Virtual Member Functions

Now that you've learned about inheritance and constructors, let's take a look at how derived-class member functions may be **redefined or overridden**. Open the **Repl** from Lesson **15A** and we'll continue with our simple "finger-exercise" example that lets you concentrate on one piece of the inheritance puzzle at a time.

A derived class may override member functions in the base class. The base class must permit that by using the keyword `virtual` on the prototype. Let's see how that works by modifying the `Person` class to add a new `virtual toString()` member function and a `virtual` destructor like this:

```
class Person
{
public:
    . . .
    virtual std::string toString() const;
    virtual ~Person() = default;
private:
    std::string name;
};
```

It is up to the **base class designer** to decide which member functions **may be** overridden and which may not. Member functions which allow derived classes to override them **should be preceded** with the keyword `virtual`.

As soon as you add a single `virtual` function, you should add a **`virtual` destructor**as shown in the `Person` class header. This uses the `=default` keyword to keep the synthesized destructor written by the compiler.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Implementing toString()

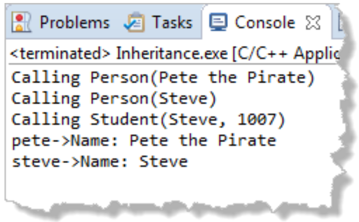
The implementation of `toString()`, in `person.cpp` does not repeat the keyword `virtual`. Let's have it display the person's name, like this:

```
string Person::toString() const
{
    return "Name: " + name;
}
```

The `Student` class inherits `Person::toString()`. If the `Student` class does nothing else, then there is no difference between a `virtual` member function and a regular member function. To see this, modify `main()` to add the following two lines:

```
cout << "pete->" << pete.toString() << endl;
cout << "steve->" << steve.toString() << endl;
```

When you run the sample program it looks like this.



The variable `pete` prints out the name as you'd expect (since `pete` is a `Person` object). The variable `steve` **also** uses the new `toString()` member function defined in `Person`. To `steve`, it is just another **inherited** member.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Overriding toString()

When another class (like `Student`) wants to provide a different implementation for a **`virtual`** member function, like **`toString()`**, it must:

1. Use **exactly the same signature** (number and type of parameters) as the original **`virtual`** function in the base class. There are no conversions between **`int`** and **`double`** for instance as with overloading.
2. Return **exactly** the same type as the original member functions.

Let's override **`toString()`** in the **`Student`** class. Here's the header:

```
class Student : public Person
{
public:
    Student(const std::string name, long sid);
    long getID() const;
    std::string toString() const;
private:
    long studentID;
};
```

Note that the prototype is copied **exactly** from **`Person::toString()`**, except for the keyword **`virtual`**. You **do not need to repeat** the word **`virtual`** in the derived class definition, (although you **may** for documentation purposes). A **`virtual`** function is always **`virtual`**, and a non-virtual function **cannot** be made **`virtual`** in one of its subclasses.

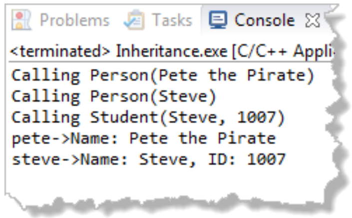
This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Implementing Student::toString()

Here's one possible implementation of `Student::toString()`:

```
string Student::toString() const
{
    return "Name: " + getName() // inherited member
        + ", ID: " + to_string(studentID);
}
```

While this works, it **doesn't** take advantage of the of the already-written **`toString()`** member function in the **`Person`** class; in the **`Student`** version of **`toString()`**, you're **duplicating exactly the same code**.



Is there some way to run the original version of **`toString()`**, and just **add on** the new parts you want, like the **`studentID`**? Is there some way you can **combine** inherited and overridden member functions?

Yes, there is.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Combining toString()

`Student` inherits both `getName()` and `toString()` from `Person`. When you create a **`Student`**, you can use both of those members if they were defined in **`Student`**.

Put that to work by **calling** the **inherited** version of **`toString()`** **from inside** the new overridden **`toString()`** member function. Use the **scope resolution operator** to specify that you wish to call the base class version of **`toString()`**.

```
string Student::toString() const
{
    return Person::toString() // base-class member
        + ", ID: " + to_string(studentID);
}
```

If you forget to use the scope-resolution operator, your program **blows up the stack and crashes**. At least in Java it is polite enough to give you a StackOverflowError when you try to run it. In C++, you'll just see a seg-fault message.

> *Don't confuse method **overriding** (which is what we're doing here), with method **overloading**. With overloading, two or more methods have the same name, but different parameter lists. Overloaded methods are in the same class but overridden methods are in a subclass and they must have exactly the same parameters and return type as the method that they are overriding.*

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

The override Keyword

While always a logic error for a derived class to redefine a non-virtual function, it **is not** a syntax error. C++ 11 added new **contextual keywords** that **allow the compiler to catch such logic errors** that previously were often hidden, and turn them into syntax errors that can be caught at compile time.

To tell the compiler that you **intend to override** a base class function, add the contextual keyword `override` to the end of the member function declaration like this:

```
std::string toString() const override;
```

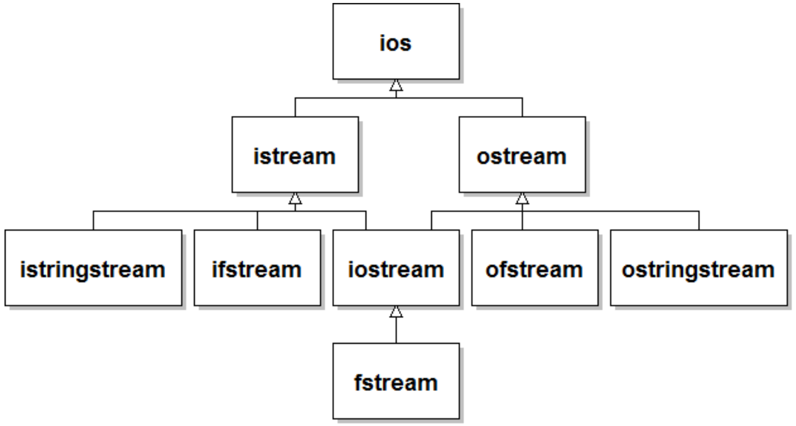
Now, if you were to forget the `virtual` in the base class, trying to (incorrectly) override a non-virtual inherited member function, or misspell the name of the member function, or provide the wrong arguments, the **compiler catches those errors** and warns you when you compile, like this:

```
error: 'Student::toString()' marked override, but
    does not override std::string Person::toString()
```

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Class Hierarchy

You first met a class hierarchy when we looked at the related stream classes. The class `ios` represents a **general** stream type, used for any kind of I/O. The classes `istream` and `ostream` generalize the notions of input and output streams. The C++ file and string-stream classes fall naturally into their appropriate position.



Each class shown here is a **derived class** (subclass) of the class that appears above it in the hierarchy. `istream` and `ostream` are both derived classes of `ios`, while `ios` is a **base class** (superclass) of both `istream` and `ostream`. Similar relationships exist at all different levels of this diagram. For example, `ifstream` is **derived from** `istream`, and `ostream` is the base class of `ofstream`.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Stream Substitutability

Writing data to a file is almost as easy as printing it on the screen. Once an `ofstream` object is set up, you can use the `<<` operator with the file stream in the same way you can with the `cout` object:

```
int x = 42;
ofstream out("myfile.dat");
cout << "x->" << x << endl;   // of course this works
out << "x->" << x << endl;    // this works as well
```

Well, the question is, **why does that work?** To understand this, think back to the `write` function that you created for the `Point` structure:

```
ostream& write(ostream& out, const Point& p)
{
    out << "(" << p.x << ", " << p.y << ")";
    return out;
};
```

This works with `cout` and `cerr`, both of which are `ostream` objects.

```
Point p = {4, 2};
write(cout, p) << endl;
write(cerr, p) << endl;
```

So, what do you have to do to adapt the function so that it works with `ofstream` objects and maybe even `ostringstream` objects? The answer, perhaps surprisingly, is that **you do not have to do anything**; it already works with `ofstream` objects just as it does with `ostream` objects like `cout`, **because every `ofstream` object IS-A `ostream` object** through the **principle of substitutability**.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Substitution vs. Conversion

C++ allows automatic conversions between the built-in numeric types; with numeric conversion, the compiler runs a built-in algorithm and tries to calculate the closest value that you desire. That's **not what happens** with objects in a class hierarchy.

When you pass an `ofstream` object to a function that expects an `ostreams&`, **no conversion takes place at all!** Instead, the `ofstream` object is automatically treated as if it **were** an `ostream` object, because the `ostream` and `ofstream` classes are related as in a special way through inheritance. Because the `ofstream` class is derived from the `ostream` class we can **substitute it** for the expected `ostream` parameter.

We can do that because the derived class inherits all of the characteristics of its base class, so that anything an `ostream` object can do, an `ofstream` object can do as well, by definition. This ability to allow a derived or subclass object to be used in any context that expects a base-class object is known as the **Liskov Substitution Principle**, after computer scientist Barbara Liskov.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Class Relationships

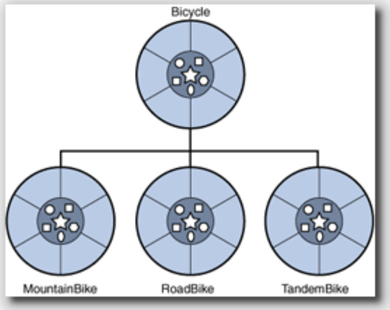
Value-Oriented or Object-Based programming involves creating new, user-defined types. There are four strategies for building a new type:

- Build it **completely from scratch**, using only built-in components.
- Build it from scratch, but **make use of the classes that others have written** to do some of the work.
- **Combine simpler types** to create complex types. This is **composition**.
- **Extend** a general class, adding new features. This is called **inheritance**.

Programming with inheritance is called **Object Oriented Programming**.

These strategies express three kinds of "class relationships":

- The **uses-a** relationship, (or **association**), occurs when your class uses the services of other classes. For instance, if your class uses `cout` in one of your member functions, your class is dependent on the `ostream` class.
- The **has-a** relationship, says one class is a combination of other objects. In the **has-a** relationship one type is composed of different parts. A `Bicycle` class thus may contain two instances of the `Wheel` class.
- The **is-a** relationship, when one class is an extension or "kind of" another class. The **is-a** relationship occurs when members of one class are a **subset** of another class. The **is-a** relationship is implemented using **public inheritance**. In the relationship shown here, we'd say that a `MountainBike` **is-a** `Bicycle`.



This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Polymorphic Inheritance

Public inheritance is a form of specialization. The derived class inherits both the member functions and the data members from the base class, while optionally adding more of both. The derived class **IS-A specialized form** of the more general base class.

A derived class **may override** a **`virtual`** member function to add specialized behavior, as we did with **`Student::toString()`**. This is called **polymorphic inheritance**, it provides **specialized behavior** in response to the same messages.

Let's see if that's true. Let's use our simple **`Person<-Student`** hierarchy from the last few lessons and see what happens with some experiments. Click the Running Man on the left to open a copy of the lab for this lesson. Make sure you **Fork** it so that you have your own copy.

Change **`toString()`** in each class so it identifies the class at the beginning of the method. Here are the modified **`toString()`** member functions. Notice that this version of the **`Student::toString()`** no longer calls its base class version; it **entirely replaces it**.

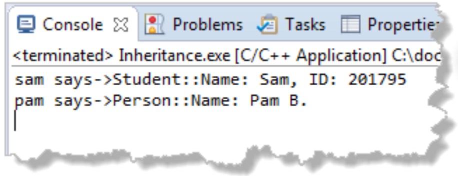
```
string Person::toString() const
{
    return "Person::Name: " + name;
}

string Student::toString() const
{
    return "Student::Name: " + getName()
        + ", ID: " + to_string(studentID);
}
```

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Static Polymorphism

Use `make run` to see the main program running. This is a kind of "polymorphism", known as **static polymorphism**. You send the same message to different objects and each **responds to the same message according to its nature**, just like the duck, dog and cat in the picture.



This is not what we mean when we talk about polymorphism. This would work exactly the same even if `Person` and `Student` were completely unrelated classes.

What we mean by polymorphism is an inheritance relationship where the request can be sent to **any kind of `Person` object**, and the specialized `Person`, such as a `Student` or an `Employee` responds appropriately.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

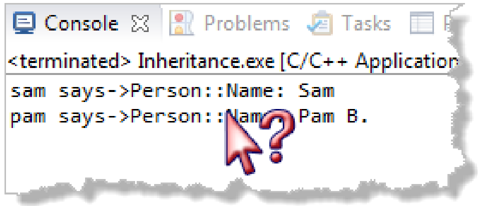
A Perplexing Problem

Change the example (`main.cpp`) again, so it looks like this:

```
int main()
{
    Person sam = Student("Sam", 201795);
    Person pam = Person("Pam B.");

    cout << "sam says->" << sam.toString() << endl;
    cout << "pam says->" << pam.toString() << endl;
}
```

Now you have two `Person` objects: one "regular" `Person`, and one specialized `Person` who is a `Student`. It the output the same as previously? **No!!!**



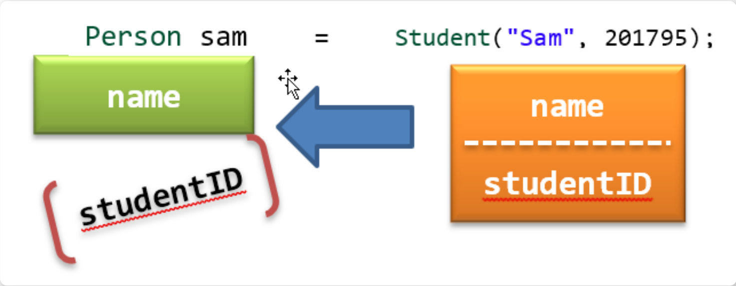
For the `Student sam`, you know longer see the `ID`. And, both the `Student` and the `Person` are identified with `Person::Name`, even though we do have an overridden member function, `Student::toString()`.

Why does this happen?

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

The Slicing Problem

Here's why this happens. First, objects in C++ are value types, unlike the reference types in Java. When you assign a derived class object to a base class variable, **only the base class portion** of the object is copied. **This is called the slicing problem.**



If you pass a derived class object **by value** to a function that expects a base class object, the same slicing will occur as well. This is easy to fix. Just **always** follow this rule:

> Never ever ever ever assign a derived class object to a base class variable. Ever!

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

References & Pointers

While slicing is a problem it is not the only culprit here. Even without slicing, the code would still not work because in C++ **polymorphism only works with references or pointers**.

To see, this, make the following changes to `main.cpp`:

```
Student sam = Student("Sam", 201795);
Person pam = Person("Pam B.");
Person& samRef = sam;
Person* samPtr = &sam;
cout << "sam says->" << samRef.toString() << endl;
cout << "sam says->" << samPtr->toString() << endl;
cout << "pam says->" << pam.toString() << endl;
```

Now, the `Person&` reference `samRef` refers to the Student object `sam`, and when we call `samRef.toString()` it calls `Student::toString()`, not `Person::toString()` like our previous examples did.

The same thing happens if we use the `Person* samPtr`. It is also polymorphic.

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

Polymorphic Functions

What we really want are polymorphic functions like this:

```
// A polymorphic function
void greet(const Person& p)     // any kind of Person
{
    cout << "Hello, I'm " << p.toString() << endl;
}
```

This function is polymorphic because the formal parameter is a **reference to a base class**. (Note, **not** a base class object.) You can **pass any kind** of `Person`, such as a `Student` or an `Employee` object and it will behave appropriately.

> *Polymorphic functions should operate on references or pointers to a base class. Functions should **never use pass-by-value** with base class objects.*

This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.