12.2 — Virtual functions and polymorphism

BY ALEX ON JANUARY 30TH, 2008 | LAST MODIFIED BY ALEX ON JUNE 14TH, 2018

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

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

int main()
{
    Derived derived;
    Base &rBase = derived;
    std::cout << "rBase is a " << rBase.getName() << '\n';
}
```

This example prints the result:

```
rBase is a Base
```

Because rBase is a Base reference, it calls Base::getName(), even though it's actually referencing the Base portion of a Derived object.

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

Virtual functions and polymorphism

A **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class. This capability is known as **polymorphism**. A derived function is considered a match if it has the same signature (name, parameter types, and whether it is const) and return type as the base version of the function. Such functions are called **overrides**.

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

Here's the above example with a virtual function:

```
class Base
{
public:
    virtual const char* getName() { return "Base"; } // note addition of virtual keyword
};

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

int main()
{
```

```
    Derived derived;
    Base &rBase = derived;
    std::cout << "rBase is a " << rBase.getName() << '\n';

    return 0;
}
```

This example prints the result:

```
rBase is a Derived
```

Because rBase is a reference 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 between Base and Derived. In this case, it will resolve to 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 c;
    A &rBase = c;
    std::cout << "rBase is a " << rBase.getName() << '\n';

    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 reference, which we set to reference 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 call the most-derived match between A and C. In this case, that is C::getName(). Note that it will not call D::getName(), because our original object was a C, not a D, so only functions between A and C are considered.

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, along with some test code:

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

    // 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 name)
        : m_name(name)
    {
    }

public:
    std::string getName() { return m_name; }
    const char* speak() { return "???"; }
};

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

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

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

    const char* speak() { return "Woof"; }
};

void report(Animal &animal)
{
    std::cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

This prints:

```
Fred says ???
Garbo says ???
```

Here's the equivalent class with the speak() function made virtual:

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

    // 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 name)
        : m_name(name)
    {
    }

public:
    std::string getName() { return m_name; }
    virtual const char* speak() { return "???"; }
};

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

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

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

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

void report(Animal &animal)
{
    std::cout << animal.getName() << " says " << animal.speak() << '\n';
}

int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

This program produces the result:

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

It works!

When animal.speak() is evaluated, the program notes that Animal::speak() is a virtual function. In the case where animal is referencing 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 animal references the Animal portion of a Dog object, the program resolves the function call to Dog::speak().

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.

Similarly, the following array example now works as expected:

```
Cat fred("Fred"), misty("Misty"), zeke("Zeke");
Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

// Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
Animal *animals[] = { &fred, &garbo, &misty, &pooky, &truffle, &zeke };
for (int iii=0; iii < 6; ++iii)
    std::cout << animals[iii]->getName() << " says " << animals[iii]->speak() << '\n';
```

Which produces the result:

```
Fred says Meow
Garbo says Woof
Misty says Meow
Pooky says Woof
Truffle says Woof
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

If a function is marked as virtual, all matching overrides are also considered virtual, even if they are not explicitly marked as such. 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. Consider the following example:

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

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

In this case, Derived::getValue() is not considered a matching override for Base::getValue() (it is considered a completely separate function).

Do not call virtual functions from constructors or destructors

Here's another gotcha that often catches unsuspecting new programmers. You should not call virtual functions from constructors or destructors. Why?

Remember that when a Derived class is created, the Base portion is constructed first. If you were to call a virtual function from the Base constructor, and Derived portion of the class hadn't even been created yet, it would be unable to call the Derived version of the function because there's no Derived object for the Derived function to work on. In C++, it will call the Base version instead.

A similar issue exists for destructors. If you call a virtual function in a Base class destructor, it will always resolve to the Base class version of the function, because the Derived portion of the class will already have been destroyed.

Rule: Never call virtual functions from constructors or destructors

The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient -- resolving a virtual function call takes longer than resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions. We'll talk about this more in future lessons in this chapter.

Quiz time

1) What do the following programs print? This exercise is meant to be done by inspection, not by compiling the examples with your compiler.

1a)

```
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:
// Note: no getName() function here
};

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

int main()
{
    C c;
    A &rBase = c;
    std::cout << rBase.getName() << '\n';

    return 0;
}
```

Hide Solution

B. rBase is an A reference pointing to a C object. Normally rBase.getName() would call A::getName(), but A::getName() is virtual so it instead calls the most derived matching function between A and C. That is B::getName(), which prints B.

1b)

```
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 c;
    B &rBase = c; // note: rBase is a B this time
    std::cout << rBase.getName() << '\n';

    return 0;
}
```

Hide Solution

C. This is pretty straightforward, as C::getName() is the most derived matching call between classes B and C.

1c)

```
class A
{
public:
    const char* getName() { return "A"; } // note: not virtual
};

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 c;
    A &rBase = c;
    std::cout << rBase.getName() << '\n';

    return 0;
}
```

Hide Solution

A. Since A is not virtual, when rBase.getName() is called, A::getName() is called.

1d)

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

class B: public A
{
public:
    const char* getName() { return "B"; }  // note: not virtual
};

class C: public B
{
public:
    const char* getName() { return "C"; } // note: not virtual
};

class D: public C
{
public:
    const char* getName() { return "D"; } // note: not virtual
};

int main()
{
    C c;
    B &rBase = c; // note: rBase is a B this time
    std::cout << rBase.getName() << '\n';

    return 0;
}
```

Hide Solution

C. Even though B and C aren't marked as virtual functions, A::getName() is virtual and B::getName() and C::getName() are overrides. Therefore, B::getName() and C::getName() are considered implicitly virtual, and thus the call to rBase.getName() resolves to C::getName(), not B::getName().

1e)

```
class A
{
public:
    virtual const char* getName() const { return "A"; } // note: function is const
};

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 c;
    A &rBase = c;
    std::cout << rBase.getName() << '\n';

    return 0;
}
```

Hide Solution

A. This one is a little trickier. rBase is an A reference to a C object, so rBase.getName() would normally call A::getName(). But A::getName() is virtual, so it calls the most derived version of the function between A and C. And that is A::getName(). Because B::getName() and c::getName() are not const, they are not considered overrides! Consequently, this program prints A.

1f)

```
#include <iostream>
class A
{
public:
    A() { std::cout << getName(); } // note addition of constructor

    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 c;

    return 0;
}
```

Hide Solution

A. Another tricky one. When we create a C object, the A part is constructed first. When the A constructor is called to do this, it calls virtual function getName(). Because the B and C parts of the class aren't set up yet, this resolves to A::getName().

12.2a -- The override and final specifiers, and covariant return types