

Assign, Copy & Destroy

You may assign one object to another, just as you can assign one `int` variable to another, even though the data members are `private`. As with the built-in types, the result is a **copy** of the data in the original members.

With class types, this is carried out by the overloaded **assignment operator**.

```
Time& operator=(const Time& rhs);
```

The assignment operator **is not used** when you pass an object by value, or **initialize** a new object variable with another. Instead, the **copy constructor** is called:

```
Time(const Time& rhs);
```

When an object is destroyed, its **destructor** is called. If your class allocates dynamic memory in the constructor, for instance, you would free it in the destructor. The destructor looks like the default constructor preceded by the **tilde**:

```
~Time();
```

C++ automatically writes a *synthesized* assignment operator, a *synthesized* copy constructor, and a *synthesized* destructor which work for simple types such as those in this course. In CS 250 you'll learn how to write your own assignment operators, copy constructors, and destructors to create more dynamic types.

You can **prevent** pass by value or assignment by adding the following to the definitions.

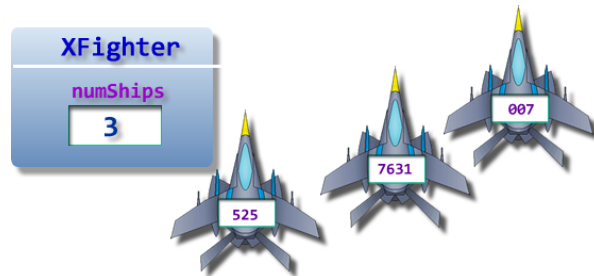
```
Time& operator=(const Time& rhs) = delete;  
Time(const Time& rhs) = delete;
```



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 Data Members

Suppose you are creating a space-wars type video game, and one of your player types is an **XFighter** class. How do you easily keep track of **how many XFighter** ships are currently available?



That's easy: with a **static**, or **shared data member** as so:

```
class XFighter
{
public:
    XFighter(int vin) : m_VIN(vin) { numShips++; }
    ~XFighter { numShips--; }
private:
    static int numShips;
    int m_VIN;
};
```

The **static** data member **numShips** is **private** to the **XFighter** class, but there is **only one copy** of the member, **not one for each ship**, like the vehicle identification number (**m_VIN**). When an **XFighter** is constructed, the constructor increments the shared **numShips**, and, when a ship is destroyed, the destructor decrements it.

There is one wrinkle with this. In C++, you must **initialize the static member** as a global object in your **.cpp** file with something like this:

```
int XFighter::numShips = 0;
```

This **cannot** go in the header file.



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 Member Functions

How do you use the `numShips` variable, since it's `private`? That is, how do you find out how many ships exist? With a **static member function**, like this:

```
static int xFighters() { return numShips; }
```

A `static` member function can **only** access `static` data members, or call other `static` member functions. It **cannot** access regular data members or call regular member functions. If the function is defined **outside of the header**, then you **do not** repeat the keyword `static` in the definition:

```
int XFighter::xFighters() { return numShips; }
```

Place `static` member functions in the `public` section of your class and call them using the **class name** and the **scope operator**, **not** an instance and the dot operator:

```
cout << XFighter::xFighters() << endl;
```



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 const Data Members

When you have a constant that only applies to the situation represented by your class you can add it as a static const data member.



```
class Bizarro
{
public:
    static const int kAnswer{-42};
    static constexpr double kE{3.14159};
    static const double kPi;
};
```

In the Bizarro world, almost **everything is backwards**. The "answer to everything" is **-42**, not **42** as in our world. The mathematical constant **e** is **3.14159**, and the constant **PI** is the square root of **PI** in our own world.

For **integral types**, you may initialize **static const** data members inside the class (as with **kAnswer**); no other initialization is required. Starting in C++11 you can also do this for other types, using the keyword **constexpr**, instead of **const** (as with **kE**), provided that the value can be calculated at compile time.

However, if a type needs to calculate a value at runtime, (as **kPi** does), you'll still need to provide a separate **definition** in the **.cpp** file.

```
const double Bizarro::kPi = sqrt(acos(-1.0));
```

The data members are **static** (there is **only one copy** stored), they are **const**, (they **cannot be changed**), and they are **public** (you can use them outside of the class.)

```
cout << Bizarro::kAnswer << endl;
cout << Bizarro::kPi << endl;
cout << Bizarro::kE << endl;
```



This course content is offered under a [CC Attribution Non-Commercial](https://creativecommons.org/licenses/by-nc/4.0/) license. Content in this course can be considered under this license unless otherwise noted.

Classification

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**.



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.

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.

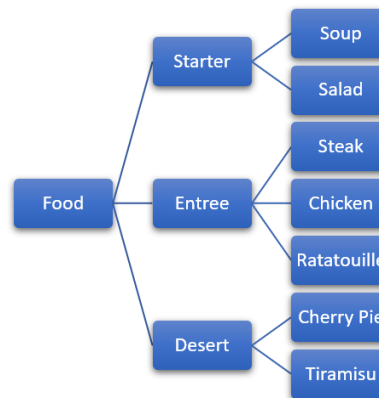
Yet, in addition to belonging to the food class, cherry pie is a **kind of dessert**, but broccoli is a **kind of vegetable**. Both **dessert** and **vegetable** represent **subcategories** of foods. Both cherry pie and broccoli are kinds of food, but, thankfully, the food class itself consists of **more** than just these two items. Cherry pie and broccoli are just two small **subsets** of all possible food types.

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

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.

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.




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.

Person < - Student

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 in **Replit**. **Fork the Repl** so you'll have your own copy to work on.

Extending Person

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 Replit editor.



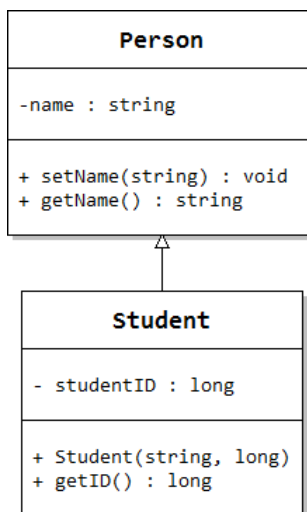
This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

The UML Diagram

The **Person** class represents people, and **Student** is a new (specialized) kind of **Person**. On the right 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 retrieve the value of **name**.
- The **plus sign** before the member functions indicates that they are **public**.
- In each entry, the word appearing after the colon is the data member type, or member function **return type**.



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

- In the UML diagram, 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.



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.

Inherited Members

Open `main.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:

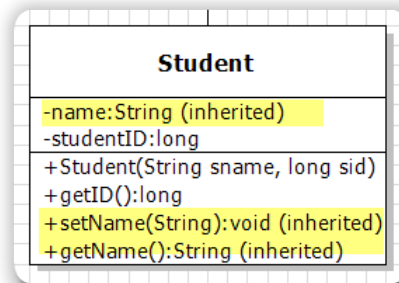
```
./inherit
getName() -> Stephen
getID() -> 1007
```

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()` member functions, 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 `Student` 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.



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.

Private Base-Class Members

Private base-class data members are not directly accessible to the derived class object member functions. To see how this works, 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 this out
    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 **Student** class, the data member **exists inside the Student object nonetheless**. You know that is true, because the **inherited setName()** and **getName()** member functions work correctly, and without the existence of the **private** data member, that would not be possible.

*Private data members **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 member functions in the derived object are not free to directly access that variable.*



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.

Base-class Constructors

 Now that you've learned about inheritance and inherited members, let's look at how **derived-class constructors** are written. We'll use our simple "finger-exercise" example that lets you concentrate on one piece of the inheritance puzzle at a time. You can re-open it directly from [Replit](#), or you can **click the Running Man** to open my copy, and **Fork** it again.

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. In **person.h** add this code:

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

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 **m_name** data member in addition to **printing a diagnostic message**.

```
Person::Person() { cout << "Calling Person()" << endl; }

Person::Person(const string& pname)
{
    cout << "Calling Person(" << pname << ")" << endl;
    name = pname;
}
```



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.

Derived-class Constructors

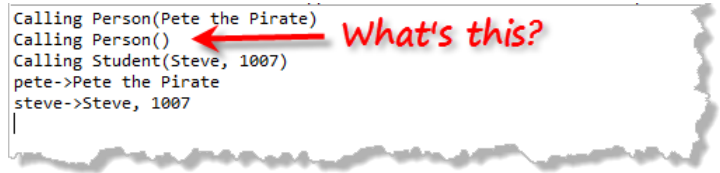
The derived **Student** class already has a constructor. Go ahead and modify it as well, so it prints a message like this:

```
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 existing 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 only two constructor calls, which you'd expect, **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
|
```



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.

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, if **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 member function
3. assign the **sid** parameter to the data member
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.

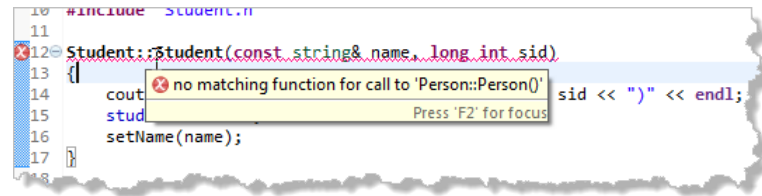


This course content is offered under a CC Attribution Non-Commercial license. Content in this course can be considered under this license unless otherwise noted.

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.

So, how exactly do you explicitly call a superclass constructor?



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.

Calling the Base Constructor

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 initializer list**.

When you do this, the **Student** constructor invokes the **Person(String)** constructor, instead of using the **setName()** member function, 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)    // this initializes name
{
    // setName(sname); // Remove this!!!
    studentID = sid;
    cout << "Calling Student(" << name << ", "
         << sid << ")\n";
}
```

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

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.

A base class **may allow** a derived class access to a data member by using the keyword **protected** instead of **private**. 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.*



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.