

Inheritance

Classification is a mechanism which we use to understand the natural world around us. As infants we begin to recognize different **categories**, like food, toys, pets, and people. As we mature, we divide these general classes into **subcategories** like siblings and parents, vegetables and dessert.

When faced with a **new** object, we **understand it** by fitting it into the categories with which we're acquainted:

- Does it taste good? Perhaps it's dessert.
- Is it soft and fuzzy? Maybe it's a pet.
- Otherwise, it's most certainly a toy of some sort!



Encapsulation—the specification of attributes and behavior as a single entity—allows us to build on this understanding of the natural world as we create software. By creating classes and objects that **model categories** in the "real world," we have confidence that our software solutions closely track the problems we are trying to solve.

Once we've designed our own classes, instead of using computer files and variables, our programs can be expressed in terms of **Customers**, **Invoices**, and **Products**.

Introducing Inheritance

Inheritance adds to encapsulation the ability to express **relationships between classes**. Think back to the categories, "desserts" and "vegetables." Cherry pie and broccoli are both, arguably, edible items; for humans, they belong to the **food** class.

Yet, in addition to belonging to the food class, cherry pie is a **kind of dessert**, but broccoli is a **kind of vegetable**. Both **dessert** and **vegetable** represent **subcategories** of foods.

Both cherry pie and broccoli are kinds of food, but, thankfully, the food class itself consists of **more** than just these two items. Cherry pie and broccoli are just two small **subsets** of all possible food types.



Superclass-Subclass

Thus, the **relationship** between food and cherry pie class is one of **superset** (food) and **subset** (cherry pie). In object-oriented terms, we call this the **superclass-subclass relationship**. C++, which has its own terminology, calls it the **base class – derived class**.

Base and derived classes are arranged in a **hierarchy**, with one base class divided into numerous derived classes, and each derived class divided into more specialized kinds of derived classes. That's what we find with the food class.

It can be divided into desserts, vegetables, soups, salads, and entrees. Each category can be further divided into **more specialized** kinds of food.

Specialization & Generalization

A classification hierarchy is based on **generalization and specialization**. Base classes in such a hierarchy are very general, and their attributes few; the only thing that a class must do to qualify as food, for instance, is to provide nutrients.

As you move down the hierarchy, the derived classes become **more specialized**, and their attributes and behavior become more **specific**. Thus, although broccoli qualifies as food (it is, after all, digestible), it lacks the necessary qualifications to make it a dessert.

The Person<-Student Example

Inheritance **introduces quite a few new possibilities** into your programs. It is easy to miss some of the details that you really **must** master to make effective use of the object-oriented technique of inheritance.



So, instead of working with fun stuff, like card games and shooting down aliens, we'll start by returning to the old, boring "finger-exercise" example that lets you concentrate on one piece of the inheritance puzzle at a time. **Click the Running Man** to open the lab example, following along in the CS50 Sandbox.

Extending a Class?

In Java, you use the **extends** keyword to specify the parent or **superclass** (called the **base class** in C++) when you define the child or **subclass** (called the **derived class** in C++). Instead of using the **extends** keyword, as in Java, we **use a colon** in exactly the same position. In addition, we specify that the **base class** is **public**.

```
class Student : public Person
{
    // members of the derived class
};
```

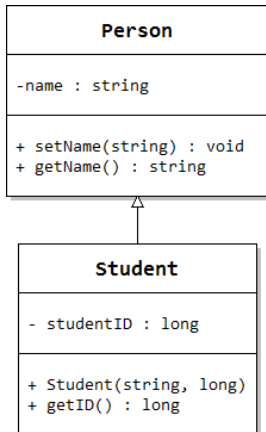
Student is the derived class, while **Person** is the base class. Each of these class definitions is placed in its own header file, with the implementation of the member functions in **person.cpp** and **student.cpp**. You can open these in the Sandbox editor.

The Person Class

The **Person** class represents people, and **Student** is a new (specialized) kind of **Person**. On the left is the **UML (Unified Modeling Language)** class diagram for these classes.

Person is our base class. Each **Person** has:

- a single **data member**, **name**, stored as a **string**. The minus sign preceding **name** tells us that the data member will be **private**.
- one **mutator**, **setName()** that allows you to change the name of the **Person**.
- one **accessor**, **getName()**, which allows you to read the value of **name**.
- The **plus sign** before the member functions indicates that they are **public**.
- The word appearing after the colon is the member function **return type**.



The Student Class

The **Student** class is **derived from Person**.

- The hollow-headed arrow pointing from **Student** to **Person** says **Student** is **derived from** the **Person** class.
- **Student** has one **private** data member, **studentID**, stored as a **long**.
- The class has a **public constructor** that takes two arguments.
- The class has an accessor to retrieve the value in **studentID**.

There are no mutators to set or change the ID. While a student might change their name (because of marriage, for instance) they can never change their student ID.

Inherited Members

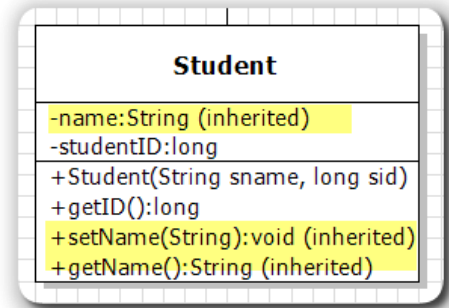
Open **client.cpp** and look at the **main()** function, which creates a **Student** object (**steve**), and then call some of its member functions. Run the project by typing **make run** in the terminal. You'll see something like this:

```
./inherit
getName()->Stephen
getID()->1007
$
```

Of course the **Student** object named **steve** can call the **getID()** member function, which was defined in the **Student** class. No surprises there!

However, it can also call the `setName()` and `getName()` members, which were not defined in `Student`, but in `Person`. More importantly, those member functions can read and change the `name` data member in the `Person` class as if `name` were declared inside the `Person` class. Why?

When you create `Student` objects, each derived class object contains all of the data members and member functions of its base class. If you were to look at a "logical" diagram of the `Student` class, it would look something like that shown here.



However, (very important), the data members will not be directly accessible to the derived class object, because they were declared `private` in the base class.

In the editor, change the `Student` constructor so that it attempts to set the `private name` data member directly, (instead of using the `setName()` member function).

```
Student::Student(const string sname, long sid)
{
    //setName(sname); // comment out this
    name = sname;      // add this
    studentID = sid;
}
```

Type `make` in the console. You'll see an error message that looks something like this:

```
student.cpp:8:3: error: string Person::name is private here:
    name = sname;
    ^~~~
```

Even though you can't access the `private name` data member from the derived class, the data member exists inside the `Student` object nonetheless. You know that because the inherited `setName()` and `getName()` member functions work correctly, and without the `private` data member, that would not be possible.

While private methods are not inherited at all, private instance variables are inherited, in the sense that each derived class object contains a copy of each private variable defined in the base class, even though the methods in the subclass object are not free to directly access that variable.

Inheritance & Constructors



Now that you've learned about inheritance and inherited members, let's look at how **derived-class constructors** are written. We'll with our simple "finger-exercise" example that lets you concentrate on one piece of the inheritance puzzle at a time. Click the **Running Man** to open the lab example, following along in the CS50 Sandbox.

First, though, let's revisit the topic of **private** base-class data members, and consider an alternative.

Protected Members

The member functions and data members which are not declared **private** in the base class are called **inherited** members. An object may use its inherited members without any further qualification, exactly as if they were defined inside the object's own class (as done in the example here.)

A base class **may allow** a derived class access to a data member by using **protected**. Protected members are half-way between **public** and **private**; the derived (child) classes can directly access them, but the general public cannot.



These access specifiers work the same way most of us manage our own households. My grandchildren are free to open my refrigerator, getting a glass of orange juice **without asking me**; you, on the other hand, would have to knock at the front door, and ask first. My refrigerator has something similar to protected access.

On the other hand, even my grandchildren **aren't permitted to grab my credit card** and charge up a storm on the Internet; **my credit card is private**.

In general, avoid using protected access to grant derived classes access to data members. This unnecessarily exposes the implementation of the base class and prevents easy modification. Add some protected member functions instead.

Defining Constructors

A constructor **must** have the same name as the class, so, when you create a new class, it cannot inherit any of the base class constructors. Instead, it defines new ones.

To see how that works, modify the **Person** class, which now uses only the **synthesized default constructor** that is automatically written by your compiler, when you don't supply any explicit constructors:

```
class Person
{
public:
    Person();
    Person(const std::string& pname);
    void setName(const std::string& name);
    . . .
};
```

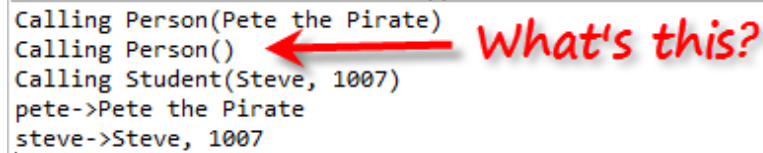
In **person.cpp** add an implementation that prints a message so you can keep track of which constructor is called. The working constructor should use its **string** parameter to initialize the **name** instance variable in addition to **printing a diagnostic message**. Modify the **Student** constructor as well, so it prints a message like this:

```
Student::Student(const string sname, long sid)
{
    setName(sname);
    studentID = sid;
    cout << "Calling Student(" << name << ", "
         << sid << ")\n";
}
```

Modify **main** inside **client.cpp** to create two objects, one **Person** and one **Student**, and to print out their info, just like the previous example.

```
Person pete("Pete the Pirate");
Student steve("Steve", 1007);
cout << "pete->" << pete.getName() << endl;
cout << "steve->" << steve.getName() << endl;
```

Type **make run** to compile and run the modified program. You'll see that **instead of two constructors**, both **Person** constructors have been called, along with the **Student** constructor, for a total of three, even though **only two objects are created**. Why?



```
Calling Person(Pete the Pirate)
Calling Person()
Calling Student(Steve, 1007)
pete->Pete the Pirate
steve->Steve, 1007
```

Constructor Chaining

Before a derived class constructor can do **any** of its work, it must first initialize **all of the base class data members**. This must happen **before** the derived constructor ever runs. (If this sounds familiar, it should. It is the same reason that your class data members are already initialized before the first line of your constructor runs!)

You needn't do anything special to make this happen. When your constructor runs, it **implicitly calls the default or no-argument constructor** in the base class as its first line of code, and that constructor calls **its** base class constructor, and so on, all the way up to the first class in your hierarchy. This is called **constructor chaining**.

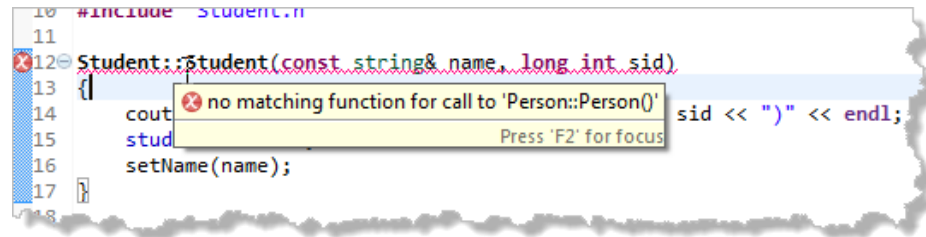
So, consider for a second, the **Student** class constructor in the previous section. Even though not explicit in the source code, the following commands are executed when the **Student** constructor is invoked (after memory for both the **Person** part of the student and the **Student** part of the student has been allocated).

1. call the **Person** default constructor
2. call the **setName()** inherited method
3. assign the **sid** parameter to the instance variable
4. print the diagnostic message

That's why you saw **three diagnostic messages** when you only created two objects. When you created the **Student** object named **steve**, the **Student** constructor first called the **default** constructor for the **Person** class.

Default Constructors

What happens if you **remove** the default constructor from the **Person** class? Comment it out and you'll see that the **Student** constructor **stops working**:



As the error message shows, every time you extend a class, the superclass **must**:

1. have an **explicit default constructor** like **Person()**, **or**
2. have a **synthesized default constructor** which is automatically written if the base class has **no** explicit constructors, **or**
3. the **derived class** (**Student** in this case) must **explicitly** call **another** constructor in the base class.

How exactly do you explicitly call a superclass constructor?

Explicit Base Class Initialization

Just as you can initialize data members before the constructor runs, you can initialize your object's "base part" by **calling the base class constructor in the initialization list**.

The **Student** constructor invokes the **Person(String)** constructor, instead of using the **setName()** method, as you've done up until now. This is the **normal way to write derived class constructors**.

```

Student::Student(const string sname, long sid)
    : Person(sname)
{
    setName(sname);
    studentID = sid;
    cout << "Calling Student(" << name << ", "
         << sid << ")\n";
}

```

Now, when you run our sample program, instead of **implicitly** calling the **Person** default constructor (which no longer exists), we **explicitly** chain to the **Person(string)** constructor to initialize the name field.

Overridden Members



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**. We'll use our simple "finger-exercise" example that lets you concentrate on one piece of the inheritance puzzle at a time. Click the **Running Man** to open the lab example, following along in the CS50 Sandbox.

Virtual Member Functions

A derived class **may override** member functions in the base class. The base class must permit that by using **virtual** on the prototype. Modify 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 **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.

The implementation of **toString()**, in **person.cpp** should display the person's name:

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

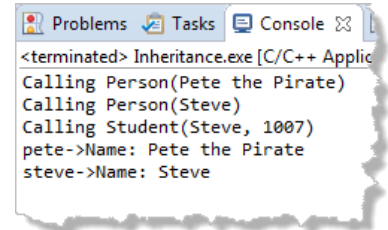
The **Student** class inherits **Person::toString()**. If the **Student** class does nothing else, then there is no difference between a **virtual** function and a regular function.

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.



Overriding the Method

When another class (like `Student`) wants to provide a different implementation for a method like `toString()` it must:

1. Use **exactly the same signature** (number and type of parameters) as the original method 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 method.

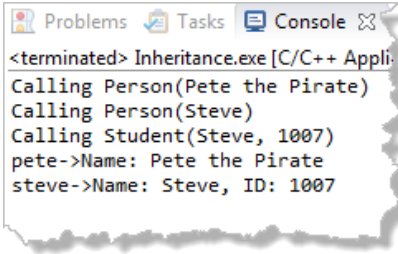
Let's override `toString()` in the `Student` class. Here's the header:

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

You **do not need to** repeat the word `virtual` in the derived class (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.

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 `toString()` 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 the `toString()` method, and just **add on** the new parts you want, like the `studentID`? Is there some way you can **combine** inherited and overridden methods?

Yes, there is.

Calling Superclass Methods

`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 the 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.

*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.*

The override Contextual Keyword

While always a **logic error** for a derived class to redefine a non-virtual function, C++ 11 added new **contextual keywords** that **allow the compiler to catch such logic errors** that previously were often hidden.

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

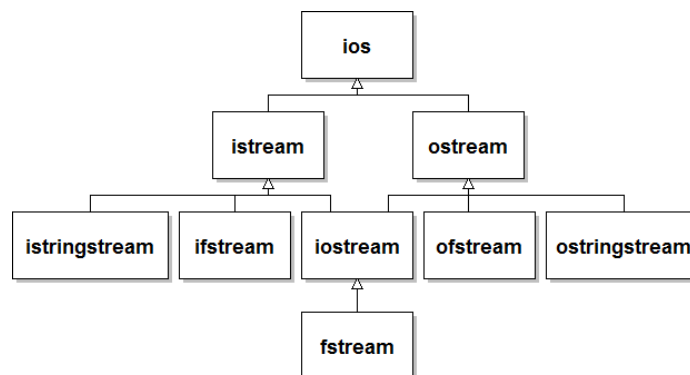
```
std::string toString() const override;
```

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

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

Substitutability

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**.

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 the same as with the **cout** object:

```
ofstream out{"myfile.dat"};
int x{10};
out << "x->" << x << endl;
```

You've also seen how **operator<<** can be overloaded for a user-defined type. For a **Point** structure, say, you can easily write this:

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

This works with **cout** and **cerr**, both of which are **ostream** objects. So, what do you have to do to adapt the functions so that it works with **ofstream** objects and maybe even **ostringstream** objects?

The answer, perhaps surprisingly, is **you do not have to do anything**; it already works with **ofstream** objects just as it does with **ostream** objects like **cout**. But, how can this be, since the **cout** object has type **ostream** which is not the same class as **ofstream**?

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 **ostream&**, **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. 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.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
 - a. Make sure you **submit** the assignment using **make submit**.
 - b. Make sure you check the [CS150 Homework Console](#) to see that your scores got reported, **before** the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.