12.2 — Virtual functions and polymorphism

BY ALEX ON JANUARY 30TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In the previous lesson on **pointers and references to the base class of derived objects**, we took a look at a number of examples where using pointers or references to a base class had the potential to simplify code. However, in every case, we ran up against the problem that the base pointer or reference was only able to call the base version of a function, not a derived version.

Here's a simple example of this behavior:

```
class Base
{
protected:

public:
    const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    const char* GetName() { return "Derived"; }
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;
}
```

This example prints the result:

```
rBase is a Base
```

Because rBase is a Base pointer, it calls Base::GetName(), even though it's actually pointing to the Base portion of a Derived object.

In this lesson, we will address this issue using virtual functions.

Virtual functions and polymorphism

A **virtual function** is a special type of function that resolves to the most-derived version of the function with the same signature. This capability is known as **polymorphism**.

To make a function virtual, simply place the "virtual" keyword before the function declaration.

Note that virtual functions and **virtual base classes** are two entirely different concepts, even though they share the same keyword.

Here's the above example with a virtual function:

```
class Base
{
protected:

public:
    virtual const char* GetName() { return "Base"; }
};

```

```
class Derived: public Base
{
public:
    virtual const char* GetName() { return "Derived"; }
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

This example prints the result:

```
rBase is a Derived
```

Because rBase is a pointer to the Base portion of a Derived object, when **rBase.GetName()** is evaluated, it would normally resolve to Base::GetName(). However, Base::GetName() is virtual, which tells the program to go look and see if there are any more-derived versions of the function available. Because the Base object that rBase is pointing to is actually part of a Derived object, the program will check every inherited class between Base and Derived and use the most-derived version of the function that it finds. In this case, that is Derived::GetName()!

Let's take a look at a slightly more complex example:

```
class A
{
public:
    virtual const char* GetName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* GetName() { return "B"; }
};

class C: public B
{
public:
    virtual const char* GetName() { return "C"; }
};

class D: public C
{
public:
    virtual const char* GetName() { return "D"; }
};

int main()
{
    C cClass;
    A &rBase = cClass;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

What do you think this program will output?

Let's look at how this works. First, we instantiate a C class object. rBase is an A pointer, which we set to point to the A portion of the C object. Finally, we call rBase.GetName(). rBase.GetName() evaluates to A::GetName(). However,

A::GetName() is virtual, so the compiler will check all the classes between A and C to see if it can find a more-derived match. First, it checks B::GetName(), and finds a match. Then it checks C::GetName() and finds a better match. It does not check D::GetName() because our original object was a C, not a D. Consequently, rBase.GetName() resolves to C::GetName().

As a result, our program outputs:

```
rBase is a C
```

A more complex example

Let's take another look at the Animal example we were working with in the previous lesson. Here's the original class:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Woof"; }
};
```

And here's the class with virtual functions:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
```

```
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

Note that we didn't make Animal::GetName() virtual. This is because GetName() is never overridden in any of the derived classes, therefore there is no need.

Now, using the virtual Speak() function, the following function should work correctly:

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main()
{
    Cat cCat("Fred");
    Dog cDog("Garbo");

    Report(cCat);
    Report(cDog);
}
```

Indeed, this program produces the result:

```
Fred says Meow
Garbo says Woof
```

When cAnimal.Speak() is evaluated, the program notes that it is a virtual function. In the case where rAnimal is pointing to the Animal portion of a Cat object, the program looks at all the classes between Animal and Cat to see if it can find a more derived function. In that case, it finds Cat::Speak(). In the case where rAnimal points to the Animal portion of a Dog object, the program resolves the function call to Dog::Speak().

Similarly, the following array example now works as expected:

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");
```

```
3
4   // Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
5   Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };
6   for (int iii=0; iii < 6; iii++)
7       cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

Which produces the result:

```
Fred says Meow
Garbo says Woof
Pooky says Woof
Truffle says Woof
Tyson says Meow
Zeke says Meow
```

Even though these two examples only use Cat and Dog, any other classes we derive from Animal would also work with our Report() function and Animal array without further modification! This is perhaps the biggest benefit of virtual functions -- the ability to structure your code in such a way that newly derived classes will automatically work with the old code without modification!

A word of warning: the signature of the derived class function must exactly match the signature of the base class virtual function in order for the derived class function to be used. If the derived class function has different parameter types, the program will likely still compile fine, but the virtual function will not resolve as intended.

Use of the virtual keyword

Technically, the virtual keyword is not needed in derived class. For example:

```
1   class Base
2   {
3   protected:
4
5   public:
6       virtual const char* GetName() { return "Base"; }
7   };
8
9   class Derived: public Base
10  {
11  public:
12      const char* GetName() { return "Derived"; } // note lack of virtual keyword
13  };
14
15  int main()
16  {
17      Derived cDerived;
18      Base &rBase = cDerived;
19      cout << "rBase is a " << rBase.GetName() << endl;
20
21      return 0;
22  }
```

prints

```
rBase is a Derived
```

Exactly the same as if Derived::GetName() was explicitly tagged as virtual. Only the most base class function needs to be tagged as virtual for all of the derived functions to work virtually. However, having the keyword virtual on the derived functions does not hurt, and it serves as a useful reminder that the function is a virtual function rather than a normal one. Consequently, it's generally a good idea to use the virtual keyword for virtualized functions in derived classes even though it's not strictly necessary.

Return types of virtual functions

Under normal circumstances, the return type of a virtual function and its override must match. Thus, the following will not work:

```
class Base
{
public:
    virtual int GetValue() { return 5; }
};

class Derived: public Base
{
public:
    virtual double GetValue() { return 6.78; }
};
```

However, there is one special case in which this is not true. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called covariant return types. Here is an example:

```
class Base
{
public:
    // This version of GetThis() returns a pointer to a Base class
    virtual Base* GetThis() { return this; }
};

class Derived: public Base
{
    // Normally override functions have to return objects of the same type as the base functio
n
    // However, because Derived is derived from Base, it's okay to return Derived* instead of
Base*
    virtual Derived* GetThis() { return this; }
};
```

Note that some older compilers (eg. Visual Studio 6) do not support covariant return types.

12.3 -- Virtual destructors, virtual assignment, and overriding virtualization

Index

12.1 -- Pointers and references to the base class of derived objects

Share this: