


## 25.2 — Virtual functions and polymorphism

---

 [learncpp.com/cpp-tutorial/virtual-functions/](http://learncpp.com/cpp-tutorial/virtual-functions/)

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:

```
#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:

```
#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
```

### 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**:

```
virtual ~Base() = default;
```

We discuss virtual destructors in lesson [25.4 -- Virtual destructors, virtual assignment, and overriding virtualization.](#)

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:

```
#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:

```
C c{};
std::cout << c.getName(); // will always call C::getName

A a { c }; // copies the A portion of c into a (don't do this)
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:

```
int add(int, int);
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:

```

#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:

```

#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:

```

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.

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.

## Rule

If a function is virtual, all matching overrides in derived classes are implicitly virtual.

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:

```
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)

```

#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

1b)

```

#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

1c)

```

#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

1d)

```

#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

1e)

```

#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

1f)



```

#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](#)