

Inheritance (Chapter 4)

In C++, we say that a class A inherits from class B, which is to say that B is a *base class* for A, and A is a *derived class* of B. Inheritance is a composition mechanism, which allows the programmer to define a composite type, which consists of a base class and one or more derived classes. At run-time, the composite type results in a composite object.

In C++, inheritance is used to accomplish two programming goals:

1. code reuse
2. subtyping

Code reuse arises when we find that some class implements much of the functionality that we need, but we want to add some additional functions. So, rather than changing the class, we make it a base class of a new derived class that implements the incremental functionality.

Often, a derived class is called a “subtype” of a more “abstract” base class, meaning, the derived class is a specialization of an already defined type. For example, a Vehicle type is abstract, but can be specialized (made more concrete) by deriving a Car type, or a Truck type, or a Motorcycle type, from the Vehicle type.

C++ implements three forms of polymorphism (the first two we have already seen):

1. *ad hoc polymorphism* or operator overloading
2. *parametric polymorphism* or template classes/functions
3. *subtype polymorphism* or subtyping using inheritance and “virtual functions”

Like operator overloading and templates, inheritance is a static, compile-time mechanism but virtual functions are a dynamic, run-time mechanism.

Inheritance Example

When using inheritance, the inherited class is called the *base* class, and the inheriting class is called the *derived* class. When a derived class is constructed, the base class constructor is executed prior to the derived class constructor. Thus the derived class can pass arguments to the base class constructor using the initializer list.

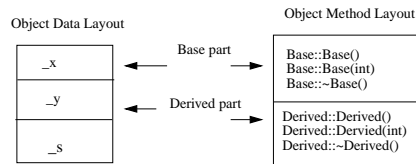
```
class Base {
private:
    int _x;
public:
    Base() : _x(0) {}
    Base(int x) : _x(x) {}
    ~Base() {}
    ...
protected: // protected data and methods only visible to subclasses
    void print(ostream& s);
};

class Derived : public Base { // default is private inheritance if public not specified
private:
    int _y;
    char* _s;
public:
    Derived() : Base(), _y(0), _s(NULL) { ... }
    Derived(int x, int y) : Base(x), _y(0), _s(NULL) { ... }
    ~Derived() {}
    ...
    void print(ostream& s);
};
```

Question: What is sizeof(Derived)?

Composite Object Layout

Inheritance results in a “composite” object at run-time. Public methods are combined into one set of public methods, but with distinct names qualified by the class name in which they are defined. Data layout is combined as well, with compile-time access control (e.g., private) still applies:



With inheritance, private data in the base class is still private. C++ introduces the “protected clause” for allowing a derived class to have special access privileges to the base class. Protected data and methods are only visible to base classes. Use protected data and methods to selectively break encapsulation so that the subclass can access the data and/or methods.

NOTE: when you construct a composite object, the base class constructor is executed before the derived class constructor. That is objects are constructed from the base down the inheritance hierarchy. Usually, destruction is the reverse--a derived class destructor is executed before a base class destructor. We will see a case later where this is not always the case.

Using Base Class Methods

1. you can inherit a method as is and call it as though it were one of your member methods.
2. you can overload a method with a new implementation.
3. you can extend them with a method of the same name that calls the inherited method at some point.

```
class Base {
private:
    int _x;
public:
    ...
protected:
    // protected data and methods only visible to subclasses
    void print(ostream& s) { s << _x; }
};

class Derived : public Base {
private:
    int _y;
    char* _s;
public:
    ...
    void print(ostream& s) { Base::print(s); s << _y; }
};

Base b;
b.print(cout); // ERROR! Base::print protected

Derived d;
d.print(cout); // OK, Derived::print public
```

Virtual Methods

Virtual methods are identified by a special keyword “virtual” prefixing a method declaration. Virtual methods defined in a class that be overridden by a subclass.

```
class Base {
public:
    ...
    virtual void print(ostream& s) { s << "hello, world"; }
};

class Derived : public Base {
public:
    ...
    virtual void print(ostream& s) { s << "goodbye, world*"; }
};
```

What happens in each of the following cases?

```
Base* b = new Base();
b->print(cout);

Derived* d = new Derived();
d->print(cout);

Base *c = new Derived();
c->print(cout);
```

Note that in this last case, we are using a Base pointer to point at an instance of a Derived class. Why can we do this? Because a Derived object “is-a” Base object, so we can always use a pointer to a base class to refer to some instance of a derived class. However, we cannot use a Derived pointer to point at a Base object.

Virtual Methods

If you define a class with virtual methods, you should always make the destructor virtual.

```
class Base {
public:
    int _x;
    Base() : _x(0) {}
    ...
    ~Base () {}
};

class Derived : public Base {
public:
    char * _string;
    Derived(const char* s) { _string = new char[strlen(s) + 1]; strcpy(_string,s); }
    ~Derived() { delete _string; }
};
```

Question: What happens in the following case?

```
Base* b = new Derived("hello, world*");
delete b; // calls Base::~Base() or Derived::~Derived() ?
```

Since we are deleting an object using a Base pointer, the compiler will think it should call Base::~Base. The destructor should be defined with the virtual keyword, which means that a derived class destructor will be called before the base class destructor when deleting a derived object using a base class pointer.

Abstract Classes and Pure Virtual Functions

It is sometime useful to define a class that is completely “abstract”. In C++, an abstract class is one that has one or more virtual methods that have no associated implementation. Virtual methods with no implementation are called “pure virtual” functions. The implementation for a pure virtual function is provided by a derived class. So, you can never instantiate an abstract base class by itself, you must derive a class from the base class and provide an implementation for the pure virtual methods.

```
class Shape {
protected:
    String name;
public:
    Shape(const char* n) : name(n) {}
    virtual ~Shape();

    // pure virtual functions signified by a NULL implementation using '= 0' syntax
    virtual void draw() = 0;
    virtual double area() const = 0;
};

const double pi = 3.1415927;

class Circle : public Shape {
public:
    Circle(double r) : radius(r) { name = "Circle"; }

    virtual void draw() { /* code to draw a circle */ }
    virtual double area() const { return pi * radius * radius; }
};
```

Abstract Class and Pure Virtual Functions

```
class Rectangle : public Shape {
public:
    double x, y;
    Rectangle(double l, double w) : x(l), y(w) { name = "Rectangle"; }
    virtual double area() const { return x * y; }
    virtual void draw() { /* code to draw a rectangle */ }
};

class Square : public Rectangle {
public:
    Square(double l, double w) : Rectangle(l, w) { name = "Square"; }
};
```

Now, suppose that we want to define a list of Shape objects.

```
Shape* list[3];
list[0] = new Circle(0.0);
list[1] = new Rectangle(5.0,10.0);
list[2] = new Square(1.0,1.0);

for (int i = 0; i < 3; i++) {
    if (list[i].area() > 0)
        list[i].draw();
}
```

Each object in the list is an instance of one of the derived class of Shape. So each derived shape can be referenced using a Shape*. When you call a virtual function using a Shape*, a run-time decision is made about which method to invoke. When you define a virtual function, the compiler inserts a hidden data member into each object, which is a special pointer, called a “vptr”, which points at an internal table of virtual functions. This has the effect of increasing the size of an object by 4 bytes.