Print

Interfaces and Abstract Classes

Virtual Functions With Dynamic Binding

Back to Top

When we declare an object in C++, the source code makes the commitment and specifies the type of the object. This type of association between the object type and name cannot be broken during your project execution. The same rule applies to functions. The function prototype includes an identifier as the name of the function. The function name then becomes associated with the object code generated for this particular function. Again, this type of commitment cannot be changed during project execution. We might define some other functions using the same function name, and these functions with a different signature might be in the same class or in another class with the same or different signature, but, ultimately, these functions are different functions; they just have the same function name.

Did C++ function overloading break the C commitment of unique function names? For the human brain, we see the same function name when the name of the function is reused in the same class. However, compilers are not human brains. They see all these functions as having different names, as the name they know is a concatenation of the function name, function parameter list, function return types, and the name of the class to which the function belongs. When a compiler sees a function definition, it actually creates modified names that add the parameter list, return type, and class name to the function identifier. Each function name now becomes a unique name for the compiler.

For example, a function *calculateArea*() might be defined in several classes with different method signatures. Each one will represent its own implementation as a separate entity.

Note: The following code is not a complete code and will not run.

```
class Rectangle
{
 private:
       int width, height;

public:
      void calculateArea();
      //. . . .
};

class Box
{
private:

     int width, height, depth;
public:
    void calculateArea();
    //. . . .
};
```

When the compiler processes the object declaration, the association between the object name and type takes place at compile time. The same applies to member function call. When the compiler processes the member function call, the association between the object name and function takes place. For example:

```
//object name/object type are associated
//compiler knows that object r is of type Rectangle
Rectangle r;


//object name/object type are associated
//compiler knows that object b is of type Box
Box b;


//object name/function are associated
//compiler knows that the first call to calculateArea() refers to Rectangle::calculateArea()
r.calculateArea();


//object name/function are associated
//compiler knows that the second call to calculateArea() refers to  Rectangle::calculateArea()
b.calculateArea();
```

In a modern programming language like C++, we would like to change this linking process from static binding, which happens during compile time, to dynamic binding, which happens during program execution. With static binding, the linking process is fixed at compile time and cannot be changed later on during program execution dynamically. With dynamic binding, the search for the function call happens at runtime, during program execution. You might wonder what possible advantages dynamic binding brings. Consider processing an external input stream with objects of different types—we don't actually know the exact type of the objects that come from the outside environment.

For example, a program will never know how to calculate the area of a shape if it does not know what it is, because all shapes have different formulas for calculating surface area. When we do specify the type of the shape, the program will know exactly which one to call: Rectangle::calculateArea() or Box::calculateArea(). However, it would be a lot more dynamic if we could make the call using only one statement and change its meaning depending on the actual nature of the shape object.

```
shape.draw();// from class Rectangle or Box
```

If the object shape is a Rectangle in the current pass through, this statement should call Rectangle::calculateArea(). If it is a Box, it should call Box::calculateArea() instead. With static binding, this process is absolutely impossible. The compiler will only search for the definition of shape object, find its class, and look over the shape class definition. If calculateArea() is found in the shape class, it will be called at compile time, and there is no more room for searching for the meaning of the function calculateArea() at runtime. If calculateArea() is not found, the application will generate an error message.

However, when we are using the technique of dynamic binding, we can assume that several calculateArea() functions exist in different classes. We want one function call calculateArea() to mean several function calls: Rectangle::calculateArea(), Box::calculateArea(), and so on. We also want this meaning to be established dynamically during runtime so different formulas can be implemented depending on the meaning of the function call.

Guess what? This is what C++ virtual functions are set to achieve. The keyword *virtual* will create the runtime type resolution property for a message sent to an object of a derived type. To use this message, you implement the virtual function with the same name in the base class and in all derived classes.

Note: The following code is not a complete code and will not run.

```
class Shape
{
 private:
     String name;
     String color;

 public:
    virtual void calculateArea(); //a virtual function but don't have too much to do
     //. . . .
};

class Rectangle : public Shape
{
 private:
     int width, height;

 public:
    void calculateArea(); //a concrete function knows exactly what to do
     //. . . .
};
```

Pure Virtual Functions

Back to Top

As demonstrated above in the Shape class, the base virtual functions calculateArea() have no job to do since they have no meaning within the shape class: If we don't know what shape it is, we don't know how to supply implementation. The job of this base virtual function calculateArea() is just to define the interface as a standard for its future derived classes. However, in addition to this virtual function, we can still define some other useful data members and concrete members functions in the base Shape class. The derived classes Rectangle and Box can still inherit all these useful functions.

Since the Shape class objects have nothing to do, we will not create class objects. C++ makes this possible through the use of abstract classes as well as pure virtual functions. What is exactly a pure virtual functions? it is a virtual function that should not be called (like calculateArea() in Shape class) and contains no implementation. A syntax error will occur if you make a function call to a pure virtual function. An abstract class is a class that contains at least one pure virtual function, and no object can be created from the abstract class.

Are there any keywords for pure and abstract? The answer is no. There are no C++ keywords for pure virtual functions and abstract classes. Instead, compilers can identify a pure virtual function when its declaration is initialized to zero.

Note: The following code is not a complete code and will not run.

```
class Shape
{
 private:
     String name;
     String color;

 public:
    virtual void calculateArea(); //a virtual function but don't have too much to do
     //. . . .
};

class Rectangle : public Shape
{
 private:
     int width, height;

 public:
    void calculateArea(); //a concrete function knows exactly what to do
     //. . . .
};
```

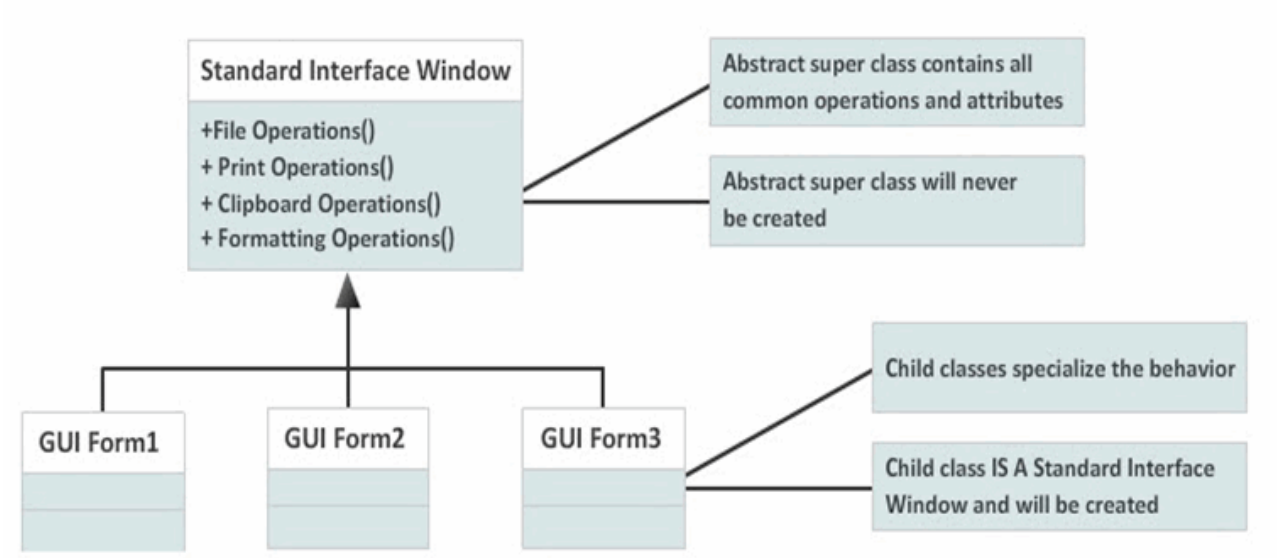
A pure virtual function does not have function implementation—the function body—at all. You will get a syntax error when you try to supply the function body. The presence of pure virtual function will make a class an abstract class. The abstract class is required to have at least one derived class. The derived class is then required to supply the implementation for the inherited pure virtual function. If you skip the implementation, you have to make your derived class abstract as well. The derived class should implement pure virtual function the same way it implements regular virtual function.

Abstract Classes

Back to Top

When we first begin thinking about classes, we typically assume that we will create an object of that class. There are many cases, however, in which we will want to include common behavior and attributes in a super class but will never intend to actually create an object of the super class.

For example, we have all used Windows programs. For some, in fact, you are using one as you read this lecture. Over the years, we have come to expect certain operations from every Windows program. We need to be able to save our work; print our work; copy, cut, and paste to the clipboard; and format text. Now, suppose you are required to build a Windows application, and you will have multiple windows in your program. You would not want to have to recreate the code for each of these common applications, so you would place this code in a superclass. As the diagram below shows, you would never really want to create a "blank" window, but each subclass of the super class would specialize the behavior, so only the subclasses are ever created.



Click here for transcript of this image.

In the structure above, the Standard Interface Window abstract super class creates invariant functionality that is common to all the child classes. This way, you only have to create the design and code of the invariant operations once, which makes coding much easier. If you need to change the behavior, you only have to make the changes in one location. Then, each child class can add behavior specific to that class's purpose.

Last week, we learned about **polymorphism** using **overloading**. That concept can be extended to inherited classes so that, if necessary, a child class can override the behavior of the superclass. For example, suppose you needed to be able to copy, cut, and paste pictures to the clipboard in one of the forms above. You could then override the super class's methods to specialize the clipboard operations to include pictures. Remember when we extended the functionality of the print function in the child class in Week 3? This was another example of overloading in inheritance.

This may be one of the more difficult implementation techniques you learn in this course. Adding abstract classes to your programming toolkit, however, will enable you to create reusable components, provide a consistent behavior to your applications, and make maintenance much easier.

Abstract Classes and Pure Virtual Functions

Back to Top

Click here for a definition from the MSDN Library.

Doing some research on virtual (that which implements dynamic and not static binding) versus pure virtual functions in C++ would prove beneficial at this point.

As an example, a Shape class may have functions to draw and move a shape. However, since the base class Shape does not have any specific details of any particular shape, it would be extremely difficult and perhaps fruitless to define these functions within the base class. In reality, this class would not be used to instantiate objects because it is too general. One would use pure virtual functions within this base class, thus converting the Shape class into an abstract one.

The following code shows how this is carried out in C++ code.

```
class shape
{
public:
    virtual void draw() = 0;
    //function to draw the shape
    //note that this is a pure virtual function
    virtual void move(double x, double y) = 0;
    //function to move the shape at coordinate (x,y). Also a pure virtual function.
    //.............
    //..........
};
```

What can I say about abstract classes and pure virtual functions?

- Any class that contains pure virtual functions is an abstract class.
- You cannot create objects from an abstract class.
- Virtual functions need not be defined.
- Abstract classes also may contain functions that are not virtual, as well as constructors. These must, however, be defined.

Program Design Tip - Overriding Base Class Functions

When overriding a base class function, you should not change the behavior dramatically; you should only extend or specialize the behavior. At some point in the override function, you should invoke the super class function that is being overridden. If you override a method and dramatically change the behavior or do not invoke the base class method that is being overridden, there is a good possibility that the child (derived) class is not a true descendent of the super class and you should rethink the design.

Summary

Back to Top

When creating objects, you may want to create a base class that provides invariant functionality in the super class but leaves implementation of other members to derived classes. You do this by creating an **abstract class**, which contains **pure virtual functions** that must be implemented in the inheriting class. Unlike other classes, however, an abstract class cannot have objects of its own; it must be inherited first. An abstract class may contain one or more pure virtual functions that need not be defined. But any other non-virtual functions it contains must have their own definitions within the base class.

Back to Top