25.2 — Virtual functions and polymorphism

👤 **ALEX**[1]   🕐 **DECEMBER 11, 2024**

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/)](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

A **virtual function** is a special type of member function that, when called, resolves to the most-derived version of the function for the actual type of the object being referenced or pointed to.

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:

```
1   #include <iostream>
2   #include <string_view>
3
4   class Base
5   {
6   public:
7       virtual std::string_view getName() const { return "Base"; } // note addition of
8   virtual keyword
9   };
10
11  class Derived: public Base
12  {
13  public:
14      virtual std::string_view getName() const { return "Derived"; }
15  };
16
17  int main()
18  {
19      Derived derived {};
20      Base& rBase{ derived };
21      std::cout << "rBase is a " << rBase.getName() << '\n';
22
23      return 0;
    }
```

This example prints the result:

```
rBase is a Derived
```

> **Tip**
>
> Some modern compilers may give an error about having virtual functions and an accessible non-virtual destructor. If this is the case, add a virtual destructor to the base class. In the above program, add this to the definition of `Base`:
>
> ```
> 1   virtual ~Base() = default;
> ```
>
> We discuss virtual destructors in lesson 25.4 -- Virtual destructors, virtual assignment, and overriding virtualization (https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/)[3].

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 for a Derived object. 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
```

Note that virtual function resolution only works when a virtual member function is called through a pointer or reference to a class type object. This works because the compiler can differentiate the type of the pointer or reference from the type of the object being pointed to or referenced. We see this in example above.

Calling a virtual member function directly on an object (not through a pointer or reference) will always invoke the member function belonging to the same type of that object. For example:

```
1  C c{};
2  std::cout << c.getName(); // will always call C::getName
3
4  A a { c }; // copies the A portion of c into a (don't do this)
5  std::cout << a.getName(); // will always call A::getName
```

> **Key insight**
>
> Virtual function resolution only works when a member function is called through a pointer or reference to a class type object.

Polymorphism

In programming, **polymorphism** refers to the ability of an entity to have multiple forms (the term "polymorphism" literally means "many forms"). For example, consider the following two function declarations:

```
1  int add(int, int);
2  double add(double, double);
```

The identifier `add` has two forms: `add(int, int)` and `add(double, double)`.

Compile-time polymorphism refers to forms of polymorphism that are resolved by the compiler. These include function overload resolution, as well as template resolution.

Runtime polymorphism refers to forms of polymorphism that are resolved at runtime. This includes virtual function resolution.

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(std::string_view 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(std::string_view name)
        : Animal{ name }
    {
    }

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

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

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

class Dog: public Animal
{
public:
    Dog(std::string_view 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:

```
1   Cat fred{ "Fred" };
2   Cat misty{ "Misty" };
3   Cat zeke{ "Zeke" };
4
5   Dog garbo{ "Garbo" };
6   Dog pooky{ "Pooky" };
7   Dog truffle{ "Truffle" };
8
9   // Set up an array of pointers to animals, and set those pointers to our Cat and Dog
10  objects
11  Animal* animals[]{ &fred, &garbo, &misty, &pooky, &truffle, &zeke };
12
13  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.

Note that if a function is marked as virtual, all matching overrides in derived classes are also implicitly considered virtual, even if they are not explicitly marked as such.

This does not work the other way around -- a virtual override in a derived class does not implicitly make the base class function virtual.

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.

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, to make virtual functions work, the compiler has to allocate an extra pointer for each object of a class that has virtual functions. This adds a lot of overhead to objects that otherwise have a small size. 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;
}
```

Show Solution (javascript:void(0))[4]

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;
}
```

Show Solution (javascript:void(0))[4]

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;
}
```

Show Solution (javascript:void(0))[4]

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;
}
```

Show Solution (javascript:void(0))[4]

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;
}
```

Show Solution (javascript:void(0))[4]

1f)

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

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

    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;
}
```

Show Solution (javascript:void(0))[4]

275 COMMENTS   Newest ▾

Tim
  🕐 June 9, 2025 10:05 am PDT

I've noticed in some chapters you've returned getters (to std::strings) as a const ref since no new conversions has to occur. In this chapter they're returned as string_views. Is there a reason string_view return is used here or is it just a change in method in previous chapters that hasn't been updated in all modules..

👍 1   ↪ Reply

> **Jamison**
>   💬 Reply to Tim [10] 🕐 June 20, 2025 6:06 pm PDT
>
> EDIT:
>
> I wrote up a whole thing about std::string_view and literals but it's much simpler
>
> If you return const std::string& here (std::string& won't even compile) it creates temporary string via the constructor but you return reference to the temporary so it's UB to access it after.
>
> And std::string_view is fine cause string literals are compile time constants and live for the lifetime of the program so you can return string_view as a modern way to reference this (in C it would const char *)
>
> 📝 *Last edited 18 days ago by Jamison*
>
> 👍 0   ↪ Reply

---

redshift
  🕐 May 17, 2025 1:08 pm PDT

So C++ compilers can distinguish functions by return type after all, they just don't want to. Why is this not desirable?

👍 0   ➥ Reply

Misaka
🕐 May 13, 2025 4:48 am PDT

Hello! Thank you for the great lessons :) in the last quiz exercise, 1f,the main doesn't have the lines of code where you construct a new object and then you call getName(). Thank you!

✎ *Last edited 1 month ago by Misaka*

👍 0   ➥ Reply

Nidhi Gupta
🕐 May 6, 2025 9:15 am PDT

here is what I understand from reading:
classic case of runtime polymorphism not working as expected because the speak() function in the base class Animal is not declared as virtual.

So even though Cat and Dog override speak(), when you pass those objects to the report() function by reference to Animal, the base class version of speak() — which returns "???" — is always called.

Output of the program as written:
ruby

Fred says ???
Garbo says ???
This happens because without virtual, the function call animal.speak() inside report() is resolved at compile time, using the type of the reference (which is Animal), not the actual object (which is Cat or Dog).

👍 0   ➥ Reply

Bharath
🕐 April 22, 2025 4:21 am PDT

.

✎ *Last edited 2 months ago by Bharath*

👍 0   ➥ Reply

sandersan
🕐 April 19, 2025 9:09 am PDT

So, what factors actually affect the resolution of virtual functions? We have already seen that: function signatures, return types, and the `const` qualifier, all of which may affect virtual function resolution. Is there any other influencing factors?

👍 1　　↪ Reply

joddy
🕐 March 15, 2025 10:54 pm PDT

Dear Alex,

Why does C++ forces us to use pointers/references? Does the virtual keyword isn´t enough to clarify our intent?

Life would be much simpler if we could use a simple derived object to referece its base.

By the question here is: why pointers/references?

👍 0　　↪ Reply

MOEGMA25
🕐 November 22, 2024 5:44 am PST

But what do you do in a case where we want to use an function inside an override, but isn't in the Base. How do you resolve that? You can't make an non-existing function virtual.

👍 0　　↪ Reply

> **MOEGMA25**
> 💬 Reply to MOEGMA25 [11]　🕐 November 23, 2024 12:58 am PST
>
> Pure Virtual Functions.
> See: https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/
>
> 👍 0　　↪ Reply

Divyansh
🕐 November 10, 2024 8:14 am PST

Could we consider having `m_sound` like `m_name` and pass a string representing the sound to the `Animal` constructor within the derived class? I'm not sure if this was previously discussed, but I'd appreciate your insights on this approach.

👍 1　　↪ Reply

> **Alex**　Author
> 💬 Reply to Divyansh [12]　🕐 November 11, 2024 12:16 pm PST

You can, but this requires each Animal to allocate storage for that string (and make a copy of that string for each instantiated Animal). This is a good approach if you're avoiding virtual functions or if each Animal needs a unique sound. Since that isn't the case here, that approach is suboptimal.

👍 2   ↪ Reply

Divyansh
💬 Reply to Alex [13] 🕐 November 29, 2024 5:20 am PST

Thank you, got it.

👍 0   ↪ Reply

swiftseashell
🕐 July 30, 2024 12:12 pm PDT

im not sure if im 100% remembering properly, but i think i remember somewhere here saying to prefer references to pointers(which i personally do like references more), but does this also apply to cases like these?

👍 0   ↪ Reply

Alex `Author`
💬 Reply to swiftseashell [14] 🕐 July 30, 2024 6:55 pm PDT

Yes. Use references when you can, and pointers when you must.

👍 3   ↪ Reply

sandersan
💬 Reply to Alex [15] 🕐 April 19, 2025 9:06 am PDT

Should we use smart pointer whenever we need pointer? Like, in this lesson, we declare the array as follows:

```
1  std::array<std::unique_ptr<Animal>, 3> animals
2      {
3          std::make_unique<Cat>("Jessy"),
4          std::make_unique<Dog>("Jake"),
5          std::make_unique<Cat>("Kate"),
6      };
```

👍 0   ↪ Reply

swiftseashell
💬 Reply to Alex [15] 🕐 July 31, 2024 9:21 am PDT

very appreciated; thank you so much

👍 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. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
4. javascript:void(0)
5. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/pointers-and-references-to-the-base-class-of-derived-objects/
8. https://www.learncpp.com/virtual-functions/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-610796
11. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-604415
12. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-604021
13. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-604085
14. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-600278
15. https://www.learncpp.com/cpp-tutorial/virtual-functions/#comment-600313
16. https://g.ezoic.net/privacy/learncpp.com