Back to Resources

Inheritance

by Nathan Walters

Already know the basics of inheritance? Skip straight to inheritance in C++, polymorphism in C++, slicing, the `virtual` keyword, or an advanced description of how virtual methods work.

Motivation for inheritance

Inheritance, along with *encapsulation*, *abstraction*, and *polymorphism*, is one of the pillars of object-oriented programming. Inheritance lets new classes borrow, or *inherit*, functionality from existing classes. This lets us avoid duplicating functionality that should be shared across a variety of different but related classes. You can also specify a new implementation while remaining the same interface of an existing class.

Terminology

When we use inheritance, we say that some new class should inherit the members of an existing class. We call the existing class the *base class*, and the new class is the *derived class*. You'll sometimes hear the derived class referred to as a *subclass*, and the base class can be called the *parent class* or the *super class*.

Inheritance specifies an *is-a* relationship. Consider fruits: a Honeycrisp *is-a* Apple, and an Apple *is-a* Fruit. Such relationships are transitive: a Honeycrisp *is-a* Fruit.

Inheritance in C++

```
#include <string>

class Fruit {
  public:
    virtual std::string getName() = 0;
};

class Apple : public Fruit {
  public:
    std::string getName() {
      return "Apple";
    }
    virtual std::string getVariety() = 0;
};

class Honeycrisp : public Apple {
  public:
    std::string getVariety() {
      return "Honeycrisp";
    }
};
```

Here's what our fruits example from above looks like; let's walk through it.

The `Fruit` class

This class declaration should look a bit different than the ones you're used to seeing, particularly this line:

```
virtual std::string getVariety() = 0;
```

This line means that `getVariety()` is a *pure virtual function*. That is, the `Fruit` class doesn't provide an implementation of the `getVariety()` function; a subclass will have to provide an implementation. If a class has even one pure virtual function, we call the class *abstract*. That means we can't instantiate an instance of the class itself. Consider the following code:

```
#include <iostream>

// Class definitions would go here...

int main() {
  Fruit f;
  std::cout << f.getName() << std::endl;
}
```

If we tried to compile this, the compiler would complain that we were trying to instantiate an abstract class:

```
main.cpp:6:9: error: variable type 'Fruit' is an abstract class
  Fruit f;
        ^
```

The `Apple` class

Here we see our first example of inheritance! Look at how the `Apple` class is declared:

```
class Apple : public Fruit {
  ...
}
```

This is how we declare a class to be a subclass of another class in C++. `public` is an access specifier that specifies access rules for public or protected members of the base class. There are three access specifiers that can be used here:

- `public`: For a subclass derived as a `public` base class, any `public` members in the base class will be `public` in the derived class, and any `protected` members in the base class will be `protected` in the derived class.
- `protected`: For a subclass derived as a `protected` base class, any `public` or `protected` members in the base class will be `protected` in the derived class.
- `private`: For a subclass derived as a `private` base class, any `public` or `protected` members in the base class will be `private` in the derived class.

Note that a subclass can never access any private members of any of its parent classes.

The `Apple` class implements the `getName()` function from the `Fruit` class. But we still can't instantiate an `Apple` object! The `Apple` class also has a pure virtual method: `getVariety()`. This means that `Apple` is also an abstract class.

The `Honeycrisp` class

This is the last class in our example. It's derived from `public Apple`, and it implements `getVariety()`. This means that `Honeycrisp` is **not** an abstract class. So we can finally do something useful!

```
Honeycrisp h;
std::cout << "My name is " << h.getName() << std::endl;
std::cout << "I am a " << h.getVariety() << std::endl;
```

If we were to run the above code, this is the output we'd see:

```
My name is Apple
I am a Honeycrisp
```

Polymorphism in C++

We've now seen an example of how to declare derived classes in C++. But we haven't really been able to do anything useful with them yet! To show examples of why inheritance is useful, we need to introduce another concept: *polymorphism*. Polymorphism is the use of a single interface for objects of different types. In our fruits example, the interface determined by `Fruit` applies to both `Apple` and `Honeycrisp`: both also have a `getName()` method. And the `Apple` interface applies to `Honeycrisp` as well, since it has a `getVariety()` method.

Let's add another class to the mix to illustrate how inheritance is useful to us.

```
class RedDelicious : public Apple {
  public:
    std::string getVariety() {
      return "Red Delicious (an objectively bad cultivar)";
    }
}
```

Now, let's define a function that prints the type of an Apple:

```
void printType(Apple *apple) {
  std::cout << "This apple is a " << apple->getVariety() << std::endl;
}
```

We can now use this function to print the type of any `Apple` we might have!

```
Honeycrisp h;
RedDelicious rd;
printType(&h);
printType(&rd);
```

Running the above code would produce the following output:

```
This apple is a Honeycrisp
This apple is a Red Delicious (an objectively bad cultivar)
```

Note how we pass the `Apple` to `printType` by pointer. This is important! Remember that `Apple` is an abstract class; if we were to try to pass by value, we'd need to create a new instance of an `Apple`, which we can't do! But it's perfectly OK to have a pointer type of an abstract class, like `Fruit *` or `Apple *`.

We could also have written the function to take a reference:

```
void printType(Apple &apple) {
  std::cout << "This apple is a " << apple.getVariety() << std::endl;
}
```

Our function calls would look mostly the same, except that we don't have to take the address `h` or `rd` this time:

```
Honeycrisp h;
RedDelicious rd;
printType(h);
printType(rd);
```

Slicing

In the fruits example above, we couldn't declare a function that took a superclass argument because all of the superclasses were abstract. Let's consider a different example that can illustrate a pitfall when using inherited classes.

```
#include <iostream>

class A {
  public:
  void sayHello() {
    std::cout << "Hello from the base class!" << std::endl;
  }
};

class B : public A {
  public:
  void sayHello() {
    std::cout << "Hello from the derived class!" << std::endl;
  }
};

void printHelloMessage(A a) {
  a.sayHello();
}

int main() {
  A a;
  B b;
  printHelloMessage(a);
  printHelloMessage(b);
}
```

We might expect the above code to print the following output:

```
Hello from the base class!
Hello from the derived class!
```

But here's what's actually printed:

```
Hello from the base class!
Hello from the base class!
```

This might be somewhat surprising! Let's take a look at what's happening.

The first thing to note is that `printHelloMessage` takes an `A` object by *value*. Remember that when an object is passed by value, a copy of the original object is created. So, when we pass an instance of `B` to a function that takes an `A` by value, we're actually copying the instance of `B` into an instance of `A`. But the memory allocated for that instance of `A` doesn't have any space to remember that it was originally an instance of `B`. Since the `A` class has its own definition of `sayHello`, this is what's ultimately used.

We call this *slicing* because the information that something is an instance of a subclass is lost when we copy it into an instance of its super class.

How can we avoid this problem? You've already seen the answer to that in the last section! If we pass an object by pointer or by reference instead, we don't need to copy the object when the function is called. Instead, the *actual* type that the pointer points to is looked up at runtime so that the correct member function can be called.

With that knowledge, let's revise our code a bit:

```
void printHelloMessage(A &a) {
  a.sayHello();
}

int main() {
  A a;
  B b;
  printHelloMessage(a);
  printHelloMessage(b);
}
```

Let's look at what this new code would print.

```
Hello from the base class!
Hello from the base class!
```

Hmm. Not quite what we want yet! Turns out there’s one more piece that we’re missing.

The `virtual` keyword

When we have a pointer to a base class, the compiler doesn’t know by default that it needs to potentially look at subclasses for a different implementation of a function from the base class. In the `A`/`B` example from above, we find ourselves in exactly that situation: `B` provides a different implementation of `sayHello()` than `A` does. To tell the compiler that a subclass might override a function, we can tell it that with the `virtual` keyword. Let’s revise our code one more time:

```
#include <iostream>

class A {
  public:
  virtual void sayHello() {
    std::cout << "Hello from the base class!" << std::endl;
  }
};

class B : public A {
  public:
  void sayHello() {
    std::cout << "Hello from the derived class!" << std::endl;
  }
};

void printHelloMessage(A &a) {
  a.sayHello();
}

int main() {
  A a;
  B b;
  printHelloMessage(a);
  printHelloMessage(b);
}
```

Our only change this time was adding the `virtual` keyword before the definition of the `sayHello()` member function in `A`. Now, if we run our code, we see that we finally get the output we’ve been looking for:

```
Hello from the base class!
Hello from the derived class!
```

Yay!

If you were paying attention, you’ll remember you already saw the `virtual` keyword earlier in this note when we were discussing the fruits example. We declared `Food` as having a pure-virtual `getName()` function:

```
virtual std::string getName() = 0;
```

The meaning of the `virtual` keyword is the same there: it says to the compiler “Hey, when you’re trying to call this function on a pointer to some `Food` object, make sure you look in subclasses for a more specific implementation!” The `= 0` just tells the compiler that the `Food` class definitely doesn’t provide an implementation of this function itself.

Note that you only have to add the virtual keyword to the “most-base” class - that is, the class in the inheritance hierarchy where a function first appears. In the context of the fruit example, we could provide an implementation for `getName()` in the `Honeycrisp` class even though the intermediate `Apple` class didn’t re-declare a `virtual std::string getName()` function.

If you’re curious about how the `virtual` keyword works, read on! This is advanced content, so you won’t be tested on this on an exam. Nonetheless, it’s still interesting (or, at least, I think it is).

Bonus content: how do virtual methods work?

The following section is advanced content that's outside the scope of this class. However, it's still interesting if you're curious about how C++ works at a lower level.

First, a little bit of background on how functions are implemented in C++ (some of this might be a little hand-wavey; don't crucify me, ECE majors!). This assumes cursory knowledge of how computers work (what instructions are, how they're executed, and how memory works).

A function is essentially a sequence of instructions that operate on some data. Remember that instructions are just data too; that means they have their own location in memory. That gives us something interesting: we can have a pointer to a function! This is illustrated here:

```
#include <iostream>

void sayHello() {
  std::cout << "Hello, world!" << std::endl;
}

int main() {
  // Declare a variable "funcPointer" that can hold a pointer
  // to a function that takes zero arguments and returns nothing
  void (*funcPointer)();
  // Assign sayHello to funcPointer
  funcPointer = sayHello;
  // Tell the runtime to interpret "funcPointer" as a void*, which we can then
  // print to see the address of the function
  std::cout << reinterpret_cast<void*>(funcPointer) << std::endl;
  // We can even "call" a function pointer!
  funcPointer();
}
```

That code will print the following output when run. Note that the address of `funcPointer` (`0x400830`) might be different depending on the machine you're working on.

```
0x400830
Hello, world!
```

When a method is declared as `virtual`, the compiler will add a pointer to that function to a so-called *virtual method table* for the class. These functions are the same for all instances of that class, so the same table will be used for all instances as well.

Next, when you instantiate an object that has virtual methods, a pointer to this virtual method table is added as a hidden member of that object. This pointer is called a *virtual table pointer*, *vpointer*, or just *VPTR*. Now, when you try to call a function on an object with virtual methods, the compiler will use the VPTR to look up which implementation to actuall call at runtime!

Let's walk through a concrete example using the `A`/`B` code from above. Here it is again for convenience:

```
#include <iostream>

class A {
  public:
  virtual void sayHello() {
    std::cout << "Hello from the base class!" << std::endl;
  }
};

class B : public A {
  public:
  void sayHello() {
    std::cout << "Hello from the derived class!" << std::endl;
  }
};

void printHelloMessage(A &a) {
  a.sayHello();
}

int main() {
  A a;
  B b;
  printHelloMessage(a);
  printHelloMessage(b);
}
```

When the compiler creates space for the object `b`, it'll include a pointer to the virtual method table for the `B` class, which includes the implementation of `sayHello()` that prints "Hello from the derived class!" When we call `printHelloMessage()`, the virtual table pointer will be used to figure out the correct version of the function to call.

A lot of people ask why methods aren't virtual by default in C++ like they are in Java or other languages. One reason is that C++ is very focused on efficiency and speed. If a method isn't ever going to be overridden by a subclass, the compiler can skip generating the code to look up the correct function to call at runtime. This means that the resulting code will run faster! So, unless you specifically opt in to this behavior with the `virtual` keyboard, C++ defaults to doing the fast, efficient thing.

This is just an overview of how it works; Wikipedia provides a much more comprehensive look at virtual methods. Check it out if this was interesting for you!