Object Oriented Programming

Unit 5: Polymorphism and Generic Programming with Templates

Polymorphism and Generic Programming with Templates

5.1. Dynamic Polymorphism: Need, Advantages

5.2. Rules for Virtual Function, Definition

5.3. Pure Virtual Function

5.4. Abstract Classes

5.5. Function Templates

5.6. Class Templates

5.7. Class Template with Overloaded Operators

CE: 5.0. Polymorphism

polymorphism = poly + morphism

✓**“poly”** refers to many or multiple

✓**“morphism”** refers to actions

- The word polymorphism means…

performing many action

or

having many forms.

or

“one name and multiple forms”

or

“giving different meaning to the same thing”

CE: 5.0. Polymorphism (Conti…)

- Polymorphism is key to the power of object oriented programming.
- It is so important that languages that don't support polymorphism cannot advertise as OO languages.
- Languages that support classes, but not polymorphism are called object-based languages.
- Polymorphism is a property by which the same message can be sent to objects of several different classes, and each object can respond to it in a different way depending on its class.

CE: 5.0. Polymorphism (Conti…)

What is polymorphism?

- The ability of a message to be displayed in more than one form is called ***polymorphism***.

- In C++ polymorphism is mainly divided into two types:
  1. Compile time Polymorphism
  2. Runtime Polymorphism

CE: 5.0. Polymorphism (Conti…)

1. Compile time Polymorphism:

- It also known as Static Polymorphism or Static Binding or Static Linking or Early Binding.
- **It is achieved by Function Overloading or Operator Overloading.**
- In case of function overloading and operator overloading, we have two or more functions with same name having different number of arguments. Based on how many parameters we pass during function call, it determines which function is to be called.
- Since, the call is determined during compile time, it is called ***compile time polymorphism***.

CE: 5.0. Polymorphism (Conti…)

2. Runtime Polymorphism:

- It also known as Dynamic Polymorphism or Dynamic Binding or Dynamic Linking or Late Binding.
- **It is achieved by Function Overriding.**
- In case of function overriding, we have two definitions of the same function, one is in base class and one is in derive class.
- The call to the function is determined at runtime to decide which definition of the function is to be called, that's the reason it is called **runtime polymorphism**.
- C++ supports a mechanism known as **virtual function** to achieve runtime polymorphism.

CE: 5.0. Polymorphism (Conti…)

- For Example:

✓ **The best real life example is of an Air Conditioner (AC).**

- Is a single device that performs different actions according to different circumstances. During summer it cools the air and during winters it heats the air.

✓ **Another best example is Humans.**

- We have different behaviors in different situations and in front of different people. A man behaves differently with his father, when he is with his friend he has different behavior, when he is with his sister he has different behavior and so on.

CE: 5.1. Dynamic Polymorphism

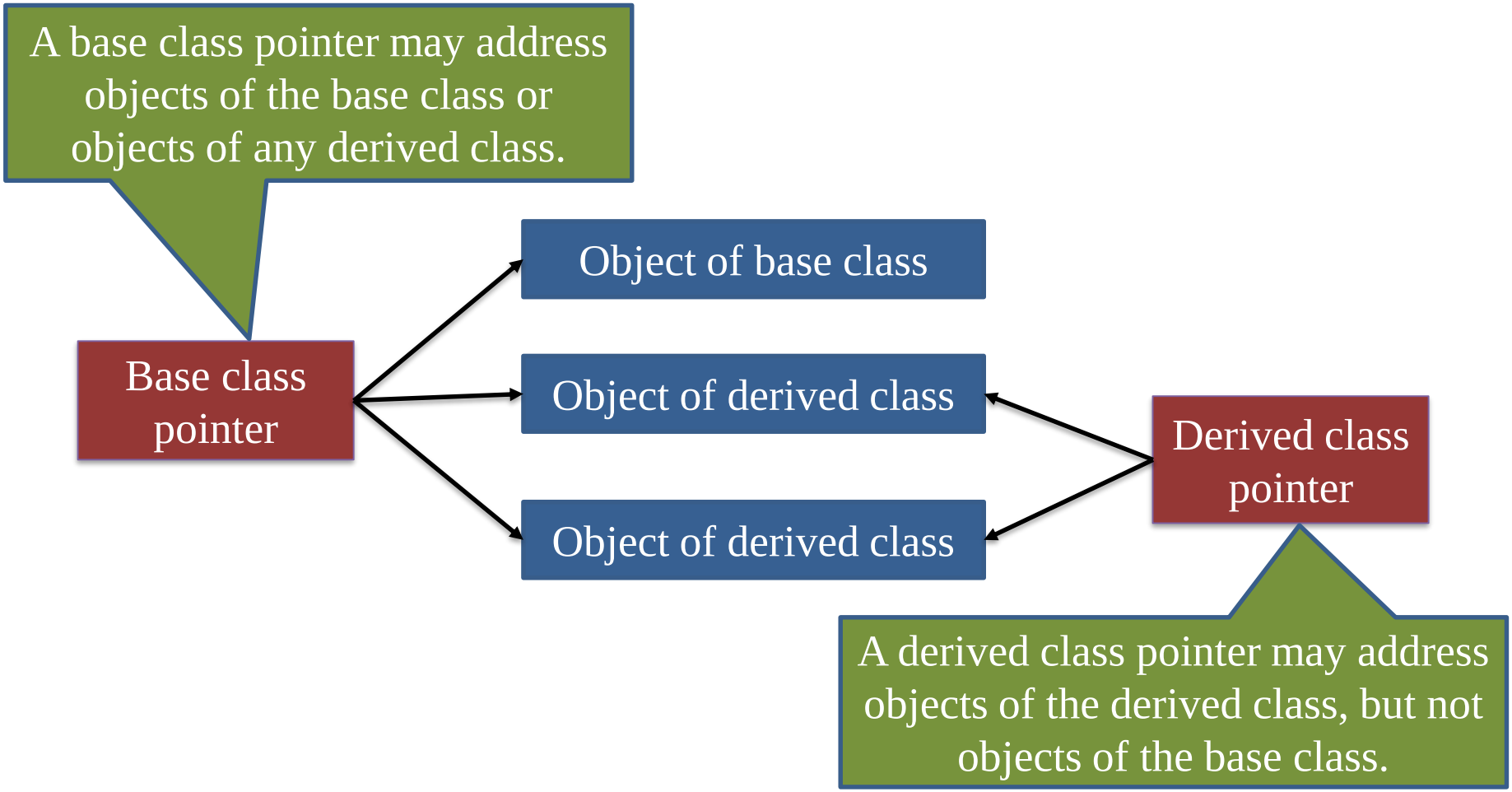
How to use polymorphism in C++?

- **But…**
- Before digging into polymorphism in C++, you should have a good sense of ***pointers*** and how ***inheritance*** works in C++.

- A base class pointer may address an object of its own class or an object of any class derived from the base class.

CE: 5.1. Dynamic Polymorphism (Conti…)

How a base class pointer may address a derived class object?



CE: 5.1. Dynamic Polymorphism (Conti…)

```
#include<iostream>
using namespace std;
class base
   {
      public:
         void display()
           {
              cout<<"Base class!\n";
           }
   };
class derived: public base
  {
       public:
         void display()
           {
              cout<<"Derived class!\n";
           }
   };
```

```
int main()
   {
        base *b;
        derived d;

        b = &d;
        b->display();
        return 0;
  }
```

Calls display() of derived class.

CE: 5.1. Dynamic Polymorphism (Conti…)

- **For Example: (Conti…)**

  - ✓ The reason for the incorrect output is that the call of the function display() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linking -** the function call is fixed, before the program is executed. This is also sometimes called **early binding,** because the display() function is set during the compilation of the program.

CE: 5.1. Dynamic Polymorphism (Conti…)

- Even though we have the parent class pointer pointing to the instance (object) of child class, the parent class version of the function is invoked.

But why we have created the pointer?

- We could have simply created the object of child class.

- Well, in above example, we have only one child class, but when we have a big project having several child classes, creating the object of child class separately is not recommended, as it increases the complexity and the code become error lying.

Solution ???

Virtual Function

CE: 5.1. Dynamic Polymorphism (Conti…)

```
#include<iostream>
using namespace std;
class base
   {
       public:
           virtual void display()
            {
                cout<<"Base class!\n";
            }
   };
class derived: public base
  {
       public:
          void display()
            {
                cout<<"Derived class!\n";
            }
   };
```

```
int main()
  {
        base *b;
        derived d;

        b = &d;
        b->display();//Run-time Polymorphism
        return 0;
  }
```

Since, we marked the function display() as virtual, the call to the function is resolved at runtime, compiler determines the type of the object at runtime and calls the appropriate function.

CE: 5.0. Polymorphism (Conti…)

Why it is needed?

- The main reason to used polymorphism is that…
  - ✓ we can implement different behaviors of the same object, depending upon the reference type passed to an object.

- It saves programmer a lot of time in re-creating code.
  - ✓ you don't have to write the complete code every time in different modules for every possible permutation.

CE: 5.0. Polymorphism (Conti…)

Advantages:

1. It helps programmers to reuse the code and classes; once written, tested and implemented. [Because it can be reused in many ways.]

2. Single variable name can be used to store variables of multiple data types(float, double, long, int etc).

3. It helps in reducing the coupling between different functionalities.

4. At Compile-time,
   - it enhances the code readability
   - and a single name can be used for the same type of methods.

5. At Run-time,
   - the same name can be used in subclass, which was used in superclass
   - also user can provide more specific definition additional to the general definition in superclass.

CE: 5.0. Polymorphism (Conti…)

Disadvantages:

- Developers find it difficult to implement polymorphism in codes.
- Run time polymorphism can lead to the performance issue; as machine needs to decide which method or variable to invoke, so it basically degrades the performances as decisions are taken at run time.
- It reduces the readability of the program. Because one needs to identify the runtime behavior of the program to identify actual execution time.

CE: 5.2
Virtual Function

CE: 5.2.1. Virtual Function

- When a same function name is declared or define in both base class and derived class, the function in a base class is declared as virtual function using the keyword “virtual” preceding it with normal declaration.
- Virtual functions should be defined in the public section of a class to realize its full potential benefits.
- When a function in made virtual, C++ determines which function is to be used at run-time based on the type of object pointed by the base class pointer (rather than the type of the pointer). This operation is referred to as *dynamic linkage*, or *late binding*.
- We can execute different versions of the virtual function, by making the base pointer to point the different objects.
- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

CE: 5.2.1. Virtual Function (Conti…)

- We can also call ***private function of derived class*** from the base class pointer with the help of virtual keyword.

- Compiler checks for access specifier only at compile time. So at run time when ***late binding*** occurs it does not check whether we are calling the ***private function*** or ***public function***.

CE: 5.2.1. Virtual Function (Conti…)

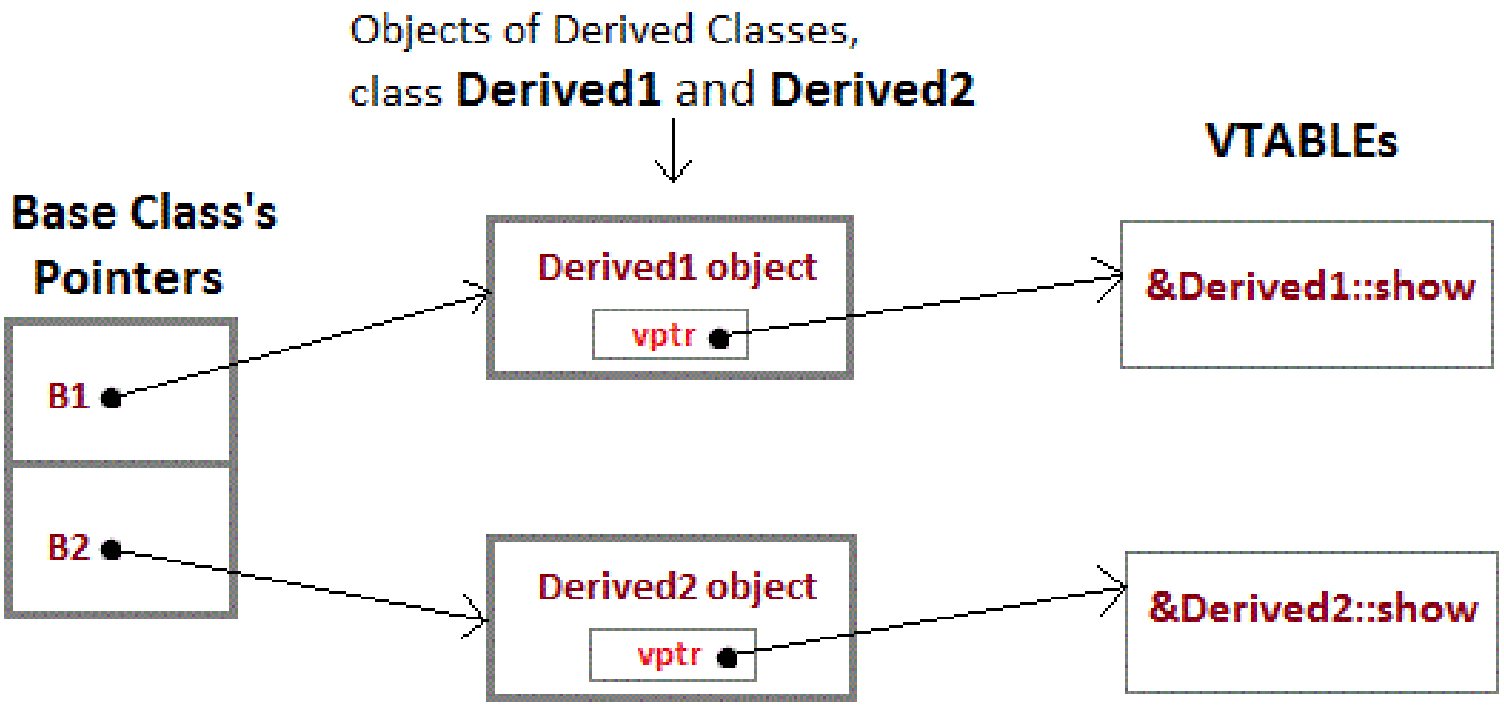
How to accessing private method of derived class?

```
class base
   {
       public:
           virtual void display()
            {
                cout<<"Base class!\n";
            }
   };
class derived: public base
  {
       private:
          virtual void display()
            {
                cout<<"Derived class!\n";
            }
   };
```

```
int main()
  {
        base *b;
        derived d;

        b = &d;
        b->display();
        return 0;
  }
```

CE: Mechanism of Late Binding in C++



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

CE: Mechanism of Late Binding in C++ (Conti…)

- To accomplish late binding, Compiler creates **VTABLEs**, for each class with virtual function. And the address of virtual functions is inserted into these tables.

- Whenever, an object of such class is created, the compiler secretly inserts a pointer called **vpointer**, pointing to **VTABLE** for that object.

- Hence when function is called, compiler is able to resolve the call by binding the correct function using the **vpointer**.

CE: 5.2.1. Virtual Function (Conti…)

Some Important Points to Remember:

- Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.

- If a function is declared as **virtual** in the base class, it will be *virtual in all its derived classes*.

- The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function.

CE: 5.2.2. Rules for Virtual Function

1. It must be members of some class.
2. The cannot be static members.
3. They are accessed by using object pointers.
4. It can be a friend of another class.
5. It must be defined in a base class, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class version must be identical (same).
   [If two functions with the same name have different prototypes, then C++ considers them as overloaded functions and the virtual function mechanism is ignored.]

CE: 5.2.2. Rules for Virtual Function (Conti…)

7. We cannot have virtual constructors, but we can have virtual destructors.

8. A base pointer can point to any type of the derived object, but the reverse cannot be possible.
   [that means we cannot use a pointer to a derived class to access an object of the base class type.]

9. When a base pointer points to a derived class, incrementing or decrementing will not make it to point to the next object of the derived class.
   [Incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.]

10. If it is defined in the base class, then it is not needed to be redefined in the derived class. In such cases, calls will invoke the base function.

CE: 5.3
Pure Virtual Function

CE: 5.3. Pure Virtual Function

- Pure virtual functions are virtual functions with no definition. They start with “virtual” keyword and ends with “= 0”.

- For Example:

  virtual void display() **= 0**;

- It is also called do-nothing function.

- The **“= 0”** tells the compiler that the function has no body and ***virtual function*** will be called as ***pure virtual function***.

CE: 5.3. Pure Virtual Function (Conti…)

Abstract Class

Pure Virtual Function

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

class Base
 {
      public:
        virtual void display() = 0;
 };

class Derived: public Base
 {
      public:
          void display()
            {
                cout << "Derived class function!\n";
            }
 };

int main()
 {
      Base b1;  //Compile Time Error
      Base *b2;
      Derived d;
      b2 = &d;
      b2->display();
 }
```

Hence, we cannot create simple object of base class.

CE: 5.3. Pure Virtual Function (Conti…)

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

class Base
 {
      public:
          virtual void display() = 0;
 };

class Derived: public Base
 {
      public:
          void display()
            {
                cout << "Derived class function!\n";
            }
 };

int main()
 {
      Base *b;
      Derived d;
      b = &d;
      b->display();
 }
```

purevirtual.cpp

CE: 5.3. Pure Virtual Function (Conti…)

- A Pure Virtual function can be given a small definition in the Abstract class, which has to be implemented by all the derived classes.
- But still we cannot create object of Abstract class.

- Also, the Pure Virtual function must be defined outside the class definition.

- If it is defined inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

CE: 5.3. Pure Virtual Function (Conti…)

```
#include<iostream>
using namespace std;
class Base  // Abstract class
  {
        public:
             virtual void display() = 0;
  };
void Base :: display()
   {
        cout << "Pure Virtual Function!\n";
    }
class Derived: public Base
  {     public:
           void display()
             {
                  cout << "Derived class function!\n";
             }
   };
```

```
int main()
  {
        Base *b;
        Derived d;
        b = &d;
        b->display();
  }
```

purevirtual.cpp

CE: 5.4
Abstract Classes

CE: 5.4. Abstract Classes

- A class which contains at least one Pure Virtual function is called ***Abstract Class***.

- Abstract classes are used to provide an Interface for its sub classes.

- Abstract Class must provide definition to the *pure virtual function*, otherwise the classes which is inheriting an abstract Class will also become abstract class.

CE: 5.4. Abstract Classes (Conti…)

Its Characteristics:

1. Abstract class cannot be instantiate, but pointers and references of Abstract class type can be created.

2. It can have normal functions and variables along with a pure virtual function.

3. They are mainly used for Upcasting, so that its derived classes can use its interface.
   [Upcasting is using the Super class's reference or pointer to refer to a Sub class's object.]

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Why can't we create Object of an Abstract Class?

- When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE, but doesn't put any address in that slot. So, the VTABLE will be incomplete.

- As the VTABLE for Abstract class is incomplete, the compiler will not let the creation of object for such class and will display an error message whenever you try to do so.

To be continued…