Chapter

28

Polymorphism

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

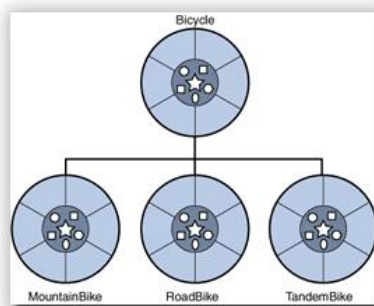
- Build it **completely from scratch**, using only built-in components.
- **Combine simpler types** to create complex types. This is **composition**.
- **Extend** a general class. adding new features. This is called **inheritance**.

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

Has-A and Is-A

These strategies express two kinds of "class relationships":

- 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. This is also called an **inheritance** relationship. In the inheritance relationship shown here, we'd say that a `MountainBike` **is-a** `Bicycle`.



Polymorphic Inheritance

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 chapter and see what happens with some experiments.

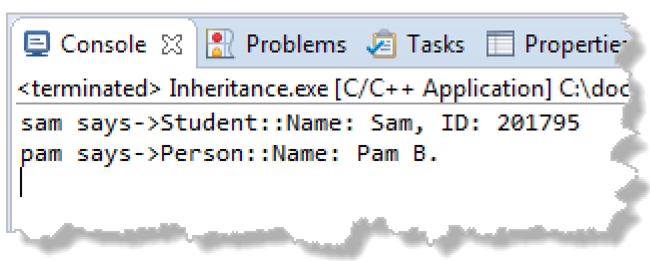
A Student *is-a* Person

Click the Running Man on the left to open a copy of the lab. 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**. I've highlighted the changes.

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

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

Use `make run` to see the client 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**.



Console ⊠ Problems Tasks Propertie

<terminated> Inheritance.exe [C/C++ Application] C:\doc

```
sam says->Student::Name: Sam, ID: 201795
pam says->Person::Name: Pam B.
```

This is not what we mean when we talk about polymorphism. This works just as if `Person` and `Student` were not related in any way.

Polymorphism means an inheritance relationship where the request can be sent to any kind of `Person` object, and the specialized `Person`s, such as `Student`s or `Employees` respond appropriately.

The Slicing Problem

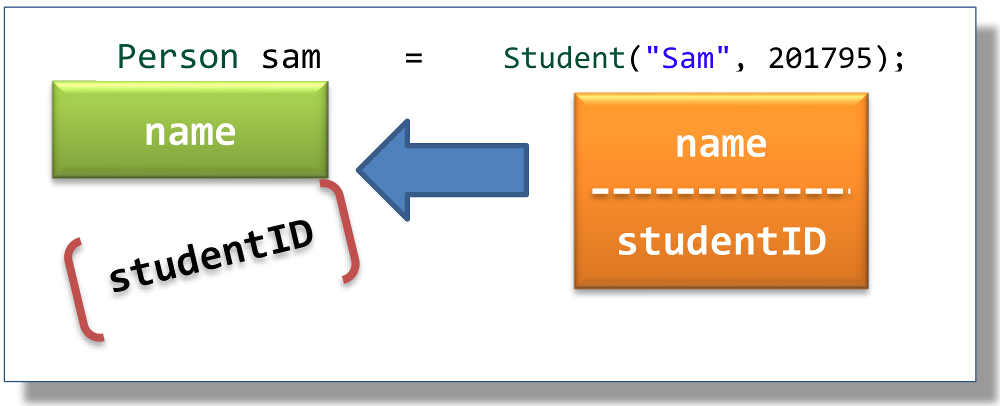
Change the example (`client.cpp`) again, so it looks like this:

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

Now you have two **Person** objects: one "plain" **Person**, a one specialized **Person** who is a **Student**. It the output the same as previously? **Of course not!**

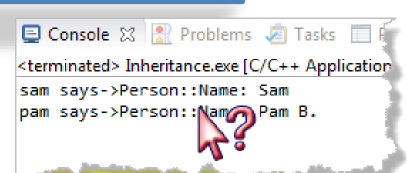
For the **Student sam**, you know longer see the **ID**. And, both the **Student** and the **Person** are identified with **Person::Name**, even though we do have an overridden member function, **Student::toString()**. Here's why this happens.

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!



No Polymorphic Objects

While slicing is a problem it is not the real culprit. The problem is **polymorphism only works in C++ with references or pointers**. Fix the **client.cpp** like this:

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

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 `Student` or `Employee` objects and they 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.*

Polymorphic Lists

Creating a list (`vector` or array) of different kinds of object also leads to slicing:

```
vector<Person> v;
v.push_back(Student("Sam", 201795));
v.push_back(Person("Pam B."));
```

When you `push_back` the `Student` or `Employee` objects, they are sliced as well. The `vector v` does not contain a `Student` and a `Person`; it contains two `Person` objects. Sam has been stripped of everything that makes him a `Student`; he has been effectively lobotomized; he no longer knows who he is.

No Lists of References

You also cannot fall back on using references, like you did with polymorphic functions, since you cannot create a `vector<Person&> v` or an array `Person& a[3]`. Both of these declarations are illegal. A reference is not a variable or object (*`lvalue`*), but an **alias** for an existing *`lvalue`*.

Pointers to the Rescue?

One solution is to create a **vector<Person*> v** or array **Person * a[2]**. Here's a short example that places two different kinds of **Person** pointers in a **vector** and prints them. Each person responds appropriately. Go ahead and add this code to **main()**. Include the **<vector>** header.

```
int main()
{
    vector<Person*> people;
    people.push_back(new Student("Sam", 201795));
    people.push_back(new Person("Pam B."));

    for (auto p : people) {cout << p->toString() << endl;}
    for (auto p : people) delete p;
}
```

Since two of these objects are created on the heap, it is up to you to reclaim their memory before the **vector** goes out of scope and it is lost. The **vector** cannot do it because it does not know if the pointers it contains point to objects on the heap or objects on the stack. If you add a stack-based pointer to this program, it crashes.

Smart Pointers to the Rescue

You can eliminate the need to reclaim memory by using the two C++11 smart pointers, **shared_ptr** and **unique_ptr**, which are declared in the header **<memory>**. Here is a version that uses **unique_ptr**, which doesn't require the **delete** loop at the end.

```
using up = unique_ptr<Person>;
vector<up> people;
people.push_back(up{new Student("Sam", 201795)});
people.push_back(up{new Person("Pam B.")});
for (auto& p : people) cout << p->toString() << endl;
```

The **unique_ptr** constructor is **explicit**, so a **type alias** helps to shorten the name. Also, **unique_ptr** objects cannot be copied, so the range-based **for** loop accesses each element by reference, which is what you need anyway.

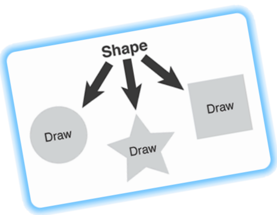
Because the **vector** now stores **only smart pointers**, you can't inadvertently add a stack-based pointer to the list.

Inheritance Forms

What would happen if you were to remove the keyword `virtual` from the definition of the `toString()` member function in the `Person` class? Your code would still compile, but the `toString()` function would no longer be overridden; it **would be hidden** in the derived class `Student`.

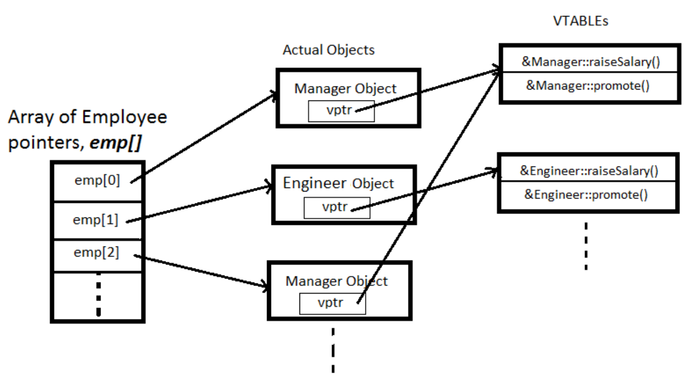
Functions are **bound** to an object depending on how they are declared. A non-virtual function is **bound at compile time** to the class that it is defined in. A non-virtual function defined in the `Person` class (such as `getName()`), will always be bound to the `Person` class, and cannot be overridden in any subsequent classes.

This is called **early binding** (or compile-time binding).



When you add the keyword `virtual` to a function, the function call is not determined (or bound) at compile time, but **when the program is run**. Instead of looking at the type of the pointer or reference used in the function call, **the actual object pointed to** is used to decide which function to call. This decision is made when your program runs. If your `Shape*` points to a `Circle` object, then `Circle::draw()` will be called, but **only if** `draw()` is a `virtual` function.

This is called **late-binding** or **dynamic dispatch**.[1] In Java, **all** methods use late binding, but in C++ you, as the base-class designer get to decide which version to use, through the application of the keyword `virtual`.



Virtual member functions are implemented by adding a new pointer to every object that contains at least one `virtual` function. This pointer is called a **vptr** and it points to a table of functions, called a **vtable**[2]. The *vtable* contains the actual addresses of the functions to be called for that class.

---

[1] Wikipedia – Dynamic Dispatch: http://en.wikipedia.org/wiki/Dynamic_dispatch

[2] Wikipedia – Virtual Method Table http://en.wikipedia.org/wiki/Virtual_method_table

Using this illustration, let's see how late binding, or dynamic dispatch works:

1. You call `emp[0]->raiseSalary()`
2. Your call is routed though the **vptr** in `emp[0]`, which is actually a `manger`, and eventually finds the address of the `Manager::raiseSalary()` function inside the `Manager` **vtable**.
3. You call `emp[1]->promote()`
4. Your call is routed through the **vptr** in `emp[1]`, which is actually an `Engineer`. This **vptr** points to the `Engineer` **vtable** where it finds the `Engineer::promote()` method.

Multiple Inheritance

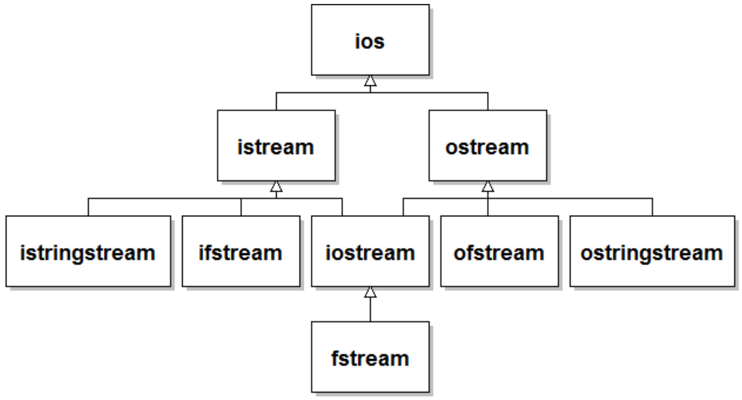
C++ includes a capability known as **multiple inheritance**, which allows a class to be derived from two (or more) base classes. Multiple inheritance lets you create a class `AmphibiousVehicle` from the parents `LandVehicle` and `WaterVehicle`.

An `AmphibiousVehicle` inherits wheels or treads from its `LandVehicle` parent, and a prop or screw and rudder from its `WaterVehicle` parent. You can't do this in Java, because Java classes can have only one parent class.

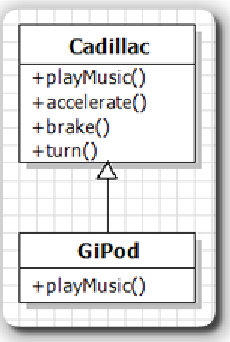
MI in the Stream Libraries

The standard stream libraries include classes that are **both** input and output streams.

The `fstream` class at the bottom is an `iostream`, which is in turn—if you follow the arrow leading up and to the left—an `istream`. The `fstream` class therefore inherits all the methods that pertain to `istreams`. If you instead follow the arrow up and to the right, you discover that the `fstream` class is an `ostream`, which means that it inherits these methods as well.



Implementation Inheritance



Suppose you have a class which simulates a `Cadillac`. It has an exceptionally fine sound system, which required a lot of effort to implement and of which you're especially proud. Now you want a `GiPod` class, for sale over the Web:

Because you've already created the `Cadillac` class, why not just create a derived class, and then eliminate all the member functions that have nothing to do with playing music, **transforming the car into a mere sound system**?

To reuse the code you've already written, you replace `brake()`, `accelerate()`, and all the other "extra" methods from the `Cadillac` class with empty braces. In traditional computer-science terms, you replace them with a **NOP** (No OPeration).

This practice, called **contraction**, is a trap! You should avoid doing this for two reasons:

- You're **violating the substitutability rule**. You will undoubtedly break some code that relied on all `Cadillac` objects carrying out certain operations.
- It's **more work than doing the right thing!**

Private Inheritance

Private inheritance is one way to solve this problem. Private inheritance means you want to **inherit the implementation of a class**, **not the interface.** A class has some functionality that you want to exploit, but you don't want to use the interface of the base class.

```
class GiPod : private Cadillac {};
```

`GiPod` objects would not, in the **is-a** sense, be `Cadillac` objects. Calling any "inherited" member functions would fail. To call any of the inherited methods, you must add those methods to the new interface, with a `using` declaration like this:

```
class GiPod : private Cadillac
{
public:
    using Cadillac::playMusic;
};
```

You don't need to specify the arguments or supply an argument list. You do need the `public:` if you want the name moved into the `public` section.

Doing this, you have reduced the interface to a single method. The relationship between the `Cadillac` and `GiPod` classes is one of "**implemented in terms of**", not "**is-a**".

Composition Revisited

Perhaps a better solution is to use **composition**. **Composition creates a new class by combining simpler classes**, using instances of the simpler class as the data members. In composition, an object is **composed of other objects**, which make up its "parts." That's why it's called (informally) a **has-a** relationship; because we can say that:

- A car **has a** motor, or
- A bicycle **has a** seat, or
- A computer **has a** CPU

Here is version of the `GiPod` that uses composition; note that it needs to **explicitly write the prototypes** for any methods that it uses.

```
class GiPod
{
public:
    void playMusic() const
    {
        caddy.playMusic();  // forward or delegate
    }
private:
    Cadillac caddy;         // data member
};
```

Specification Inheritance

A final form of inheritance is called specification inheritance. A base class may specify a set of responsibilities that a subclass **must fulfill**, but not provide any actual implementation. **The interface** (method signatures) are inherited.

The specification relationship is used **in combination** with regular specialization:

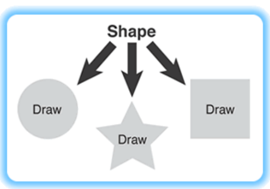
- the derived class **inherits the interface** of the base class, as in specification
- it also inherits a **default implementation** of, at least, some of its methods

A derived class **may override** a `virtual` member function to add specialized behavior, as we did with `Student::toString()`, **or**, it may be **required** to implement a particular member function, which could not be provided in the base class.

Abstract Classes

The classes we've used so far are called concrete classes. We can also create **abstract classes**. An abstract class is usually an **incomplete** class, a class that contains certain methods that are **specified**, but not **implemented** in the definition of the class.

Because the abstract class contains these incomplete methods, **it cannot be used to create objects**—it can only be used as a base class when defining other classes. That is, it **only** makes sense in the context of **polymorphic inheritance**.



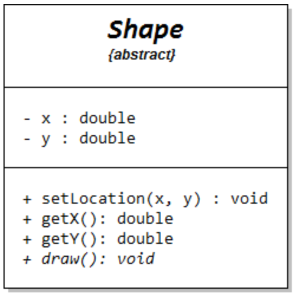
Suppose you have an **abstract** `Shape` base class that doesn't have the faintest notion of how to implement its abstract `draw()` method, yet it knows that each of its **concrete** derived classes will need to do so.

Only the **concrete** classes derived from `Shape`—`Circle`, `Square`, and `Star`—possess the necessary knowledge to actually `draw()` themselves. You, only need to program in terms of `Shape` objects; the actual shapes will take care of their own behavior.

Pure Virtual Functions

In C++, an **Abstract Base Class** (or **ABC**) is any class that has one, or more, **pure virtual member functions**, created using the following syntax in the prototype:

```
class Shape     // abstract class
{
public:
    ...
    // Pure virtual function (abstract method)
    virtual void draw() const = 0;
    ...
};
```

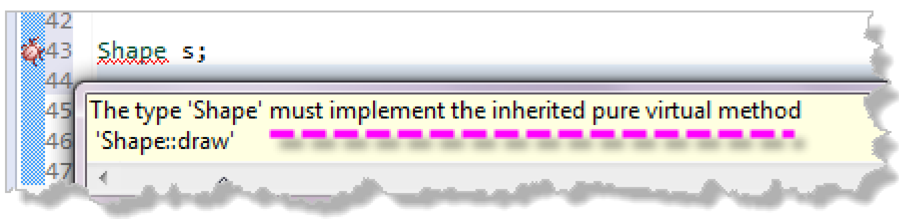


Think of the **= 0;** part of syntax as a **replacement** for the **abstract** keyword in Java. Abstract classes are **not restricted** to abstract member functions like **draw()**; you can have as many regular (concrete) member functions as you'd like, freely mixed with your abstract methods. The **Shape** class in the UML diagram at the left has a **setLocation()** member function. In UML, abstract methods, such as **draw()**, are drawn using italics.

Your concrete methods may **call** abstract methods as part of their definition, even though the member function is never implemented in the base class. C++ pure virtual functions **may have an optional implementation**. Since you cannot create an instance of an abstract base class, you must call this implementation from a derived class.

Using an Abstract Class

It is illegal to create an instance of an abstract class. Your compiler enforces this. You may, however, create **ABC** pointers or references as long as they point to, or refer to concrete objects which are derived from the **ABC**.



When you **extend an abstract class**, your derived class **must override each and every abstract function in its base class**, giving each a concrete implementation. The resulting derived class is a **concrete class**, and it can be used to create new objects.

Abstract classes thus provide a way of **guaranteeing** that an object of a given type will understand a given message. In that sense, **they specify a set of responsibilities that a derived class must fulfill**.

A Triangle Example

Let's look at an example. To create the **Triangle** (or **Circle** or **Square**) classes, using the abstract **Shape** class as the base class, all you need do is:

```
class Triangle : public Shape
{
public:
    // MUST override; pure virtual in Shape class
    void draw() const;
};

void Triangle::draw() const { /* your code here */ }
```

1. Specify the `Shape` class as the `public` base class in the class header.
2. Provide an implementation for **every** abstract method in the `Shape` class.

For `Triangle` that means you **must** define a `draw()` member function where indicated by the comments. In reality, you'll probably do a lot more; `Circle` might have a `radius` member, `Square` class could have members for the `size` of each side, and the `Triangle` class could have members for `base` and `height`.

Final Members

A derived class may **replace** an inherited member function, which the class designer wanted left alone. In the `Person` class, `getName()` returns the name of the person. **This works fine as is**; there's no reason for a derived class to change it.

However, **nothing prevents a derived class from redefining it** like this:

```
class Imposter : public Person
{
public:
    string getName() const
    {
        return "Emperor " + Person::getName();
    }
};
```

The derived class has no access to the `private` member `name`. However, because the `getName()` member function can be **redefined**, the derived class was able to effectively gain access to this field. And, because of the **principle of substitutability**, the `Person` that your function receives as a parameter may actually be an `Imposter`.

To prevent this, when you design a base class, consider which functions you want to allow others to extend and which ones should be "set in stone". No one should ever change `getName()`, so you can **seal it using `final`** like this:

```
class Person
{
public:
    virtual std::string getName() const final;
};
```

When a member function is marked **`final`** then derived classes are **prevented** from overriding it and we would see an error message like this:

```
Person.cpp:33:12: error: virtual function 'virtual
     std::string Imposter::getName() const'
     string getName() const
            ^
Person.cpp:21:8: error: overriding final function
     'virtual std::string Person::getName() const'
     string Person::getName() const { return name; }
        ^
```

Final Classes

Only **`virtual`** functions can be marked **`final`**. When designing a collection of classes, you normally **won't want all of the classes to be extensible**. To prevent others from extending your class, add **`final`** to the class header, just like you did for the method.

If you make a class final**, then there is no reason to make the methods** final **as well.**

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.