Object Oriented Software Design II

Inheritance

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

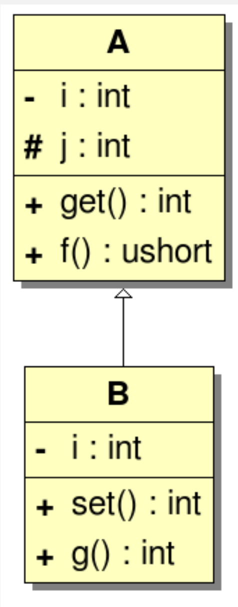
February 29, 2012

Outline

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 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

Inheritance




```
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;}
};
```

Syntax

- How to define the derived class

```
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;
  }
};
```

- class B derives publicly from A
- Therefore, to construct B, we must first construct A
- `j` is a member of A declared as `protected`; therefore, B can access it
- `i` instead is a member of B. There if another `i` that is a `private` member of A, so it cannot be accessed from B

Use of Inheritance

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

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

- See `./examples/04.inheritance-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 A are also part of the `public` part of B
  - All members in the `protected` part of A are part of the `protected` part of B
  - All members in the private part of A are not accessible from B.
- This means that if we have an object of type B, we can use all functions defined in the `public` part of B **and** all functions defined in the `public` part of A.

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, the `using` directive is necessary
- `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

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 &)
{
    h.f();
    h.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

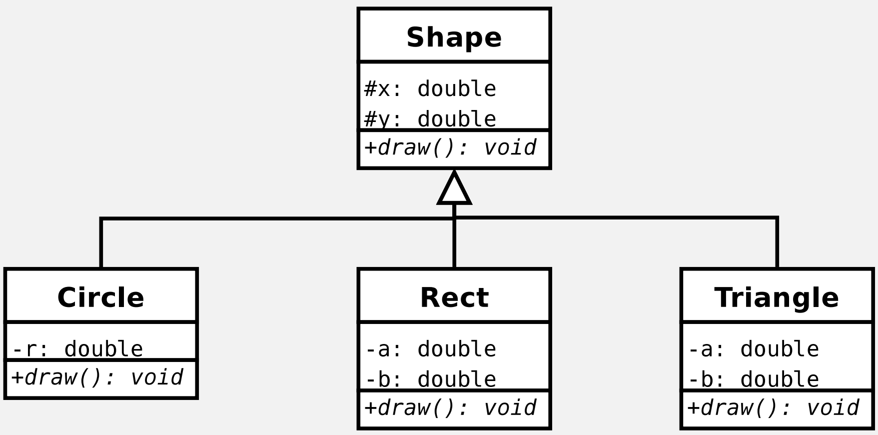
Calling the function by passing a reference to an object of a derived class: correct.

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
- Therefore,
  - we can **reuse** existing code also with the new class
  - We can extend existing class to implement new functionality
- What about modifying (customize, extend, etc.) the behaviour of existing code *without changing it*?

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();
};
```


We would like to collect 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();
```

- 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

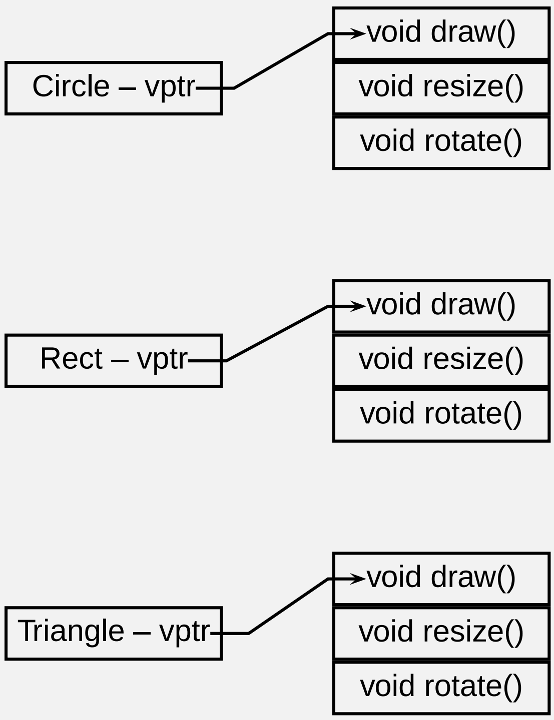
```
class Shape {
protected:
  double x,y;
public:
  Shape(double xx, double yy);
  void move(double x, double y);
  virtual void draw();
  virtual void resize(double scale);
  virtual void rotate(double degree);
};

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 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 object of a class derived from `Shape` has a `vptr` as first element.
    - It is like a hidden member variable
- The virtual function call is translated into
  - get the `vptr` (first element of object)
  - move to the right position into the vtable (depending on which virtual function we are calling)
  - call the function

Dynamic binding vs static binding

Which function 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

- When you override a virtual function, you cannot change the return value
  - Simply because the compiler will not know which function to actually call
- There is only one 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

- Examples

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();
};
```

Overloading and overriding

- When you override a virtual function, you cannot change the return value
  - Simply because the compiler will not know which function to actually call
- There is only one 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

- Examples

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();
};
```

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

- This is a big mistake!
  - The destructor of the base class is called, which “destroys” only part of the object
  - You will soon end up with a segmentation fault (or illegal access), or memory corruption
- To solve the problem, we have to declare a virtual destructor
  - If the destructors are virtual, they are called in the correct order

Restrictions

- Never call a virtual function inside a destructor!
  - Can you explain why?
- 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() << endl;}
};
```

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

Pure virtual functions

- A virtual function is pure if no implementation is provided
- Example:

```
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

- If a class only provides pure virtual functions, it is an *interface class*
  - an interface class is useful when we want to specify that a certain class *conforms* to an interface
  - Unlike Java, there is no special keyword to indicate an interface class
  - more examples in section multiple inheritance