IB9N7 C++ for Quantitative Finance

Lecture 12: Inheritance

J.F.Reizenstein@warwick.ac.uk

4 February 2016
(Week 18)

Outline

As you know, objects are useful for encapsulating functions and data, and providing convenient abstractions such as operator overloading.

But: this is arguably little more than just syntactic sugar.

In this lecture and next, we'll cover some even more powerful features of object-oriented programming:

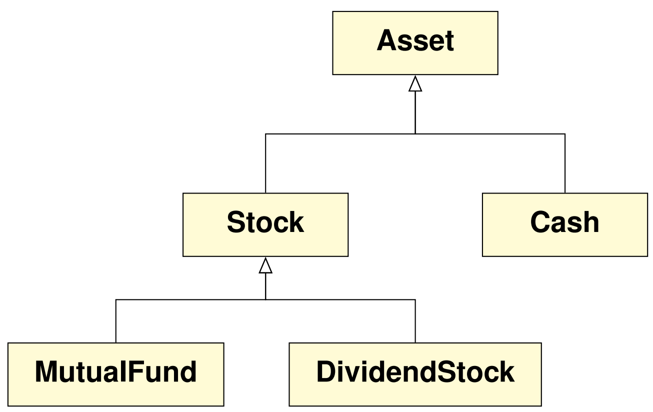
- Inheritance
- Dynamic polymorphism

Inheritance

Objects often belong in hierarchies:

- `MutualFund`s and `DividendStock`s are both types of `Stock`.
  - `Cash` and `Stock` are both types of `Asset`.

Visualisation by inheritance diagrams



OOP inheritance: motivation

We can exploit such hierarchies in the design and structure of our code to make our code more readable and more easily reusable etc.

- Data and methods that apply to **`Asset`**s also apply to **`Cash`** and **`Stock`**, etc.
  - **`Stock`**s have additional specialised properties.

Every

- **`Asset`** has a market value,
  - **`Stock`** also has a symbol,
    - **`DividendStock`** also has dividends,
    - etc.

Classes need only define the additional entities that make them unique in the hierarchy, and borrow the others.

OOP inheritance

Inheritance

Object oriented languages allow such hierarchies to be specified through the concept of **inheritance**.

Through inheritance, one specifies that a class is to be a specialisation of another class.

- A specialised class is called a **derived class** (or sub-class).
- A class from which it inherits is called a **base class** (or super-class).

The base class defines members that are common to the hierarchy.

- A derived class **extends** the functionality of a base class.

Base classes

A base class is defined just like any other class.

- There are no special requirements.

Example: a class that will become a base class

```
// Base.hpp
class Base
{
   public :
      void say_hello () const;
};
// Base.cpp
#include <iostream>
void Base::say_hello () const
{
   std::cout << "Hello from Base::say_hello" << std::endl
      ;
}
```


Derived classes

A derived class is specified by adding to its definition:

- a colon (following the identifier of the derived class),
- then an access specifier (**`public`**, **`private`** or **`protected`**),
- then the identifier of the class that is to be the base class.

Example: a class that extends the above class

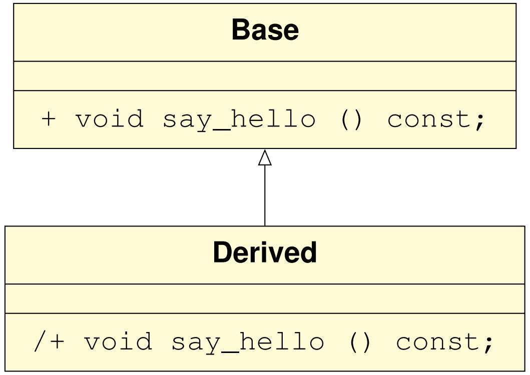
```
// Derived.hpp
#include "Base.hpp"

class Derived : public Base
{
   // ... optional extra members ...
};
```

Inheritance of class members

- All members of the base class which are `public` or `protected` become members of the derived class (possibly with different access levels, described later). A `protected` member can (only) be accessed by the class itself or one derived from it.
- These members are called **inherited members**.
- Additional members can be added to the derived class to extend the functionality of the base class.
- Base class must be declared before derived classes.

Inheritance diagram of example



Example: public members are inherited

Example: Inheritance

With the previous definitions of `Base` and `Derived`:

```
#include <cstdlib>
#include "Derived.hpp"

int main()
{
    Derived d;
    d.say_hello(); // Outputs "Hello from Base::say_hello"
    return EXIT_SUCCESS;
}
```

Note that this is valid because the function `say_hello` was `public` in `Base`, and so inherited by `Derived`.

Inheritance access specifiers

Examples: public/protected/private

The following are possible inheritance access specifications:

```
class Derived : public Base { /* ... */ };

class Derived : protected Base { /* ... */ };

class Derived : private Base { /* ... */ };
```

Members of the base class which were `public`:

- remain publicly accessible in the derived class if `public` inheritance is used.
- become `private`/`protected` in the derived class, when the respective inheritance specifier is used.

Derived can use protected members of parent

Example

```
class Base
{
   protected :
      void say_hello () const;
};
class Derived : public Base
{
   public :
      void call () const;
};
void Derived::call() const
{ say_hello(); } // OK, calls Base::say_hello
```


Protected members are private externally

A **`protected`** member is still inaccessible from outside the base or derived class.

Example

```
int main()
{
   Base b;
   b.say_hello(); // ERROR! say_hello is not accessible

   Derived d;
   d.say_hello(); // ERROR! say_hello is not accessible
   d.call(); // OK!

   return EXIT_SUCCESS;
}
```


Access levels of inherited members

	Base class member		
	`private`	**`protected`**	**`public`**
Inheritance:			
`private`	Inaccessible	**`private`**	**`private`**
`protected`	Inaccessible	**`protected`**	**`protected`**
`public`	Inaccessible	**`protected`**	**`public`**

Usually, **`public`** is the sensible choice, but sometimes **`private`** is more appropriate if the derived class needs to manage access to the base class members.

Indirect inheritance

- A derived class can itself be a base class of another class.

  Protected access is transitive.
- There are classes that cannot be derived from (e.g. those with no non-**`private`** constructors), or classes declared with **`final`** after the class name.

  (There are uses for classes with only **`private`** constructors, e.g. in the singleton pattern.)

Indirect inheritance: example

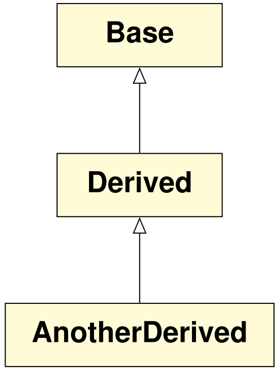
Example: successive inheritance

```
class Base {};
class Derived : public Base {};
class AnotherDerived : public Derived {};
```

Both **Base** and **Derived** are base classes of **AnotherDerived**.

- **Derived** is its **direct** base.
- **Base** is an **indirect** base.

Indirect inheritance: example diagram



Name lookup

When attempting to access an unqualified member of derived class,

- The compiler first looks up the class of the static type being called upon,
- then its direct base,
- then the direct base of the direct base,
- etc.

(Static types are discussed next lecture. For now, the static type will be the most derived class.)

One can also refer directly to the member using its full identifier,

eg if `d` is of type `Derived`,
`d.Base::call` refers to `call` from `Base` (but in the above example, `Base::call` is `protected`),
`d.Derived::call` refers to `call` from `Derived`.

Wrapping/modifying existing classes

Inheritance is useful for extending and replacing functionality of existing classes.

Overview of an example:

- Suppose there is already a class available that generates random numbers.
- This class might be designed such that each object instance has its own seed and generates an independent sequence from any other object instance.
- You might inherit from this class privately in order to either restrict changing of the seed, or to print out a message every time the seed is changed.

Overriding base class members

A derived class can **override** a base class member by defining a member with the same identifier.

- If using `public` inheritance, one should normally declare the function in the base class as `virtual`.

Note: overriding a member **hides** (but does not prevent access to) any members with the same identifier in any base class.

- The hidden functions can be unhidden by adding `using` declarations.
- If the access specifier permits, the hidden functions are still accessible if accessed through a pointer/reference to the base class, or using the full identifier.

Overriding base class members (2)

Example

This example illustrates what happens if one (but not all) overloads of a function are overridden.
Suppose we start with this base class:

```
class Base
{
   public :
      void say_hello () const
      { std::cout << "Base::say_hello" << std::endl; }

      void say_hello (int x) const
      { std::cout << "Base::say_hello␣" << x << std::endl; }
};
```


Overriding base class members (3)

Example (continued)

```
class Derived : public Base
{
   public :
      // using Base::say_hello; // uncomment this to unhide base argument version
      void say_hello () const
      { std::cout << "Derived::say_hello" << std::endl; }
};
```


Overriding base class members (4)

Example (continued)

```
int main() {
   Derived d;

   d.say_hello(); // OK, outputs "Derived::say_hello"
   //d.say_hello(2); // ERROR! (unless uncomment the using declaration)
   d.Base::say_hello(2); // OK, use full identifier

   return EXIT_SUCCESS;
}
```

In the second (unqualified) call to `say_hello`, the compiler will only find `Derived::say_hello`, which does not accept any arguments.

Promotion of access level (1)

A `protected` member can be promoted to `public` by a derived class, either through a `using` declaration or by overloading the member and passing the call on.

Example

In the following class, both `f` and `g` are `protected`.

```
class Base
{
   protected:
      void f () {}
      void g () {}
};
```


Promotion of access level (2)

The following example shows the two methods of promoting the access level from a derived class:

Example (continued)

```
class Derived : public Base
{
   public :
      using Base::f; // using declaration
      void g (); // override
};
void Derived::g ()
{
   Base::g(); // pass the call on
}
```


Promotion of access level (3)

Example (continued)

The following illustrates the effect of the access level promotion:

```
// in main()
Base b;
b.f(); // ERROR! f is private to us
b.g(); // ERROR! g is private to us

Derived d;
d.f(); // OK!
d.g(); // OK!
```


Sub-objects

Each derived class object has an instance of the base class as a sub-object.

Similar to having an instance of the base class as a member variable, but with special

- access rules;
- construction, initialisation and destruction behaviour.

There are certain properties of classes which are therefore inherited:

for example, if a base class does not have a copy constuctor or assignment operator, then the derived class will not have these either.

Sub-object construction rules

- The base class sub-object is initialised before other members of the derived class.
- The base class sub-object can *only* be initialised through an initialiser list, accessed by the name of the base class followed by arguments to one of its constructors.
- If the derived class does not initialise the base class sub-object explicitly, then the default constructor of the base class is used.
- If the base class has constructors but does not have a default constructor, then the derived class has to initialise the base class explicitly.
- The base class sub-object is destroyed after other members of the derived class. (Reverse order to initialisation.)

Slicing problem (Warning!)

If attempt assignment, passing or other casting from a derived class to base class type (non-pointer/reference case):

Base class doesn't know anything about the members that may be additional to the derived class;
It certainly can't store them;
The object is therefore said to be "sliced", with all the derived class members ignored.
Can be especially dangerous when the base class type is only the static type.

The slicing problem is serious because it can result in memory corruption, and it is very difficult to guarantee a program does not suffer from it.

Usually pass objects around by reference to try to avoid this kind of problem.

pointer conversion

You can convert a pointer or reference to the derived class to one to the base class anywhere you can see the base class. (If inheritance is public, you can do this anywhere).

```
class Base{};
class Derived : public Base {};
int main(){
    Derived d;
    Base b1 = d; //Bad – this is a slicing
    Base& b2 = d; //fine
    Base* b3 = &d; //fine
}
```


Public inheritance and the "is-a" relation

`public` inheritance is so far the most used type of inheritance, and expresses the "is-a" relation.

- Make sure all the `public` members of a base class are meaningful for a derived class. Because,
- A derived class *IS A* base class.
- But consider the distinction between the two types of composition:

  is a (aka. "kind of"), realised through inheritance and
  has a (aka. "part of"/aggregation/containment), realised through members.

Non-`public` inheritance expresses the "has-a" relation, just like containment.

Construct your hierarchy carefully

Do not misuse inheritance

Think of your classes in terms of the relevant abstractions, not in terms of real world objects.
(Perhaps, consider it to be "behaves-as-a" rather than "is-a".)

Sometimes the violation of the *is-a* relation is not obvious. . .

Violation of the "is-a" relation

Example: A penguin cannot fly!

```
class Bird
{
   public :
      void fly () { /* let the bird fly */ };
};

// A Penguin is a Bird, in the real-world sense!
class Penguin : public Bird {};

Penguin p;
p.fly(); // Em?!
```


A better abstraction

Example: Not all birds are the same!

Fix:

```
class Bird {};

class FlyingBird : public Bird
{
   public :
      void fly ();
};
class NonFlyingBird : public Bird {};

class Hawk : public FlyingBird {};
class Penguin : public NonFlyingBird {};
```


Circle-ellipse problem (isomorphism of)

While a square is mathematically a rectangle. . .

- A `Rectangle` class would probably have `set_width` and `set_height` members.
- The two functions would be independent for a `Rectangle`, . . .
- . . . but not for a `Square`.
- From an object-oriented perspective, all public members of the parent class `Rectangle` would be provided and somehow accessible in the sub-object of a `Square` class. . .
- If the `Rectangle` class had a method `set_size(width, height)`, there would be no good solution to maintain the `height` = `width` invariant.

Avoid inheritance where the derived class has different semantics to the base class.

Multiple inheritance

A class can directly inherit from more than one base class.

```
class A{};
class B{};
class C : public A, public B {};
```

Some people are afraid of this, but used sensibly it can be useful. One must be careful if a class inherits another class in more than one way. For example

```
class A{};
class B : public A {};
class C : public A {};
class D : public A, public B {}; //whoops
class E : public C, public B {}; //whoops
```

This is called the diamond problem, and we will not discuss it further.

Dynamic polymorphism

Dynamic polymorphism

Polymorphism: having more than one form.

Dynamic polymorphism

The same member function call can have different behaviour under different contexts.

(Function and operator overloading and templates are sometimes also referred to as forms of polymorphism.)

- Provide a common **interface** through a base class.
- Provide specialised **implementation** through a derived class.

For us, this is the most important application of inheritance.

“One interface, multiple methods”

Static and dynamic types

A pointer or reference that is declared to point/refer to an object of base class type may also point/refer to objects of derived class type(s).

- A reference or pointer has both a **static** type and, when it refers to something valid, also a **dynamic** type.
- The declaration of the reference or pointer determines the **static** type.
- The object referred or pointed to by the reference or pointer determines the **dynamic** type.
- They may be the same.

Object polymorphism through pointers/references

Suppose you have a base class **Base** with derived class **Derived**.

Suppose **f** has prototype

```
void f (const Base & b);
```

Then, can write

```
Base b;
Derived d;

f(b);
f(d);

Base * p1 = &b; // p1 -> static type Base, dynamic type Base
Base * p2 = &d; // p2 -> static type Base, dynamic type Derived
```


Virtual functions

A member function with a **virtual** prefix in the class definition is called a *virtual function*.

- Normally, when an overridden function is invoked through a base class pointer/reference, the base class implementation gets executed.
- When an overridden **virtual** function is invoked through a base class pointer/reference, the (most) derived class implementation (with a compatible signature and return type) gets executed.
- If **override** is added at the end of the declaration of the derived function, then the compiler will check that it does override something. This helps avoid typos.

Virtual functions, example part 1

Example: Base type with virtual function

```
class NIntegrate // for numerical integration
{
   public :
      double integrate (const std::vector<double> & grid) const
      {
         double integral = 0.0;
         for (std::vector<double>::size_type i = 1u; i != grid.size(); ++i)
            integral += segment(grid.at(i - 1u), grid.at(i));
         return integral;
      }
   private :
      // Default: use Trapezoid rule.
      // private here (only for illustrating slide 48)
      virtual double segment (double a, double b) const
      { return 0.5 * (b - a) * (f(a) + f(b)); }
};
```


Virtual functions, example part 2

Example: Overridden virtual function

Derived class re-implements the base class virtual member function.

```
class NIntegrateSimpson : public NIntegrate // Simpson's Rule
{
   public :
      double segment (double a, double b) const override // public here (again for illustrating slide 48)
      {
         return (b - a) * (f(a) + 4.0 * f(0.5 * (a + b)) + f(b)) / 6.0;
      }
};
```

Virtual functions, example part 3

Example: Virtual dispatch

```
void integrate_and_print(const NIntegrate & quadrature)
{
   std::vector<double> grid(2u);
   grid.at(0u) = 0.0; grid.at(1u) = 1.0;

   std::cout << quadrature.integrate(grid) << std::endl;
}
```

Example: `f`

```
double f (double x) { return x * x; } // better would be to use a function pointer or class (lab exercise!)
```


Virtual functions, example part 4

Example: Bringing it together

```
int main()
{
   NIntegrate trapezoidObj;
   NIntegrateSimpson simpsonObj;

   integrate_and_print(trapezoidObj); // 0.5
   integrate_and_print(simpsonObj); // 0.333333
   // (if 'segment' wasn't virtual in 'NIntegrate', both would output 0.5)

   return EXIT_SUCCESS;
}
```

Hence, we see another advantage of taking const references as formal arguments: it allows passing of subclasses.

Virtual functions: summary

- A derived class reference or pointer can be implicitly converted to a base class reference or pointer.
- The conversion is always valid since a public derived class object *is a* base class object.
- All member function calls inside other member functions are implicitly called through the pointer `this`.
- Can prevent `virtual` function lookup by using the full method identifier (invoking a `virtual` function non-`virtually`).
- Virtual function dynamic binding *ONLY* happens when a *`virtual`* function is called through a reference or pointer.

Only static types relevant at compile time

- The compiler looks for member functions by examining the *static* type.
- What the actual type of object the pointer will point to is irrelevant at compile time.

Example

```
NIntegrate * simpson = &simpsonObj;
// 'simpson' has static type 'NIntegrate'
simpson->segment(0.0, 1.0); // ERROR! NIntegrate::segment is private
```

Instead, if we do (on the same underlying object)

```
NIntegrateSimpson * simpson = &simpsonObj;
// 'simpson' has static type 'NIntegrateSimpson'
simpson->segment(0.0, 1.0); // OK! NIntegrateSimpson::segment is public
```

Dynamic binding

Let's look again at what happens with this snippet of code:

Example

```
// within main():
NIntegrateSimpson simpsonObj;
integrate_and_print(simpsonObj);
//... and within integrate_and_print:
quadrature.integrate(grid);
//... and within integrate:
this->segment(grid.at(i - 1u), grid.at(i));
```

- `NIntegrate::integrate` calls `this->segment` (`this` is implicit).
- `this` is of static type `NIntegrate`.
- `NIntegrate::segment` is `virtual`; dynamic binding is used.
- The pointer actually points to an `NIntegrateSimpson` object.
- `NIntegrateSimpson::segment` is executed.

Pure virtual functions and abstract classes

A `virtual` function declared by the following syntax is called a **pure virtual** function, and need not have an implementation:

```
virtual double segment (double a, double b) const =
    0;
```

(Then we wouldn't need to provide a 'default' quadrature scheme.)

Abstract classes

A class with one or more pure virtual functions is called an **abstract class**.

An abstract class consisting of only pure virtual functions is called an **interface class**.

Abstract and concrete classes

Concrete classes

A class which implements all the pure `virtual` functions from any parent (or has no pure virtual functions) is called a **concrete class**.

- Cannot create objects from an abstract class blueprint.
- Can only create objects from a concrete class blueprint.

Example

```
class AbstractBase { virtual void f () = 0; };
class ConcreteDerived : public AbstractBase
    { virtual void f () {} };

AbstractBase b; // ERROR!
ConcreteDerived d; // OK!
```


Virtual destructors

- When a pointer is `delete`d, the destructor is used through the pointer. It obeys the same binding rule as other member functions.
- A base class should always declare its destructor as `virtual` if you want to be able to delete any derived class via the base class pointer.
- Note that, by declaring the destructor as `virtual` in the class definition, the destructor is declared! Therefore, you also have to implement it, rather than relying on the implicitly generated one. This is usually the empty function body.

```
class Base{
public:
    virtual ~Base(){}
};
class Derived : public Base {};
int main(){
    std::unique_ptr<Base> b = make_unique<Derived>();
}
```

Dynamic memory

Usually use dynamic polymorphism:

- with dynamic memory ;
- with smart pointer class ;
- with Factory pattern .

Lab objectives

Today, you are expected to:

- Work through some examples of inheritance.