18.2 — Virtual functions and polymorphism

ALEX[1] SEPTEMBER 19, 2021

In the previous lesson on pointers and references to the base class of derived objects (https://www.learncpp.com/cpp-tutorial/121-pointers-and-references-to-the-base-class-of-derived-objects/)[2], 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:

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
    std::string_view getName() const { return "Base"; }
};

class Derived: public Base
{
public:
    std::string_view getName() const { 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 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:

```cpp
#include <iostream>
#include <string_view>

class Base
{
public:
    virtual std::string_view getName() const { return "Base"; } // note addition of virtual keyword
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { 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:

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { 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:

```cpp
#include <iostream>
#include <string>
#include <string_view>

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(const std::string& name)
        : m_name{ name }
    {
    }

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

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

    std::string_view speak() const { return "Meow"; }
};

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

    std::string_view speak() const { return "Woof"; }
};

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

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

    report(cat);
    report(dog);

    return 0;
}
```

This prints:

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

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

```cpp
#include <iostream>
#include <string>
#include <string_view>

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(const std::string& name)
        : m_name{ name }
    {
    }

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

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

    virtual std::string_view speak() const { return "Meow"; }
};

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

    virtual std::string_view speak() const { return "Woof"; }
};

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

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

    report(cat);
    report(dog);

    return 0;
}
```

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:

```cpp
Cat fred{ "Fred" };
Cat misty{ "Misty" };
Cat zeke{ "Zeke" };

Dog garbo{ "Garbo" };
Dog pooky{ "Pooky" };
Dog 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 (const auto *animal : animals)
    std::cout << animal->getName() << " says " << animal->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. In the next lesson, we'll discuss how to guard against this.

Also note that if a function is marked as virtual, all matching overrides are also considered virtual, even if they are not explicitly marked as such.

Return types of virtual functions

Under normal circumstances, the return type of a virtual function and its override must match. Consider the following example:

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

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

In this case, Derived::getValue() is not considered a matching override for Base::getValue() and compilation will fail.

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.

> **Best practice**
>
> 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)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
// Note: no getName() function here
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

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

    return 0;
}
```

Hide Solution (javascript:void(0))[3]

> 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)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { 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 (javascript:void(0))[3]

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

1c)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    // note: no virtual keyword
    std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

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

    return 0;
}
```

Hide Solution (javascript:void(0))[3]

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

1d)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    // note: no virtual keyword in B, C, and D
    std::string_view getName() const { return "B"; }
};

class C: public B
{
public:
    std::string_view getName() const { return "C"; }
};

class D: public C
{
public:
    std::string_view getName() const { 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 (javascript:void(0))[3]

> 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)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    virtual std::string_view getName() const { return "A"; }
};

class B: public A
{
public:
    // Note: Functions in B, C, and D are non-const.
    virtual std::string_view getName() { return "B"; }
};

class C: public B
{
public:
    virtual std::string_view getName() { return "C"; }
};

class D: public C
{
public:
    virtual std::string_view getName() { return "D"; }
};

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

    return 0;
}
```

Hide Solution (javascript:void(0))[3]

> 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)

```cpp
#include <iostream>
#include <string_view>

class A
{
public:
    A() { std::cout << getName(); } // note addition of constructor

    virtual std::string_view getName() const { return "A"; }
};

class B : public A
{
public:
    virtual std::string_view getName() const { return "B"; }
};

class C : public B
{
public:
    virtual std::string_view getName() const { return "C"; }
};

class D : public C
{
public:
    virtual std::string_view getName() const { return "D"; }
};

int main()
{
    C c;

    return 0;
}
```

Hide Solution (javascript:void(0))[3]

> 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().

Next lesson
18.3 The override and final specifiers, and covariant return types
4

Back to table of contents
5

Previous lesson
18.1 Pointers and references to the base class of derived objects
6

7

171 COMMENTS Newest

JustABug
June 6, 2022 9:48 am

Can we make a derived class's function not being `virtual`? (I don't know whether it can be useful in real life. Just curious.)

```cpp
class A {public: virtual void f(){} }; // virtual function
class B : public A {public: void f(){} }; // make this function non-virtual
class C : public B {public: void f(){} }; // make this function non-virtual
```

0 Reply

Alex Author
Reply to JustABug[10] June 6, 2022 12:11 pm

No, if the base function is virtual, the derived fucntions are implicitly virtual. There is no way to un-virtual a function.

0 Reply

JustABug
June 6, 2022 9:37 am

`… if a function is marked as virtual, all matching overrides are also considered virtual, even if they are not explicitly marked as such.`

Why is `virtual` placed before all overriding functions in derived classes in this lesson's examples? Just for comprehension convenience or there some other reasons exist?

0 Reply

Alex Author
Reply to JustABug[11] June 6, 2022 12:10 pm

Some of these later lessons haven't been fully updated to incorporate all the modern best practices, so there may be minor inconsistencies with recommendations in earlier lessons.

0 Reply

Kuba
May 8, 2022 5:15 am

Those topics feel more and more difficult understand, but still very enjoyable read! Makes it much easier youre ommiting any fluff and giving us the most basic examples that are easy to follow, while still showing non obvious examples.

0 Reply

Mateusz kacKacpersi
April 29, 2022 9:05 am

:)

0 Reply

Mateusz kacKacpersi
April 29, 2022 8:52 am

Hello :)
ingerneral is better **solution** using std::string_view instead of std::string yes ?

```cpp
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(const std::string& name)
        : m_name{ name }
    {
    }

public:
    const std::string& getName() const { return m_name; }
    std::string_view speak() const { return "???"; }
};
```

so why are m_name is std::string and not std::string_view ?! o.O
and second question: why we dont using std::string_view instead of "const std::string&" ?

THX <3

0 Reply

Alex Author
Reply to Mateusz kacKacpersi[12] May 2, 2022 11:01 pm

Yes. std::string is an owner (std::string_view is not). If we want each Animal to have a unique name, we need to use std::string, not std::string_view.

We don't use std::string_view as the return type of getName() because we don't want to do a conversion to std::string_view every time someone calls getName().

0 Reply

Vithelma
Reply to Alex[13] June 5, 2022 3:48 pm

```cpp
class Animal
{
protected:
    std::string m_name;
.....
```

If we made m_name a `std::string_view` wouldn't it just refer to the string literals that exist in memory for the life of the program? Eg:

```cpp
Cat cat{ "Fred" };
Dog dog{ "Garbo" };
```

Wouldn't that give each animal a unique name.
I tried doing this and it seemed to work.

0 Reply

Alex Author
Reply to Vithelma[14] June 6, 2022 10:34 pm

Yes, string_view is essentially just a pointer to a string (either a string literal, or a string held by std::string). This is fine if that's what you want, but it's not suitable if we want to do anything that requires Animal to own its own name, like when asking the user to give their Animal a name.

0 Reply

XCI
February 1, 2022 1:14 am

I try the following code

```cpp
#include <iostream>
#include <string_view>

class Base{
public:
    virtual std::string_view getName() const { return "Base"; }
};

class Derived : public Base{
public:
    std::string_view getName() const { return "Derived"; } //not use virtual here
};

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

    return 0;
}
```

which also print

```
rBase is a Derived.
```

My question is:
What is the meaning that add "virtual" keyword to both derived and base class?
In my realization, "virtual" keyword is to tell the compiler to look if there are any more-derived function between base and derived, so why not just add "virtual" to base class, or this is just a convention?

0 Reply

Alex Author
Reply to XCI [15] February 1, 2022 12:06 pm

Modern best practice is to use `virtual` on the base class function and `override` for any derived functions. The `override` specifier is covered in the next lesson.

0 Reply

Harsha Narayana P
January 27, 2022 1:35 am

what do you mean by "most derived version of the function"?

1 Reply

Harsha Narayana P
Reply to Harsha Narayana P [16] January 27, 2022 3:50 am

I think I got to know.

```cpp
/******************************************************************************

                              Online C++ Compiler.
               Code, Compile, Run and Debug C++ program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/

#include <iostream>

using namespace std;
class Animal
{
    public:
    virtual void display()
    {
        cout<<"I am an animal\n";
    }
};

class Vertebrate: public Animal
{
    public:
    void display()
    {
        cout<<"I am a Vertebrate\n";
    }
};

class Human: public Vertebrate
{
    public:
    void display()
    {
        cout<<"I am a Human\n";
    }
};

class Man: public Human
{
};


int main()
{
    Man man;
    Animal& animal{man};
    man.display();
    return 0;
}
```

It makes sense that it should display "I am a Human" as a man is human.

0 Reply

Ishtmeet
October 21, 2021 8:08 pm

```cpp
// 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 };
```

Don't we need to pass the number of arguments inside the `[]` (pre C++11?).

Last edited 7 months ago by Ishtmeet

0 Reply

Alex Author
Reply to Ishtmeet [17] October 22, 2021 10:22 am

This site currently assumes C++14 as a minimum version. There's very little reason to use an older version of the language these days, especially pre-C++11.

0 Reply

George
October 19, 2021 11:56 pm

How is a reference to an object of different type possible? You get compile error if you refer an int or a double variable with a different type (for example int x{4}; double &y{x}), yet here reference to another class works fine.

0 Reply

nascardriver Sub-admin
Reply to George [18] October 20, 2021 7:45 am

A derived type is a base type. The reference refers to the base portion of the derived class.
`int` and `double` are entirely different types, they have nothing in common.

0 Reply

George
October 18, 2021 9:53 am

Do virtual functions require that the class object is a reference to a parent class, instantiated from a derived class?

0 Reply

Alex Author
Reply to George [19] October 18, 2021 4:25 pm

No, the reference (or pointer) can be to any class in the hierarchy.

0 Reply

George
Reply to Alex [20] October 19, 2021 8:58 am

Do they need the object to be a reference? What about a copy initialization:

C c;
A base{ c };
(where C is inherited from B, which is inherited from A). Will that invoke polymorphism? Well I checked and it doesn't invoke polymorphism. So we need reference. But, even with a reference construct, how the compiler knows anything about the childs, incl. the virtual function and how do we invoke it? Where is this information kept in a Base object?

Last edited 7 months ago by George

0 Reply

Alex Author
Reply to George [21] October 20, 2021 4:59 pm

We cover this topic when we discuss the virtual table: https://www.learncpp.com/cpp-tutorial/the-virtual-table/

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/121-pointers-and-references-to-the-base-class-of-derived-objects/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/pointers-and-references-to-the-base-class-of-derived-objects/
7. https://www.learncpp.com/virtual-functions/
8. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-569444
11. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-569443
12. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-568245
13. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-568323
14. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-569428
15. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-564985
16. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-564776
17. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-562169
18. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-562096
19. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-562033
20. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-562054
21. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-562086
22. https://www.ezoic.com/what-is-ezoic/