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));    // OOPS!!!
v.push_back(Person("Pam B."));
```

When you `push_back` a `Student` or `Employee` object, **the object is sliced** when it is copied into the `vector`. 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.

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 an 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.

Early & Late Binding

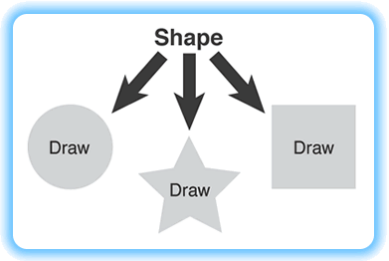
What would happen if you were to remove the keyword `virtual` from the definition of the `toString()` member function in the `Person` class? Your code would still compile, but the `toString()` function would no longer be overridden; it **would be hidden** in the derived class `Student`.

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 binding is not determined 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.

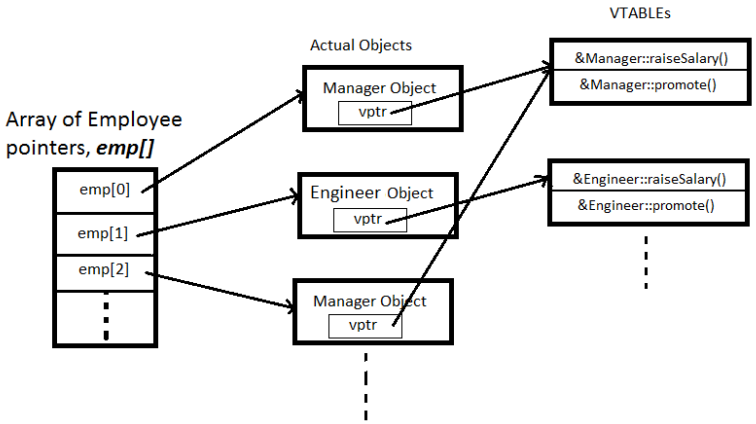
This decision is made when your program runs. If your `Shape*` points to a `Circle` object, then `Circle::draw()` will be called, but **only if** `draw()` is a `virtual` function.



This is called **late-binding** or **dynamic dispatch**. 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`.

How Late Binding Works

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*** or Virtual Method Table. The ***vtable*** contains the actual addresses of the functions to be called for that class.



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 your call 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()` function.

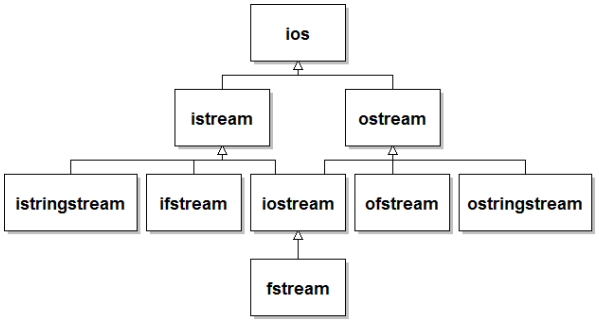
Multiple Inheritance

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.

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`.

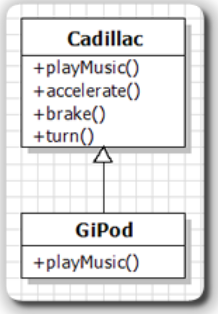
The `fstream` class therefore inherits all the member functions that pertain to `istream` objects. If you instead follow the arrow up and to the right, you discover that the `fstream` class is an `ostream`, which means that it inherits these member functions as well.



Contraction

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 to reuse that sound system in a portable `GiPod` class: 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**?



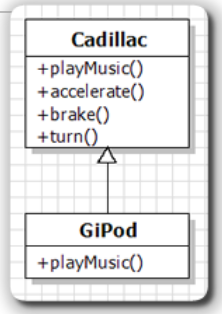
To reuse the code you've already written, you replace `brake()`, `accelerate()`, and all the other "extra" methods from the `Cadillac` class with empty braces. In traditional computer-science terms, you replace them with a **NOP** (**N**o **OP**eration).

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!**

Let's look at how you can use **private inheritance** and **composition** to do this correctly.

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.** Use this when a class has some functionality that you want to exploit, but you don't want to use the interface of the base class.

To use private inheritance,replace the keyword **`public`** with **`private`**, or, simply omit it altogether. I'd recommend that you explictly specify **`private`**to prevent subsequent maintenance programmers from "fixing" your code inadvertently.

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

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

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

When you "import" a member function like this, 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.

Composition

Instead of using private inheritance, 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 some code for a version of the `GiPod` which uses composition;
note that it needs to **explicitly write the prototypes** for any member functions that it uses.

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

Specification Inheritance

In English, the word "inheritance" has several meanings. There is biological inheritance, where you inherit you eye color from your parents, but there is also cultural inheritance and legal inheritance.

Similarly, in C++ you've seen **specialization** inheritance, **polymorphic** inheritance and **implementation** inheritance. In this lesson, we'll look at another form of inheritance, called **specification inheritance**.

With specification inheritance, a base class may **specify a set of responsibilities** that a derived **must fulfill**, but not provide any actual implementation. **The interface** (method signatures) are inherited. This is similar to legal inheritance, where your grandparents may leave you some money to be used only for college.

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.

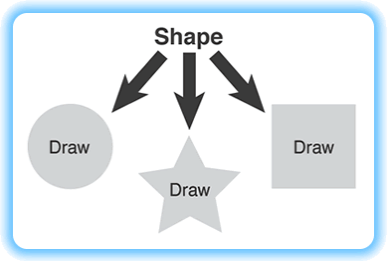
Abstract Classes

The classes used in specification inheritance are called abstract classes. The classes we've used so far are called **concrete** 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.

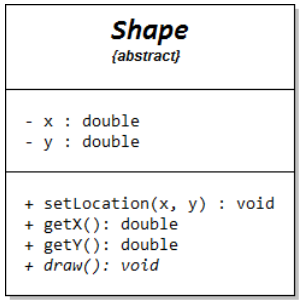


Creating an Abstract Class

Unlike Java and Python, C++ has no `abstract` keyword. Instead, 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 and Python.



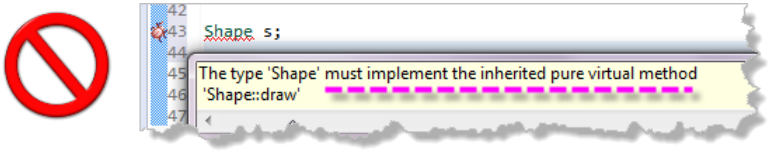
Abstract classes are **not restricted** to abstract member functions like **`draw()`**; you can have as many regular (concrete) member functions as you'd like, freely mixed with your abstract methods.

The **`Shape`** class in the UML diagram at the right 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. Unlike Java, C++ pure virtual functions **may have an optional implementation**. Since you cannot create an instance of an abstract base class, you could only 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 mustfulfill.**

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** pure virtual function (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, the **`Square`** class could have members for the size of each **`side`**, and the **`Triangle`** class could have members for **`base`** and **`height`**.

Redefining Functions

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`.

Final Classes

To prevent `Imposter` classes, 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; }
      ^
```

Only **`virtual`** functions can be marked **`final`**, (which is really annoying). 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.