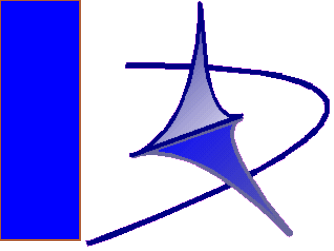
Scuola Superiore Sant'Anna

Advanced Course on C++

III

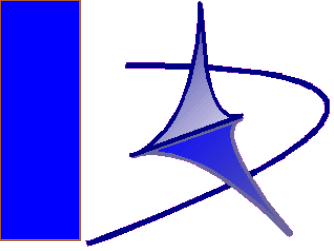
Giuseppe Lipari



Inheritance

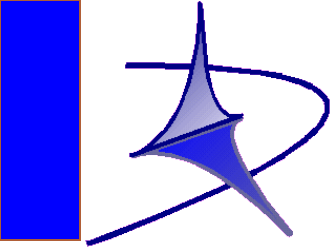
Are you ready for object oriented programming?

So far, we were kidding...



Code re-use

- In C++ (like in all OO programming), one of the goals is to re-use existing code
- There are two ways of accomplishing this goal: *composition* and *inheritance*
  - Composition consists defining the object to reuse inside the new object
  - Composition can also expressed by relating different objects with pointers each other
  - Inheritance consists in *enhancing* an existing class with new more specific code
  - Until now you've seen only composition



Inheritance syntax

- The syntax is the following:

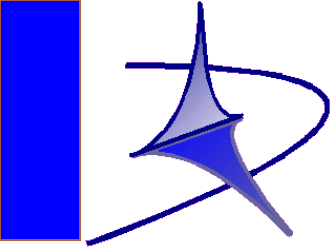
```
class A {
   int i;
protected:
   int j;
public:
   A() : i(0),j(0) {};
   ~A() {};
   int get() const {return i;}
   int f() const {return j;}
};
```

```
class B : public A {
   int i;
public:
   B() : A(), i(0) {};
   ~B() {};
   void set(int a) {j = a; i+= j}
   int g() const {return i;}
};
```

B is derived from A

Constructor

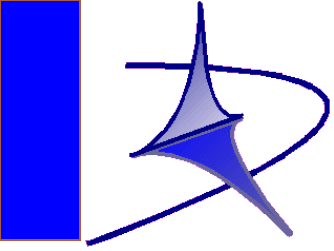
inherited member



Inheritance

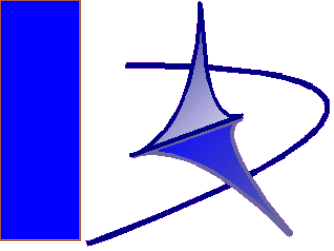
- Now we can use B as a *special version* of A

```
int main()
{
    B b;

    cout << b.get() << "\n";        // calls A::get();
    b.set(10);
    cout << b.g() << "\n"

    b.g();

    A *a = &b;                  // Automatic type conversion
    a->f();
   B *p = new A;
}
```



Constructor call order

- see ord-constr/
- Watch out for the order in which things are done inside a constructor…
- Of course, destructors are called in reverse order...



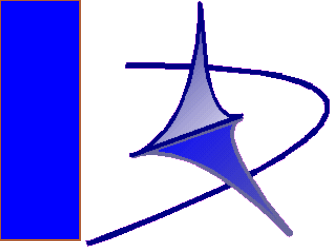
Redefinition and name hiding

- Of course, we can re-define some function member

```
class A {
   int i;
protected:
   int j;
public:
   A() : i(0),j(0) {};
   ~A() {};
   int get() const {return i;}
   int f() const {return j;}
};
```

```
class B : public A {
   int i;
public:
   B() : A(), i(0) {};
   ~B() {};
   int get() const {return i;}
   void set(int a) {j = a; i+= j}
   int f() const {return i;}
};
```

```
int main()
{
    B b;

    cout << b.get() << "\n";
    b.set(10);
    cout << b.f() << "\n"
}
```



Overloading and hiding

- There is no overloading across classes

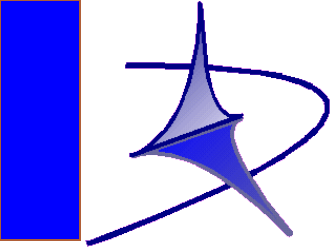
```
class A{
   ...
public:
   int f(int, double);
}
```

```
int main()
{
   B b;
   b.f(2,3.0);   // ERROR!
}
```

A::f() has been hidden by B::f()

```
class B : public A{
   ...
public:
   void f(double);
}
```

either you redefine exactly the base version;
or you will hide all the base members with the same name



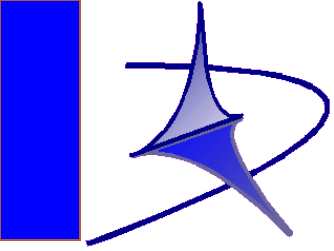
Scoping

- Suppose that B *refines* function f()
  - B::f() wants to invoke A::f()

```
class A {
public:
    int f(int i);
};
```

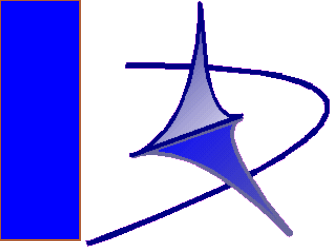
```
class B : public A {
public:
    int f(int i) { return A::f(i) + 1;}
};
```

We must use the scope resolution



Not everything is inherited

- Constructors and similar are not inherited
  - constructors
  - assignment operator
  - destructor
- Default constructor, copy constructor and assignment are automatically synthesized, if the programmer does not provide its own
  - when writing these functions, remember to call corresponding function in the base class!



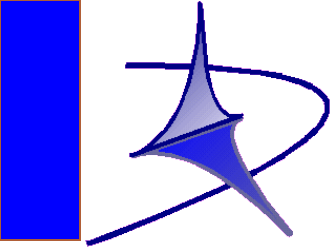
Example

```
class A {
    int i;
public:
    A(int ii) : i(ii) {};
    A(const A&a) : i(a.i) {}
    A &operator=(const A&a) {i = a.i;}
};
```

```
class B : public A {
    int j;
public:
    B(int ii) : A(ii), j(ii+1) {};
    B(const B& b) : A(b), j(b.j) {}
    B &operator=(const B& b) {
        A::operator=(b); j = b.j;
    }
};
```

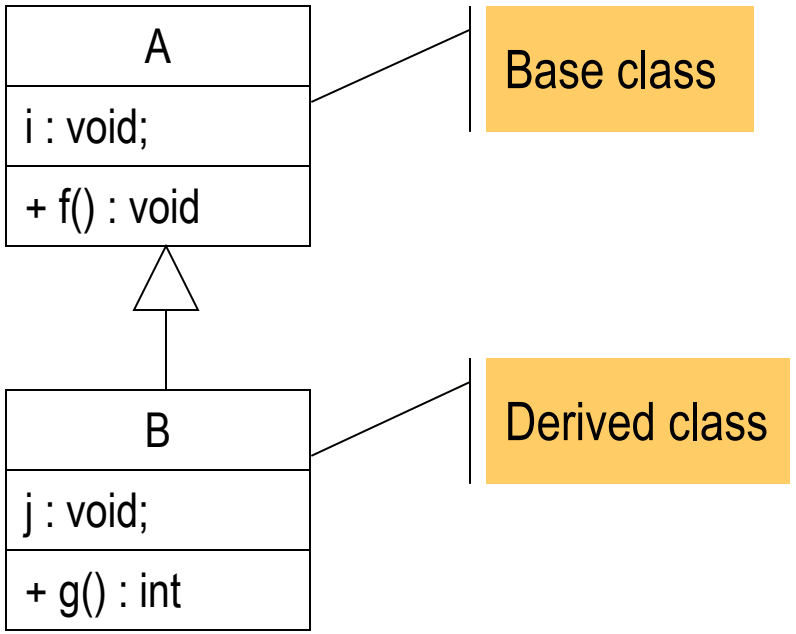
- Wherever you can use A, you can use B…
  - an object of class B *isA* subtype of A
  - This is called up-casting

Notice how this works...

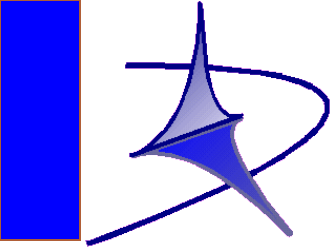


Graphical representation

- The term *upcasting* is used because of the way inheritance is often represented

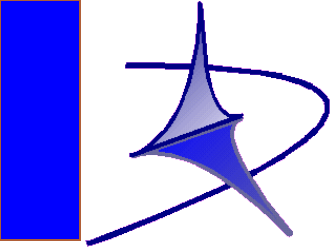


This is UML
If we have a reference to B, we can cast implicitly to a reference to A
a reference to A *cannot* be cast implicitly to B (*downcast*)



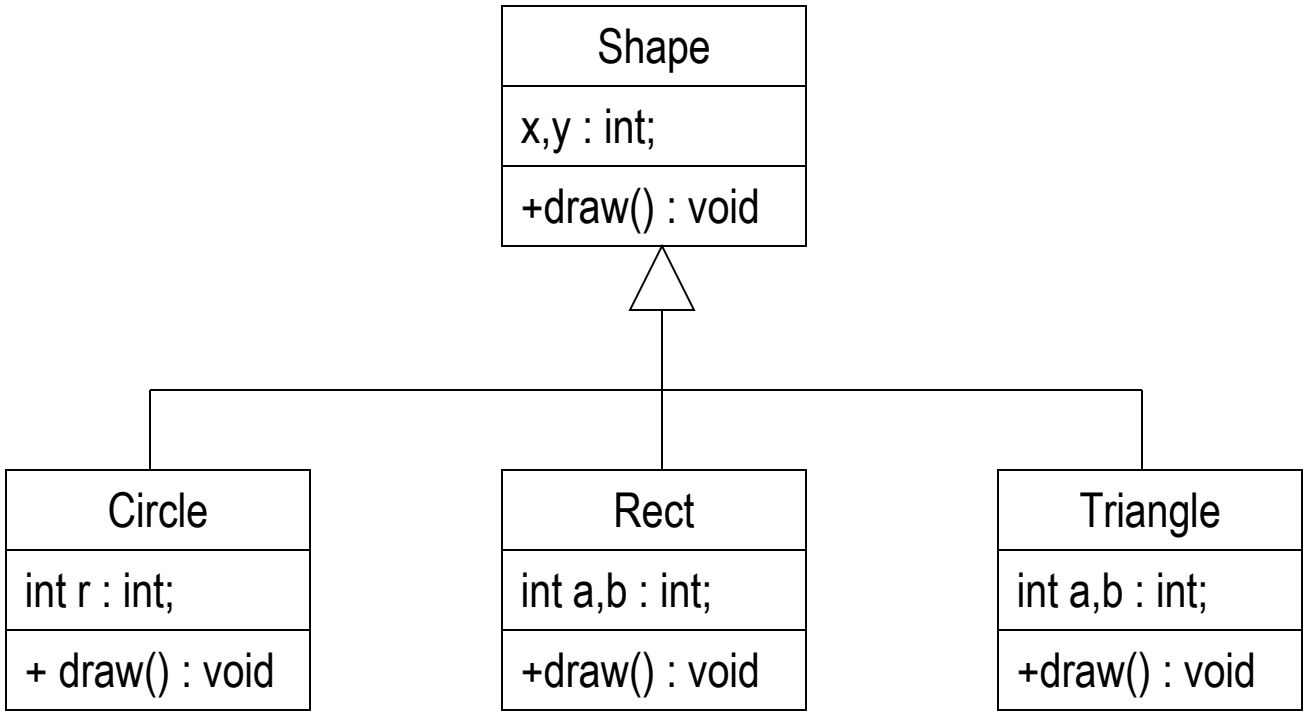
Upcasting and downcasting

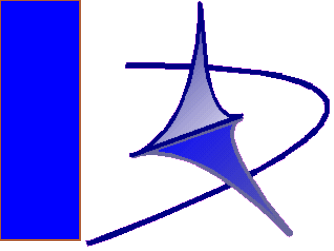
- Upcasting is a fundamental activity in OO programming (and it is safe)
- Downcasting is not safe at all, so the compiler will issue an error when you try to implicitly downcast
- To better understand *upcasting*, we need to introduce *virtual functions*



Virtual functions

- Let's introduce virtual functions with an example



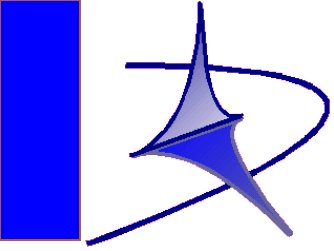


We would like to collect shapes

- Let's make a vector of shapes

```
vector<Shapes *> shapes;

shapes.push_back(new Circle(2,3,10));
shapes.push_back(new Rect(10,10,5,4));
shapes.push_back(new Triangle(0,0,3,2));

// now we want to draw all the shapes…

for (int i=o; i<3; ++i) shapes[i]->draw();
```

We would like that the right draw function is called
However, the problem is that Shapes::draw() is called
The solution is to make draw *virtual*

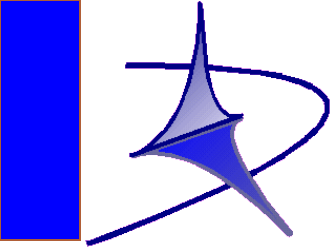


Virtual functions

```
class Shape {
protected:
   int xx,yy;
public:
    Shape(int x,y);
    void move(int x,int y);
    virtual void draw();
    virtual void resize(int scale);
    virtual void rotate(int degree);
}
```

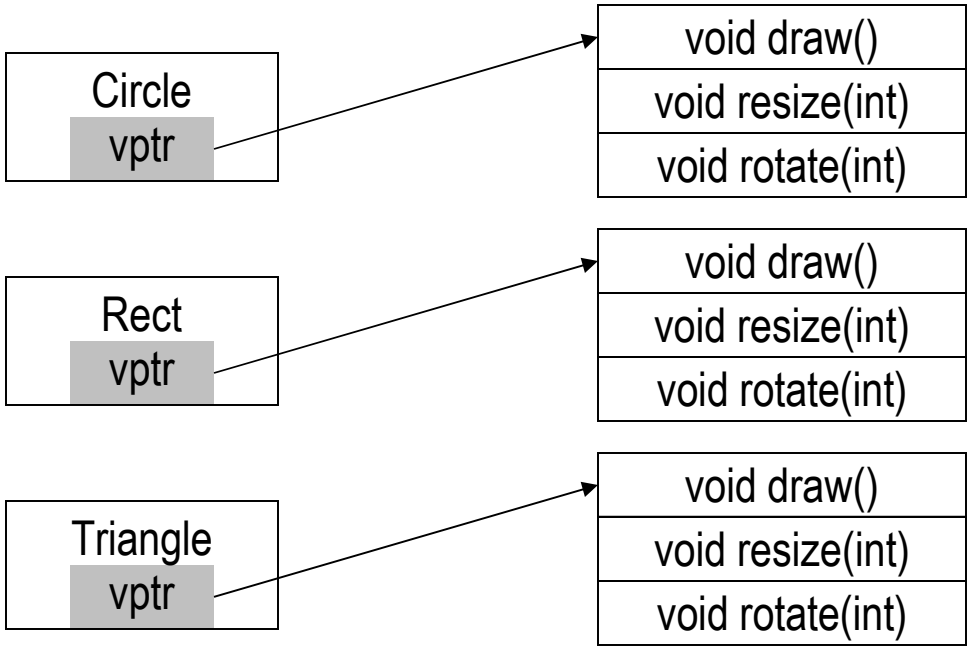
```
class Circle : public Shape {
   int rr;
public:
    Circle(int x,y,int r);
    virtual void draw();
    virtual void resize(int scale);
    virtual void rotate(int degree);
}
```

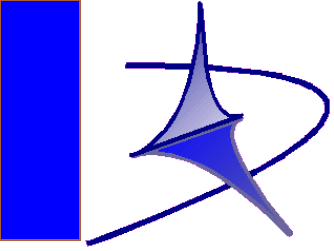
- move() is a regular function
- draw(), resize() and rotate() are virtual
- see shapes/



Virtual table

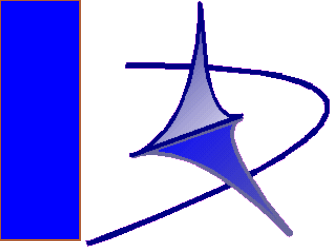
- When you put the *virtual* keyword before a function declaration, the compiler builds a *vtable* for each class





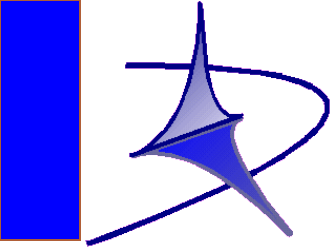
Calling a virtual function

- When the compiler sees a call to a virtual function, it performs a *late binding*, or *dynamic binding*
  - each class derived from Shape has a vptr as first element. It is like a hidden member variable
- So, the virtual function call is translated into
  - get the *vptr*
  - move to the right position into the *vtable*
  - call the function



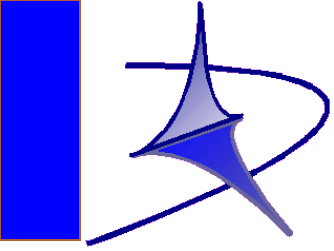
Equivalent in C

- It is easy to replicate this behavior in C
  - it suffices to use array of pointers to functions
  - However, in C this has to be done explicitly
  - It is not nice code, it is error-prone
- In C++, it is automatic!
  - it is quite efficient,
    - if you look at the generated assembler code, it is just two assembler instructions more than a regular function call
  - let's go back to *upcasting and downcasting*



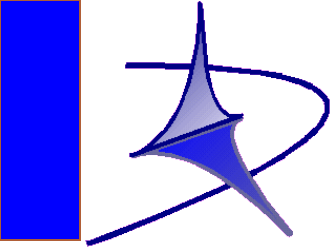
Examples

- See shapes/
- See virtual/



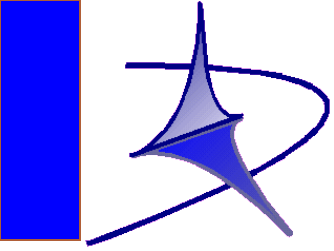
When inheritance is used

- Inheritance should be used when we have a *isA* relation between objects
  - you can say that a circle *is a* kind of shape
  - you can say that a rect *is a* shape
- What if the derived class contains some special function that is useful only for that class?
  - Suppose that we need to compute the diagonal of a rectangle



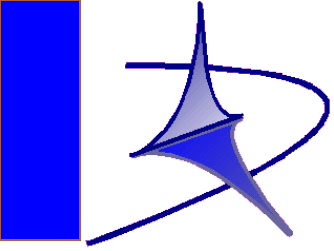
isA vs. *isLikeA*

- If we put function diagonal() only in Rect, we cannot call it with a pointer to shape
  - in fact, diagonal() is not part of the interface of shape
- If we put function diagonal() in Shape, it is inherited by Triangle and Circle
  - diagonal() does not make sense for a Circle… we should raise an error when diagonal is called on a Circle
- What to do?



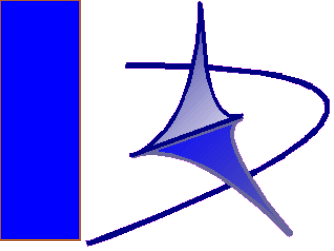
The fat-interface

- one solution is to put the function in the Shape interface
  - it will return an error for the other classes like Triangle and Circle
- another solution is to put it only in Rect and then make a downcasting when necessary
  - see diagonal/ for the two solutions
- This is a problem of inheritance! Anyway, the second one it probably better



Overloading and overriding

- When you override a virtual function you cannot change the return value
  - when the function is not virtual, you can do it!!
- There is an exception to the previous rule:
  - if the base class virtual method returns a pointer or a reference to an object of the base class…
  - … the derived class can change the return value to a pointer or reference of the derived class

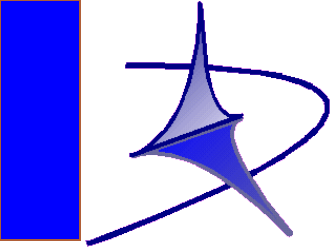


Overload and override

- An example

```
class A {
public:
   virtual A& f();
   int g();
};

class B: public A {
public:
   virtual B& f();
   double g();
};
```

```
class A {
public:
   virtual A& f();
};

class C: public A {
public:
   virtual int f();
};
```



Private inheritance

- A base class can be inherited as private, instead of public:

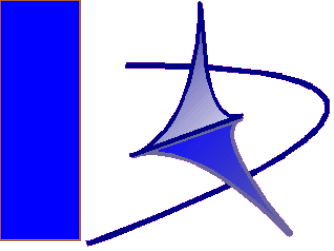
```
class A {
protected:
  void f();
public:
  int g();
};
```

```
class B : public A {
public:
  int h();
};
```

```
int main() {
  B b1;
  b1.f();    // NO
  b1.g();   // OK
}
```

```
class C : private A {
public:
  int h();
};
```

```
int main() {
  C c1;
  c1.f();   // NO
  c1.g();    // NO
}
```



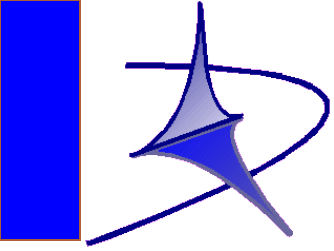
Destructors

- What happens if we try to destruct an object through a pointer to the base class?

```
class A {
public:
  A();
  ~A();
}
```

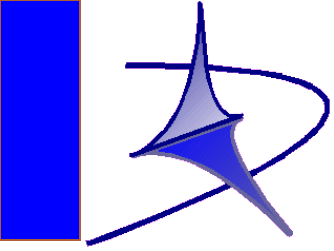
```
class B : public A {
public:
  B();
  ~B();
}
```

```
int main() {
  A *p;
  p = new B;
  …;
  delete p;
}
```



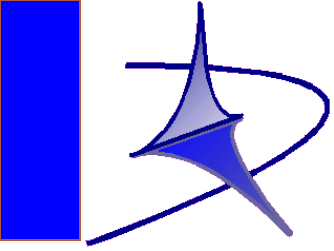
Virtual destructor

- In this case, we have to declare a virtual destructor
  - If the destructors are virtual, they are called in the correct order
- Never call a virtual function inside a destructor!



Restrictions

- You can not call a virtual function inside a constructor
  - in fact, in the constructor, the object is only half-built, so you could end up making a wrong thing
  - during construction, the object is not yet ready! The constructor should only build the object
- Same thing for the destructor
  - during destruction, the object is half destroyed, so you will probably call the wrong function



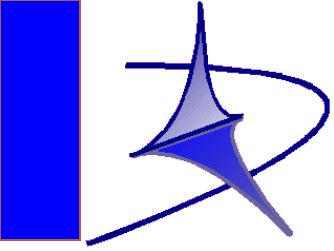
Restrictions

- Example

```
class Base {
  string name;
public:
  Base(const string &n) : name(n) {}
  virtual string getName() { return name; }
  virtual ~Base() { cout << getName() << "\n";}
};
```

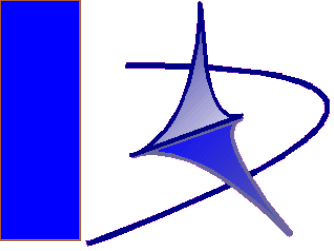
which function is called? Suppose we are destroying a object of Derived

```
class Derived : public Base {
  string name2;
public:
  Derived(const string &n) : Base(n), name(n + "2") {}
  virtual string getName() {return name2;}
  virtual ~Derived() {}
};
```



An exercise on inheritance

- You have to build a list of shapes;
  - every shape has a position, a name, and a dimension
  - you can add a circle, a rectangle or a triangle
  - Re-use your List container!
- The user can
  - ask to add a shape, or remove a shape;
  - Ask for info on a particular shape
    - name, position, area, dimensions
  - Operate a transformation on a shape

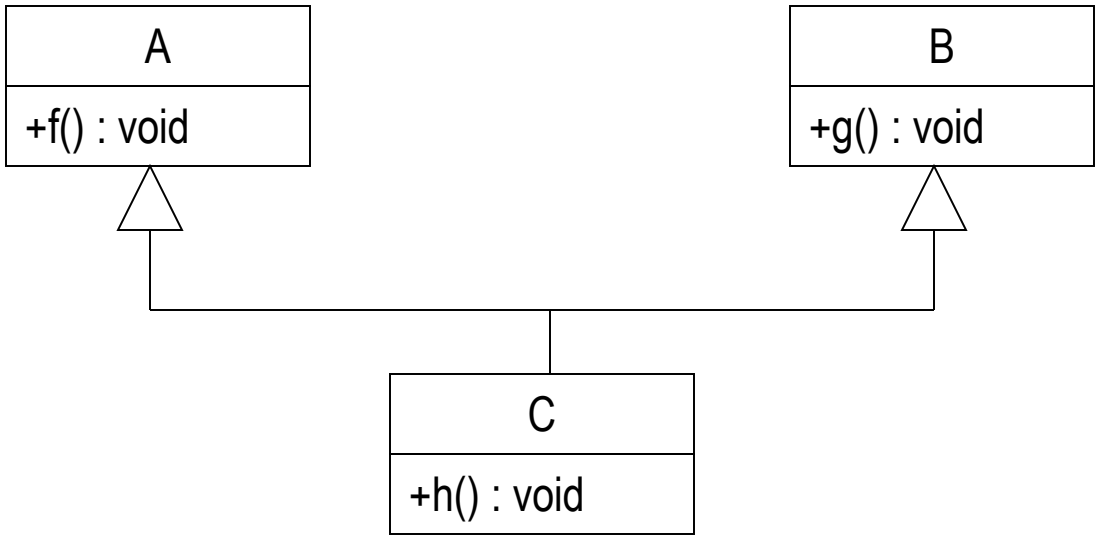


Exercise

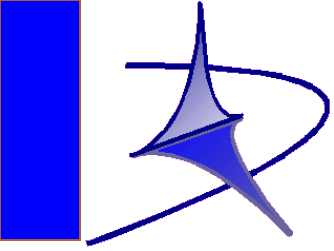
- Re-write the SmartPointer, for dealing with inheritance

Multiple inheritance

- A class can be derived from 2 or more base classes



C inherits the members of A and B



Multiple inheritance

- Syntax

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

```
class A {
public:
   void f();
};
```

```
class B {
public:
   void f();
};
```

If both A and B define two functions with the same name, there is an ambiguity

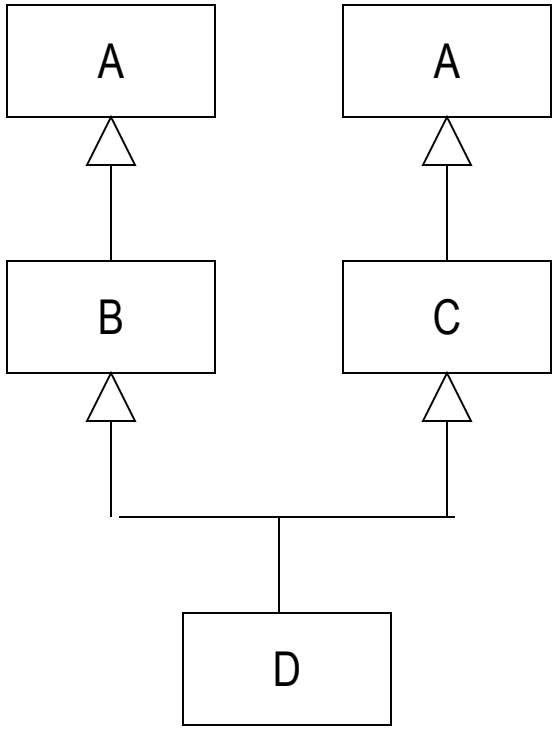
it can be solved with the scope operator

```
C c1;

c1.A::f();
c1.B::f();
```

Virtual base class

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

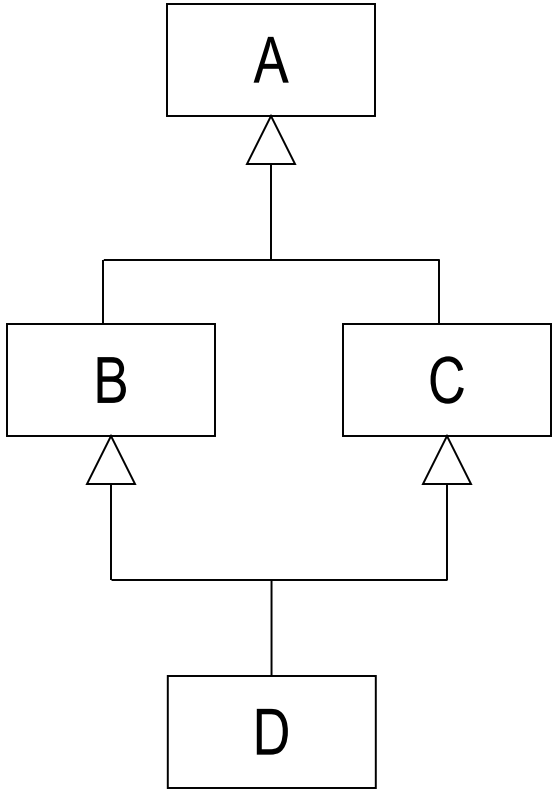
With public inheritance the base class is *duplicated*
If we want only one base class (diamond), we have to specify *virtual*

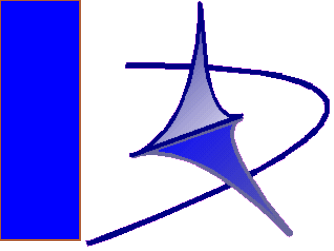


Virtual base class

```
class A {…};
class B : virtual public A {…};
class C : virtual public A {…};
class D : public B, public C {…};
```

With virtual public inheritance the base class is inherited only once





The diamond problem

- If the base class calls a virtual function, this function must be “finalized” in the last derived class
  - otherwise the compiler will raise an error
  - see multiple-inheritance/