.3 Introduction to UML

- **Aggregation**
  - Aggregation is really just a stronger version of association and is used to indicate that a class actually owns but may share objects of another class.
  - Aggregation is shown by using an empty diamond arrowhead next to the owning class, as shown in the figure.



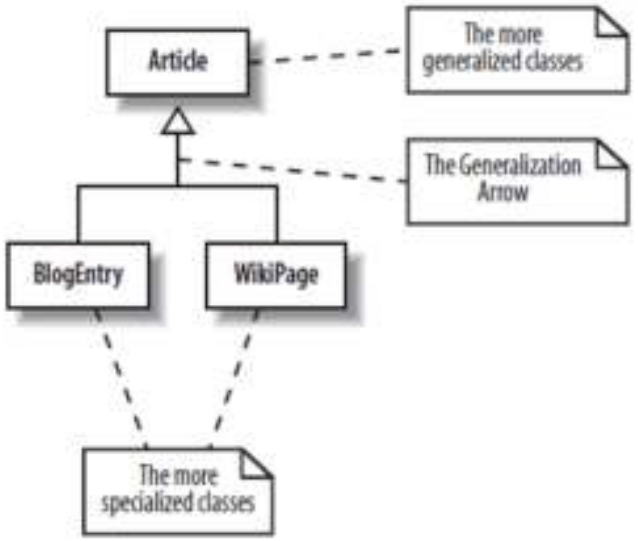
8.3 Introduction to UML

- **Composition**
  - Composition is an even stronger relationship than aggregation, although they work in very similar ways.
  - Composition is shown using a closed, or filled, diamond arrowhead, as shown in the figure:

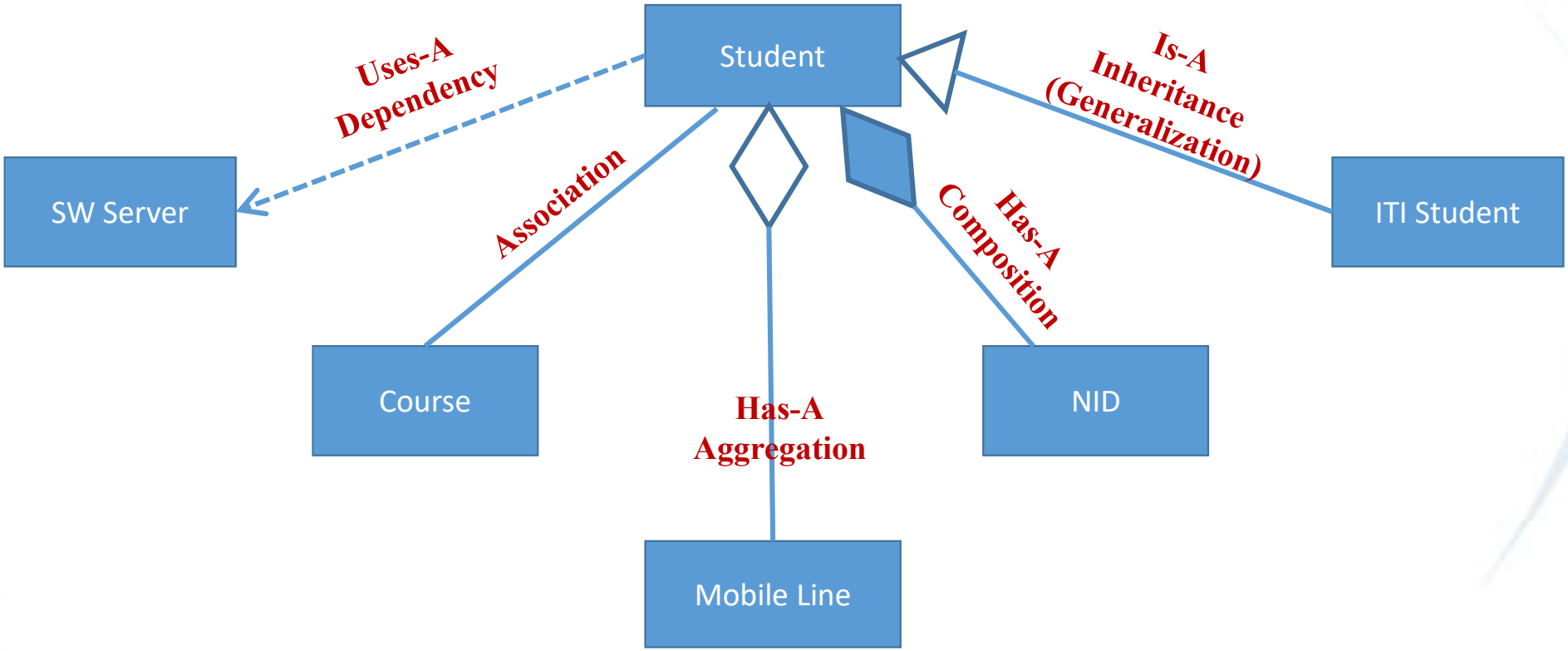


8.3 Introduction to UML

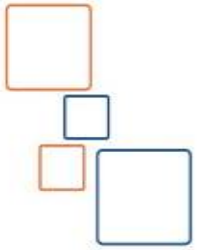
- **Inheritance (Generalization)**
  - Generalization is the strongest form of class relationship because it creates a tight coupling between classes.
  - The child (specialized) class inherits all of the attributes and methods that are declared in the parent (generalized) class and may add operations and attributes that are only applicable in child (specialized) class.
  - In UML, the generalization arrow is used to show that a class is a type of another class, as shown in the figure:



8.3 Introduction to UML



Lab Exercise



Assignments

- Stack class with Templates.

With My Best Wishes

Ahmed Loutfy

Invest In Yourself,
Develop Your Career