

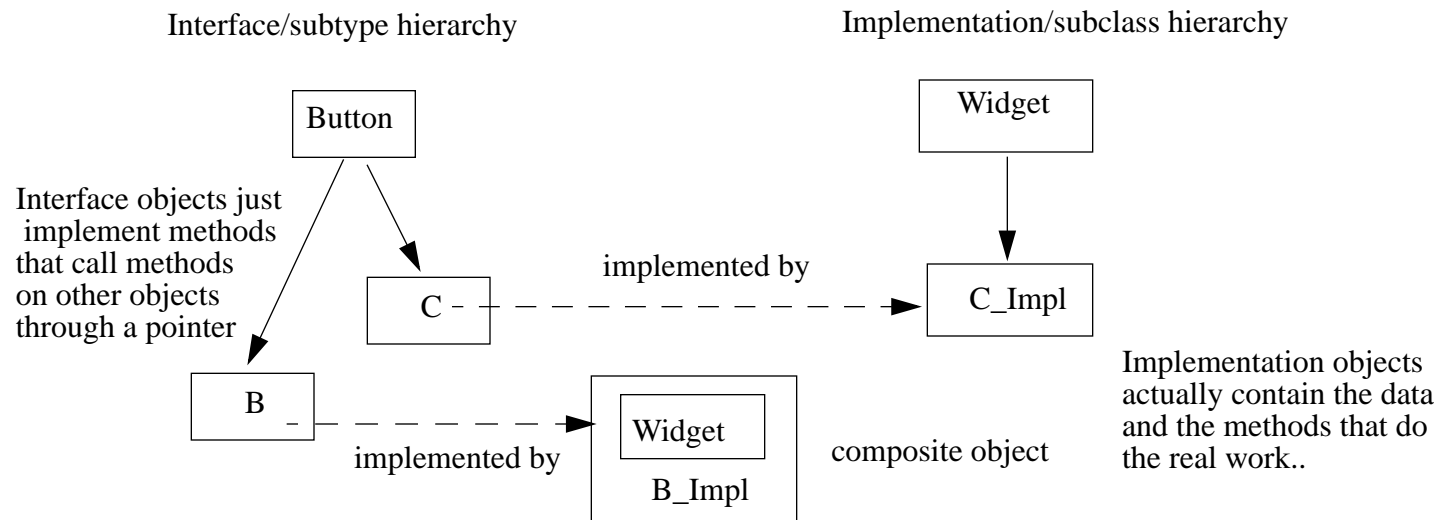
Interface versus Implementation

Inheritance and composition together allow us to cleanly separate an object's interface from its implementation.

1. use inheritance for the purpose of *subtyping* to construct a hierarchy of type related interfaces
2. use inheritance for the purpose of *subclassing* and/or composition to build implementation objects.
3. relate an interface object with an implementation object using a pointer to the object or using composition (has-a) relationship (contains an object as an instance)

This separation is often necessary to facilitate reuse and extensibility. If you cleanly separate the parts, it is easier to recombine them in new ways.

To achieve this separation, OO programmers commonly build class interface hierarchies and related class implementation hierarchies. The user of your classes never needs to know about the details of the implementation class, just the interface classes. In other words, interface classes very often just call methods on other objects.



Interface Inheritance or Subtyping

A common technique is to define an interface as an *abstract base class*, with public **pure virtual** methods and no instance variables. The methods simply define an abstract interface consisting of method declarations to which all derived classes must conform. The goal is to separate the interface definition from the implementation details.

```
class AbstractBase {
/* no private data */
public:
    virtual void a(...) = 0; // denotes a pure virtual function having NO IMPLEMENTATION
    virtual int b(...) = 0;
    virtual void c(...) = 0;
    ...
};
```

Once the interface is defined, publicly derive a *concrete* subclass that “conforms” to the type signatures of the methods in the abstract interface (i.e., has the same method declarations), but contains concrete implementation details. We say the concrete subclass inherits (is derived) from the abstract base class. In the case of interface inheritance, we also say that the concrete derived class is a subtype or type specialization of the base class. A subclass is only a subtype of the base class if it provides exactly the same interface methods. The subclass may add methods, but it has to provide ALL of the base class methods in its interface.

```
class ConcreteSubclass : public AbstractBase {
    // some private implementation data
public:
    virtual void a(...) { ... } // some methods implemented inline
    virtual int b(...); // some methods implemented in a separate file
    virtual void c(...);
};
```

You can also go a step further in separating interface from implementation, by having the concrete subclass contain in its private section a pointer to another object that contains the actual implementation details.

Inheritance of an Abstract Interface

Consider the following example of an abstract base class `Shape` that defines a common abstract interface for different kinds of shapes, and each derived class implements the interface. That is, the computation of the area and the drawing of a shape are abstract operations that only have meaning (i.e., an implementation) when specialized using inheritance and a concrete implementation is provided for these methods.

```
class Shape {
public:
    ...
    virtual void draw() = 0; // "pure" virtual member functions have no implementation
    virtual double area() = 0;
};

class Rectangle : public Shape {
    Point top_left, bottom_right; // concrete data for a specific kind of shape
public:
    ...
    virtual void draw(); // subclass must (eventually) implement pure virtual methods
    virtual double area();
};

class Circle : public Shape {
    Point center;
    double radius;
public:
    ...
    void draw(); // although not explicitly declared virtual by default
                // "virtual-ness" is inherited since Shape::draw is virtual
    double area(); // ditto
};
```

Using an Abstract Base Class

An abstract base class is most commonly used as an abstract type name of a corresponding pointer or reference type that is used to refer to concrete subtypes in a generic way:

```
void display(Shape& s)
{
    s.draw(); // which virtual draw() method is called?
    cout << "The area is: << s.area(); // which area() method is called?
}

Rectangle r; // Rectangle is-a Shape
Circle c;    // Circle is-a Shape
...
display(r);
display(c);
```

Similarly, you can use a Shape* to refer to a subtype object, since a Rectangle is-a Shape and a Circle is-a Shape:

```
Shape *shape1 = new Rectangle(x,y);
Shape *shape2 = new Circle(p,r);

shape1->draw(); // draws a rectangle
shape2->draw(); // draws a circle
```

Question: What happens as a result of each of the following assignment statements?

```
shape1 = shape2; // assign a Shape pointer to a Circle to a pointer to a Rectangle
shape1->draw();

*shape1 = *shape2; // incorrect run-time assignment of a Circle to a Rectangle!!
shape1->draw(); // Can the compiler know that shape2 points at a circle?
```

Order of Construction under Inheritance

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. Order of construction is top down. What about order of destruction?

Note: that destructors can be virtual but constructors cannot. What is a virtual destructor?

```
class Base {
    int _x;
public:
    Base() : _x(0) {}
    Base(int x) : _x(x) {}
    virtual ~Base(); // destructor should be virtual in a class with other virtual methods
    virtual f(); /* default implementation */
    virtual g();
};

class Derived : public Base {
    int _y;
public:
    Derived() : Base(), _y(0) { ... }
    Derived(int x, int y) : Base(x), _y(0) {...}
    virtual ~Derived() { ... }
    virtual f(); /* overridden implementation for Base::f() */
};
```

NOTE: a class that has been named, but not yet declared cannot be used as a base class. Why??

```
class Vehicle; // forward declaration
class Truck : public Vehicle {
    ...
};
```

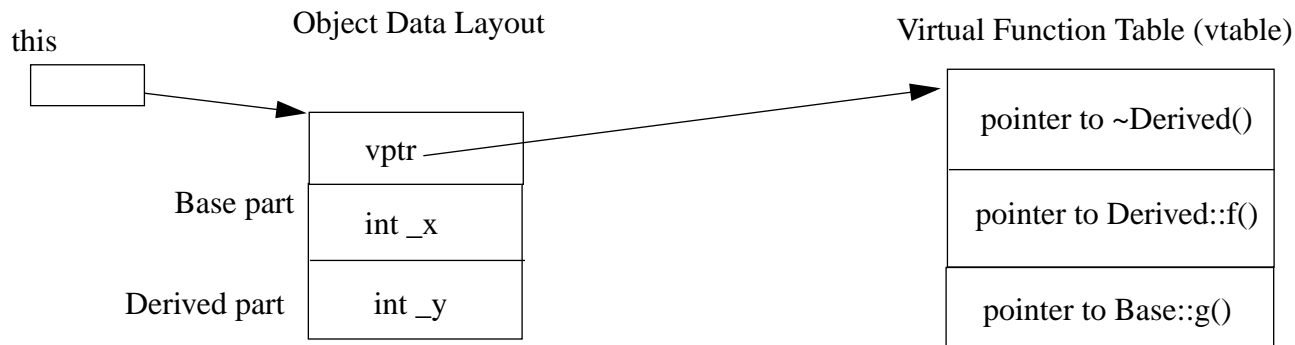
Composite Object Layout

Inheritance results in a “composite” object structure at run-time. The composite object data area is a concatenation of the data areas of the component objects in a COMPILER DEPENDENT manner. The sizeof an object is the sum of the sizeof it components and the components of its base class(es). If virtual methods are used, additional “bookkeeping” information is also stored in the data portion of the object. This is usually in the form of a “vptr” or virtual function table pointer.

Public methods are combined into one set of methods, but with distinct names due to “name mangling.” Data layout is combined into a single contiguous chunk of memory, but compile-time access control (i.e., private) still applies

IMPORTANT NOTE: because a base class object is constructed before a derived class object, you should never call a virtual function from a constructor. The simple explanation is that since a derived object is not yet fully constructed, if you call a virtual function in a base class constructor, it may not yet have completely initialized the layout for the object, so you are likely to encounter a segmentation violation as result of dereferencing a pointer that has not yet been set to a valid address.

More precisely, the reason is that each object containing virtual functions has a special “virtual function table” (vtable) pointer associated with the object. This pointer is typically called the “vptr” and it is a hidden data member in each object whose class defines one or more virtual functions (so it affects the sizeof an object). Virtual function calls are made by using the ‘this’ pointer to locate the vptr, and then using the vptr plus an offset into the vtable to locate a function pointer in the vtable that is used to make the correct virtual function call for an object at runtime.



Calling Base Class Methods from a Derived Class

- You can inherit both virtual and non-virtual methods as they are defined and call them as a normal member methods.
- You can override an inherited method with a new implementation.
- You can extend a method of the same name that calls the inherited method at some point.

```
class Base {
    int _x;
public:
    ...
    virtual void print(ostream& s) { s << "Base class"; }
};

class Derived : public Base {
    int _y;
public:
    ...
    virtual void print(ostream& s) { Base::print(s); s << "...Derived class "; }
};
```

What happens in each of the following print statements?

```
Base *b = new Base();
Derived *d = new Derived();
b->print(cout);
d->print(cout);
delete b;
b = d;
b->print(cout); // at run-time, the vptr of the object pointed to by b is used
                // to dynamically lookup which print function to call. Since really
                // b points at an instance of Derived, the vtable of a Derived object
                // is used, and hence the Derived::print method is called, not Base::print
```

Using Base Class Methods

There was once a proposal in C++ to add a new reserved word called **inherited** that would be used in such situations. The idea was to permit you to write:

```
void print(ostream& s) { inherited::print(s); s << "Derived class"; }
```

The reason was that if the base class name changes, then one has to change all occurrences of `Base::method` in the code, so the indirect name for a base class is more flexible. The idea of introducing the `inherited` keyword into C++ was rejected when Michael Tiemann (the implementor of g++) noticed that you could achieve this very easily using a *nested typedef*. So, if you ever change the Base class name, you just change the typedef and everything is fine. Note that this is a similar technique to the way nested typedefs are used with templates.

```
class Derived : public Base {  
    typedef Base inherited;  
    ...  
public:  
    ...  
    virtual void print(ostream& s) { inherited::print(s); s << "Derived class"; }  
};
```

Question: if the `Base::print` method is written as follows, what happens when the call to `print` is made as shown below?

```
class Base {  
    int _x;  
public:  
    ...  
    virtual void print(ostream& s) { print(s); }  
};
```

```
Base* b = new Derived();  
b->print(); // which print method is called??
```

A Simple Rule for When to Use Public Inheritance

Use public inheritance to model “is-a” relationships.

That is to say, a derived class D should inherit public from a base class B only if you can use an object of type D where an object of type B is expected. Note that this does NOT mean that you can use an object of type B in place of a D. Here is a simple example:

```
class Person {
public:
    enum Gender { Male, Female };
    Person(String name, int age, Gender gender)
        : _name(name), _age(age), _gender(gender) {}
    String name() const { return _name; }
    int age() const { return _age; }
    Gender gender() const { return _gender; }
protected:
    String _name;
    int _age;
    Gender _gender;
};

class Student : public Person {
public:
    enum Rank { Freshman, Sophomore, Junior, Senior, Graduate };
    Student(String name, int age, Gender gender, String ssn)
        : Person(name, age, gender), _ssn(ssn) {}
    ...
    Rank rank() const { return _rank; }
    String ssn() const { return _ssn; }
private:
    String _ssn;
    Rank _rank;
};
```

Simple Test for Determining if Public Inheritance is Appropriate

Imagine a collection of objects of the base type B. e.g., `list`. Is it reasonable to put objects of derived type D in the collection and treat them as though they were objects of type B. For example: Assuming that class `Faculty` also inherits from `Person`. Does the following code make sense? That is, can you put both `Student` and `Faculty` objects in a collection like a list, and then treat both types of objects as `Person` objects?

```
list<Person> people;
people.push_front(Student("Mary Smith", 22, Person::Female, "900-99-1010"));
people.push_front(Faculty("Ed Jones", 40, Person::Male));

list<Person>::const_iterator i;
for (i = people.begin(); i != people.end(); ++i) {
    cout << "name: " << (*i).name() << endl;
    cout << "age: " << (*i).age() << endl;
    cout << "gender: " << (*i).gender() << endl;
}
```

Note however that the following code does not work. Why?

```
cout << "level: " << (*i).rank() << endl;
```

Similarly, ask yourself whether or not a function that operates on an object of type B can just as well operate on a object of type D. E.g., Assume that the output operator is defined for a person object, but there is not one defined taking a `Student` as an argument.

```
friend ostream& operator<<(ostream&, const Person&);
Student student(...);
cout << student;
```

The above code calls `operator<<(ostream&, const Person&)` because the `Student` type is a subtype of `Person`, and if there is no overloaded `operator<<` taking a `Student` argument, the compiler will use the `operator<<` method taking a `Person`.

Mis-Using Inheritance in C++

There are three ways in which inheritance is used in C++.

1. Type or interface inheritance - inheriting for the purpose of *subtyping* as we've just seen
2. Implementation inheritance - inheriting for the purpose of *code reuse*.
3. Confused inheritance - inheriting from a class just because you can and it seems a neat thing to do in C++

NOTE: The fact that the same general-purpose class inheritance mechanism is used to implement both subtyping and subclassing is one of the major criticisms of C++. Other object-oriented languages (e.g., Java) define interface inheritance for the purpose of subtyping as distinct from class inheritance for the purpose of code reuse.

Since the public interface of list is inherited, it is possible to violate the LIFO stack discipline by calling public list methods on a stack object that inherits from list. What do you think about the following example?

```
template<class T> class list { // ANSI C++ list template
    ...
public:
    ...
    T back();
    void push_back(const T& x);
    void pop_back();
    void push_front(const T& x);
    void pop_front();
};

template<class T> class stack : public list<T> { // stack<T> is-a list<T> ??
public:
    ...
    void push(T x) { ...; list<T>::push_back(x); }
    T pop() { T x = list<T>::back(); list<T>::pop_back(); return x; }
};
```

Mis-Using Inheritance in C++

Does inheriting private to hide the public list<T> methods help? When you inherit private from a class, the public methods of the base class are not accessible to a user of the derived class. In effect, the inherited public methods are made private in the derived class.

```
template<class T> class stack : private list<T> {
    ...
public:
    void push(T x) { list<T>::push_back(x); }
    T pop() { T x = list<T>::back(); list<T>::pop_back(); return x; }
    ...
};
```

In C++, you can use a pointer to a publicly inherited base class to point at an instance of a derived class. So, ask yourself the following question: Does using a list<T> pointer to a stack<T> object make sense? **That is, can you use a stack in place of a list?** If not, then a stack is not a true subtype of a list. In C++, you can legitimately write the following since stack<T> inherits from list<T>:

```
list<int>* slist = new stack<int>();
slist->push_front(5); // insert at the "bottom" of the stack instead of the "top"
```

It is then possible to call the list<int>::push_front method since it is in the public interface of a list<T>, and the compiler run-time doesn't restrict this. Thus it is better to have the list<T> object be encapsulated as an object used by the stack object to prevent this type of overt bypassing of the interface of a stack object by casting a stack to a list, and then using the stack as though it were a list. A stack abstraction should strictly enforce a stack interface.

A stack is an adapter that is a restriction on the more general list interface. Inheritance is not generally used to implement a restricted interface, but to implement or expand upon an inherited interface. So, although it may seem attractive to have a stack inherit a list for the purpose of inheriting its implementation (code reuse), a better approach is to encapsulate the list as part of the internal implementation of a stack rather than use inheritance, since a stack is not a subtype of list.