

Conception Objets Avancée

Inheritance

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Inheritance

Virtual functions

Abstract classes

Downcasting

Slicing

Multiple inheritance

Outline

Inheritance

Virtual functions

Abstract classes

Downcasting

Slicing

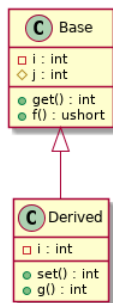
Multiple inheritance

Code reuse

- ▶ 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**
 - ▶ it consists of using an object with its interface inside another object
- ▶ **Inheritance**
 - ▶ Inheritance consists in enhancing an existing class with new, more specific code
- ▶ Most of the times, the two mechanism must work together

Inheritance

► Class Diagram

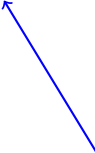


```
class Base {
    int i;
protected:
    int j;
public:
    Base() : i(0),j(0) {};
    ~Base() {};
    int get() const {return i;}
    int f() const {return j;}
};

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

Syntax

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



class Derived derives publicly from
Base

Syntax

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

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

Syntax

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

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

j is a member of Base declared as protected; therefore, Derived can access it

Syntax

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

class Derived derives publicly from Base

Therefore, to construct Derived, we must first construct Base

j is a member of Base declared as protected; therefore, Derived can access it

i is a member of Derived. There is another i that is a private member of Base, so it cannot be accessed from Derived

Use of Inheritance

- Now we can use Derived as a special version of Base

```
int main()
{
    Derived b;
    cout << b.get() << endl; // calls A::get();
    b.set(10);
    cout << b.g() << endl;
    b.g();
    Base *a = &b; // Automatic type conversion (upcasting)
    a->f();
    Derived *p = new Base; // error!
}
```

- See [examples/example1.cpp](#)

Public Inheritance

- ▶ Public inheritance means that the derived class *inherits* the same interface of the base class
 - ▶ All members in the public part of Base are also part of the public part of Derived
- ▶ All members in the protected part of Base are part of the protected part of Derived
- ▶ All members in the private part of Base are not accessible from Derived.
- ▶ This means that if we have an object of type Derived, we can use all functions defined in the public part of Derived **and** all functions defined in the public part of Base.

Overloading and hiding

- ▶ There is no overloading across classes

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

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

- ▶ `A::f()` has been hidden by `B::f()`
- ▶ to get `A::f()` into scope, you can use the using directive:

```
using A::f(int, double);
```

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

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

```
A* p;  
p = new B();  
p->f();  
p->g();
```

A pointer to the base class

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

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

```
A* p;  
p = new B();  
p->f();  
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

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

```
A* p;  
p = new B();  
p->f();  
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Call a function of the interface of the base class: correct

Upcasting

- It is possible to use an object of the derived class through a pointer to the base class.

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

```
A* p;  
p = new B();  
p->f();  
p->g();
```

A pointer to the base class

The pointer now points to an object of a derived class

Call a function of the interface of the base class: correct


Error! g() is not in the interface of the base class, so it cannot be called through a pointer to the base class!

References

- ▶ Same thing is possible with references

```
class A {  
public:  
    void f() { ... }  
};  
class B : public A {  
public:  
    void g() { ... }  
};  
  
void h(A &x)  
{  
    x.f();  
    x.g();  
}
```

Function h takes a reference to the base class



```
B obj;  
h(obj);
```

References

- ▶ Same thing is possible with references

```
class A {  
public:  
    void f() { ... }  
};  
class B : public A {  
public:  
    void g() { ... }  
};  
  
void h(A &x)  
{  
    x.f();  
    x.g();  
}
```

```
B obj;  
h(obj);
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

References

- ▶ Same thing is possible with references

```
class A {  
public:  
    void f() { ... }  
};  
class B : public A {  
public:  
    void g() { ... }  
};  
  
void h(A &x)  
{  
    x.f();  
    x.g();  
}
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

This is an error! g() is not in the interface of A

```
B obj;  
h(obj);
```

References

- ▶ Same thing is possible with references

```
class A {  
public:  
    void f() { ... }  
};  
class B : public A {  
public:  
    void g() { ... }  
};  
  
void h(A &x)  
{  
    x.f();  
    x.g();  
}
```

```
B obj;  
h(obj);
```

Function h takes a reference to the base class

Of course, it is possible to call functions in the interface of the base class

This is an error! g() is not in the interface of A

Calling the function by passing a reference to an object of a derived class: correct.

Extension through inheritance

- ▶ Why this is useful?
 - ▶ All functions that take a reference (or a pointer) to A as a parameter, continue to be valid and work correctly when we pass a reference (or a pointer) to B
 - ▶ This means that we can **reuse** all code that has been written for A, also for B
 - ▶ In addition, we can write additional code specifically for B
 - ▶ However, notice that, until now, to access a function of class B, we need a pointer or a reference to class B.
- ▶ We need also to modify (customize, extend, etc.) the behaviour of existing code
 - ▶ we need to call a function of class B through a pointer to the base class A.

Outline

Inheritance

Virtual functions

Abstract classes

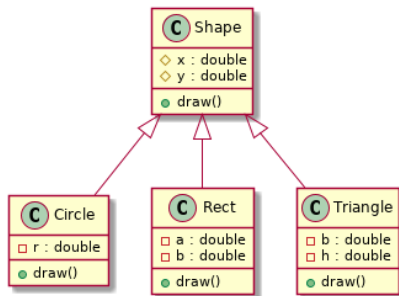
Downcasting

Slicing

Multiple inheritance

Virtual functions

- ▶ Let's introduce virtual functions with an example:



Implementation

```
class Shape {
protected:
    double x,y;
public:
    Shape(double x1, double y2);
    virtual void draw() = 0;
};
```

```
class Circle : public Shape {
    double r;
public:
    Circle(double x1, double y1,
           double r);
    virtual void draw();
};
```

```
class Rect : public Shape {
    double a, b;
public:
    Rect(double x1, double y1,
         double a1, double b1);
    virtual void draw();
};
```

```
class Triangle : public Shape {
    double a, b;
public:
    Triangle(double x1, double y1,
            double a1, double b1);
    virtual void draw();
};
```

Collecting Shapes

Let's make an array of Shapes:

```
Shapes * shapes[3];

shapes[0] = new Circle(2,3,10);
shapes[1] = new Rect(10,10,5,4);
shapes[2] = new Triangle(0,0,3,2);

// now we want to draw all the shapes ...
for (int i=0; i<3; ++i) shapes[i]->draw();
```

- ▶ The draw() of each object is called because draw() is declared virtual

Virtual vs. regular methods

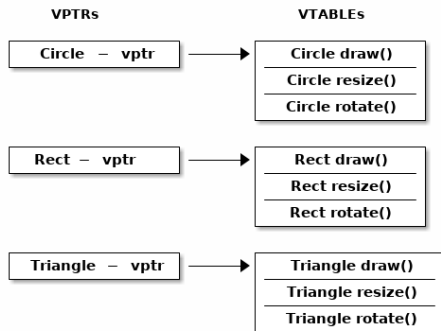
```
class Shape {
protected:
    double x,y;
public:
    Shape(double xx, double yy);
    void move(double x, double y);
    virtual void draw();
    virtual void resize(double scale);
    virtual void rotate(double degree);
};

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

- ▶ `move()` is a regular method
- ▶ `draw()`, `resize()` and `rotate()` are virtual.
- ▶ see [examples/shape.hpp](#)

Virtual table

- ▶ When you put the `virtual` keyword in front of a function, the compiler builds a **vtable** for the class
- ▶ It is an array of pointers to functions



Calling a virtual function

- ▶ When compiling the code
 - ▶ For each class → one VTABLE
 - ▶ For each object → one VPTR (first element of the object in memory)
- ▶ When the compiler sees a call to a virtual function, it performs a **late binding**, or **dynamic binding**
 - ▶ get the vptr
 - ▶ move to the right position into the vtable (depending on which virtual function we are calling)
 - ▶ call the function

Dynamic vs static binding

- Which functions are called in the following code?

```
class A {  
public:  
    void f() { cout << "A::f()" << endl; g(); }  
    virtual void g() { cout << "A::g()" << endl; }  
};  
class B : public A {  
public:  
    void f() { cout << "B::f()" << endl; g(); }  
    virtual void g() { cout << "B::g()" << endl; }  
};  
...  
  
A *p = new B;  
p->g();  
p->f();  
  
B b;  
A &r = b;  
r.g();  
r.f();
```

Overloading and Overriding

- ▶ The virtual function in all derived class must have exactly the same prototype as the virtual function in the base class
 - ▶ otherwise it is a different function
- ▶ In particular, the return type must be the same
- ▶ There is only one exception to this 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

Correct

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

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

Wrong

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

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

Virtual 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;  
}
```

Big mistake!

- ▶ The destructor of the base class is called, which *destroys* only part of the object
- ▶ Soon a segmentation fault. . .
- ▶ Solution: declare the destructor as **virtual**

Restrictions

- ▶ Calling a virtual function from constructor/destructor **is not** a good idea
 - ▶ In Java, this is an ERROR, and should never be done
- ▶ In C++:
 - ▶ if you call a virtual function in the constructor of the base class, the base class method is called, even if you are constructing an object of a derived class
 - ▶ if you call a virtual function in the destructor of the base class, the base class method is called, even if you are destroying an object of a derived class
- ▶ So, unlike Java, C++ is safe, however this causes confusion in programmers, and should be avoided
- ▶ See [C++FaqLite](#) and [Stackoverflow](#)

Example

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

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

Outline

Inheritance

Virtual functions

Abstract classes

Downcasting

Slicing

Multiple inheritance

Pure virtual functions

- ▶ A virtual function is pure if no implementation is provided

```
class Abs {  
public:  
    virtual int fun() = 0;  
    virtual ~Abs();  
};  
class Derived public Abs {  
public:  
    Derived();  
    virtual int fun();  
    virtual ~Derived();  
};
```

This is a pure virtual function. No object of Abs can be instantiated.

One of the derived classes must *finalize* the function to be able to instantiate the object.

Interface classes

- ▶ An abstract class is a class that contains a pure virtual method
 - ▶ cannot be instantiated
- ▶ An interface class is an abstract class that contains only pure virtual methods
 - ▶ Unlike Java, there is no special keyword to denote an interface

Outline

Inheritance

Virtual functions

Abstract classes

Downcasting

Slicing

Multiple inheritance

Use of inheritance

- ▶ Inheritance should be used when we have a **isA** relation between objects
 - ▶ you can say that a circle is a 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
 - ▶ see [examples/shape_diag/rect.hpp](#)

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`
- ▶ 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 `examples/shape_diag/shapes_main.cpp` for the two solutions

Downcasting

- ▶ One way to downcast is to use the `dynamic_cast` construct

```
class Shape { ... };

class Circle : public Shape { ... };

void f(Shape *s)
{
    Circle *c;

    c = dynamic_cast<Circle *>(s);
    if (c == 0) {
        // s does not point to a circle
    }
    else {
        // s (and c) points to a circle
    }
}
```

Casting

- ▶ Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- ▶ The subsequent call to member will produce either a run-time error or a unexpected result.
- ▶ There are safer ways to perform casting

```
dynamic_cast <new_type> (expression)  
reinterpret_cast <new_type> (expression)  
static_cast <new_type> (expression)  
const_cast <new_type> (expression)
```

Dynamic cast

- ▶ It can be used only with pointers and references to objects.
 - ▶ The cast is solved at run-time, by looking inside the structure of the object
 - ▶ This feature is called run-time type identification (RTTI)
- ▶ Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
 - ▶ The result is the pointer itself if the conversion is possible;
 - ▶ The result is nullptr if the conversion is not possible:

```
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
    return 0;
}
```

static_cast

- ▶ `static_cast` can perform conversions between pointers to related classes
- ▶ however, no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- ▶ Therefore, it is up to the programmer to ensure that the conversion is safe.

```
class CBase {};  
class CDerived: public CBase {};  
CBase * a = new CBase;  
CDerived * b = static_cast<CDerived*>(a);
```

- ▶ `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

reinterpret_cast

- ▶ Converts any pointer type to any other pointer type, even of unrelated classes.
- ▶ The result of the operation is a simple binary copy of the value from one pointer to the other.
- ▶ All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
- ▶ It can also cast pointers to or from integer types.
- ▶ This can be useful in low-level non portable code (i.e. interaction with interrupt handlers, device drivers, etc.)

const_cast

- ▶ This type of casting manipulates the constness of the type, either to be set or to be removed.
- ▶ For example, in order to pass a const argument to a function that expects a non-constant parameter

```
// const_cast
#include <iostream>
using namespace std;

void print (char * str)
{
    cout << str << endl;
}

int main () {
    const char * c = "sample text";
    print ( const_cast<char *> (c) );
    return 0;
}
```

Outline

Inheritance

Virtual functions

Abstract classes

Downcasting

Slicing

Multiple inheritance

What happens ?

Consider the following code snippet

```
class Employee {  
    // ...  
    Employee& operator=(const Employee& e);  
    Employee(const Employee& e);  
};  
  
class Manager : public Employee {  
    // ...  
};  
  
void f(const Manager& m)  
{  
    Employee e;  
    e = m;  
}
```

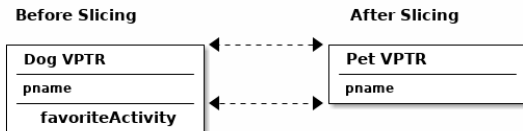
Slicing

Answer: Only the `Employee` part of `m` is copied from `m` to `e`.

- ▶ The assignment operator of `Employee` does not know anything about managers!
- ▶ This is called “object slicing” and it can be a source of errors and various problems

Another example

- ▶ If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast.
- ▶ Consider the code in `examples/slicing.cpp`
- ▶ any calls to `describe()` will cause an object the size of `Pet` to be pushed on the stack
- ▶ the compiler copies only the `Pet` portion of the object and slices the derived portion off of the object, like this:



Slicing cont.

- ▶ What happens to the virtual function call?
- ▶ The compiler is smart, and understand what is going on!
 - ▶ the compiler knows the precise type of the object because the derived object has been forced to become a base object.
 - ▶ When passing by value, the copy-constructor for a Pet object is used, which initialises the VPTR to the Pet VTABLE and copies only the Pet parts of the object.
 - ▶ There's no explicit copy-constructor here, so the compiler synthesises one.
 - ▶ Under all interpretations, the object truly becomes a Pet during slicing.

Private and protected inheritance

- ▶ Until now we have seen **public inheritance**
 - ▶ A derived class inherits the interface **and** the implementation of a base class
- ▶ With **private inheritance** it is possible to inherit only the implementation

```
class Base {  
    int p;  
protected:  
    int q;  
public:  
    int f();  
};
```

```
class Derived : private Base {  
public:  
    int g();  
};  
int main() {  
    Derived obj;  
    obj.g();  
}
```

Private inheritance

Can access q and f()

I can only call g() but not f()

Private inheritance

- ▶ Private inheritance does **not** model the classical **isA** relationship
- ▶ In particular, it is not possible to automatically upcast from derived to base class

```
class Base {};  
class DerivedA : public Base {};  
class DerivedB : private Base {};
```

```
Base *ptr;  
DerivedA pub;  
DerivedB priv;
```

```
ptr = &pub;  
// error!!  
ptr = &priv;
```

DerivedB cannot be accessed as Base

Inheritance Rules

► Protected Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as protected members	cannot access

► Public Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	as protected members	cannot access
public	as public members	access

► Private Inheritance

In Base	In Derived	Client
private	cannot access	cannot access
protected	cannot access	cannot access
public	as private members	cannot access

Private Inheritance

- ▶ Why private inheritance?
 - ▶ Because we want to re-use implementation but not the interface
 - ▶ It can be seen as a sort of composition
 - ▶ When to use it
 - ▶ Not a popular technique, composition is better done by declaring a member to another class
-

Composition

```
class B {  
    A* ptr;  
public:  
    B() { ptr = new A(); }  
    ~B() { delete ptr; }  
};
```

Private Inheritance

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

Outline

Inheritance

Virtual functions

Abstract classes

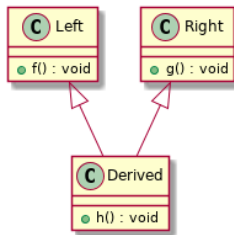
Downcasting

Slicing

Multiple inheritance

Multiple inheritance

- ▶ In C++, a class can be derived from 2 or more base classes



- ▶ Derived inherits the members from Left and from Right

Syntax

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

- ▶ If both A and B define two functions with the same name, there is an ambiguity
- ▶ Use the scope operator to solve it

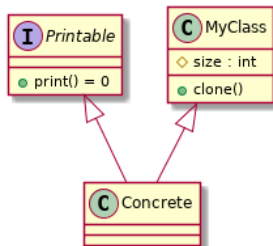
```
C c1;  
  
c1.A::f();  
c1.B::f();
```

Why multiple inheritance ?

- ▶ Is multiple inheritance really needed?
 - ▶ There are contrasts in the OO research community
 - ▶ Many OO languages do not support multiple inheritance
 - ▶ Some languages support the concept of “Interface” (e.g. Java)
- ▶ Multiple inheritance can bring several problems both to the programmers and to language designers
- ▶ Therefore, the much simpler **interface inheritance** is used (that mimics Java interfaces)

Interface inheritance

- ▶ It is called interface inheritance when an object derives from a base class and from an **interface class**
- ▶ A simple example:

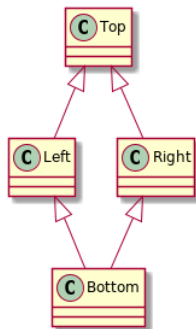


Interface and implementation inheritance

- ▶ In **interface inheritance**
 - ▶ The base class is abstract (only contains the interface)
 - ▶ For each method there is only one final implementation in the derived classes
 - ▶ It is possible to always understand which function is called
- ▶ Implementation inheritance is the one normally used by C++
 - ▶ the base class provides some implementation
 - ▶ when inheriting from a base class, the derived class inherits its implementation (and not only the interface)

The diamond problem

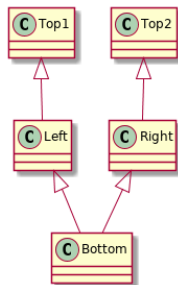
- ▶ What happens if class Bottom inherits from two classes, Left and Right which both inherit from Top?
- ▶ This may be a problem in object oriented programming with multiple inheritance!



- ▶ If a method in Bottom calls a method defined in Top ...
- ▶ ... and Left and Right have overridden that method differently,
- ▶ From which class does Bottom inherit the method: Left, or Right?
- ▶ In C++ this is solved by using keyword “virtual” when inheriting from a class

Non virtual base

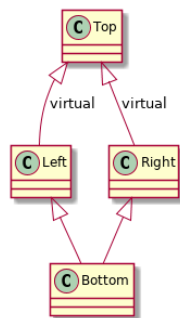
```
class Top {...};  
class Left : public Top {...};  
class Right : public Top {...};  
class Bottom : public Left, public Right  
{  
    ...  
};
```



- ▶ With public inheritance the base class is duplicated
- ▶ To use one of the methods of A, we have to specify which “path” we want to follow with the scope operator
- ▶ see [examples/duplicate.cpp](#)

Virtual Base

```
class Top {...};  
class Left : virtual public Top {...};  
class Right : virtual public Top {...};  
class Bottom : public Left, public Right {...};
```



- ▶ With **virtual** public inheritance the base class is inherited only once
- ▶ see [examples/vbase.cpp](#)

Initializing virtual base

- ▶ The strangest thing in the previous code is the initializer for `Top` in the `Bottom` constructor.
- ▶ Normally one doesn't worry about initializing sub-objects beyond direct base classes, since all classes take care of initializing their own bases.
- ▶ There are, however, multiple paths from `Bottom` to `Top`,
 - ▶ who is responsible for performing the initialization?
- ▶ For this reason, the most derived class must initialize a virtual base.
- ▶ But what about the expressions in the `Left` and `Right` constructors that also initialize `Top`?
 - ▶ They are ignored when a `Bottom` object is created
 - ▶ The compiler takes care of all this for you