C++

session #6

#6

- Interface
- Data Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

Interface

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

Interface

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by

placing "= 0"

Interface

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

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

Interface

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Interface

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.

Data Encapsulation

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

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

Data Encapsulation

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

Data Encapsulation

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

Data Encapsulation

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

Data Encapsulation

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

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

Data Abstraction

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Data Abstraction

In C++, classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

Data Abstraction

For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

Data Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

Data Abstraction

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Data Abstraction

Data abstraction provides two important advantages

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

Data Abstraction

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

Polymorphism

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

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

Polymorphism

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

Polymorphism

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

Polymorphism

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

Polymorphism

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

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

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

Inheritance

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

Inheritance

A class derivation list names one or more base classes and has the form

class derived-class: access-specifier base-class

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

Inheritance

Access Control and Inheritance:

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

Inheritance

Access Control and Inheritance:

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

Inheritance

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

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

Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.

We hardly use protected or private inheritance, but public inheritance is commonly used.

Inheritance

Public Inheritance

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

Inheritance

Protected Inheritance

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

Inheritance

Private Inheritance

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

Inheritance

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

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

Where access is one of **public**, **protected**, or **private** and would be given for every base class and they will be separated by comma as shown above.