Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Abstract Base Classes and copy constructors

Covariant return type

- An overridden method in a derived class can return a type derived from the type returned by the base-class method.

```
class Base {
public:
  virtual Base* clone() const;
};

class Derived : public Base {
public:
  virtual Derived* clone() const;
};

Derived* Derived::clone() const {
  return new Derived( *this );
}
```

```
Derived orig;
Base* pB = &orig;
Derived* clonedObj = pB->clone();
// clonedObj gets a clone of orig
```

"virtual" constructor: why ?

- Virtual constructors do not exist: `virtual` allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object we need to know the exact type of what you want to create: i.e. we need complete information.
- We can mimic the behavior of a virtual copy constructor, though…

"virtual" constructor: how ?

- Declare two virtual methods in the base class:
  - clone() for copy constructor
  - create() for default constructor
- They can be purely virtual
- Implement the methods in the derived classes, using covariant return type
- Just return new objects or new copies

Example: base class

```
class Shape {
public:
    Shape(int x=0, int y=0) : x(x), y(y) {}

    virtual ~Shape() {}

    virtual Shape* clone() const = 0;   // The Virtual (Copy) Constructor

    virtual Shape* create() const = 0;  // uses the default constructor

    virtual void print() const = 0;

    // ...

protected:

    int x;

    int y;
};
```

Example: derived class 1

```
class Circle : public Shape {
public:
     Circle(int x, int y, int r=1) : Shape(x, y), radius(r) {}
     virtual Circle* clone() const;
    virtual Circle* create() const
     virtual void print() const;
     // ...

private:
     int radius;
};

Circle* Circle::clone() const {
     return new Circle(*this); // uses copy constructor
}

Circle* Circle::create() const {
     return new Circle(0, 0); // uses constructor
}

void Circle::print() const {
     std::cout << "x: " << x << " - y: " << y << " radius: " <<
                              radius << std::endl;
}
```

Example: derived class 2

```
class Square : public Shape {
public:
     Square(int x, int y, int s) : Shape(x, y), side(s) {}
     virtual Square* clone() const;
     virtual Square* create() const;
     virtual void print() const;
     // ...

protected:
     int side;
};

Square* Square::clone() const {
     return new Square(*this);
}

Square* Square::create() const {
     return new Square(0, 0, 10);
}

void Square::print() const {
     std::cout << "x: " << x << " - y: " << y << " side: " <<
                              side << std::endl;
}
```

Example: use of clone

```
void userCode(Shape& s) {

    Shape* s2 = s.clone();

    Shape* s3 = s.create();

    // ...

    delete s2;    // You need a virtual destructor here

    delete s3;

}
```

ABCs and copy constructors

- When working with classes that have a pointer to Abstract Base Classes we can not use directly the copy constructor of the ABC…
  - use the "virtual" constructor technique seen before:
    - declare a pure virtual clone() method in the abstract base class
    - implement it in the concrete derived classes
    - use the clone() method in the copy constructor of the class containing the pointer
    - use same technique for assignment operator

Example

```
class Fred {
public:
     // p must be a pointer returned by new; it must not be NULL
     Fred(Shape* pp) : p(pp) { }
     ~Fred() {
          delete p;
     }

     Fred(const Fred& f) : p(f.p->clone()) { }

     Fred& operator= (const Fred& f) {
          if (this != &f) {               // Check for self-assignment
               Shape* p2 = f.p->clone();   // Create the new one FIRST...
               delete p;                   // ...THEN delete the old one
               p = p2;
          }
          return *this;
     }

     void print() {
          p->print();
     }
     // ...

private:
     Shape* p;
};
```

Example: use

```
Shape* s1 = new Circle(3, 4, 5);

Shape* s2 = new Square(1, 2, 4);

Fred f1( s1 );

f1.print();

Fred f2( s2 );

f2.print();

Fred f3( f2 );

f3.print();

f2 = f1;

f2.print();
```

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", cap. 3 - pp. 82-85
- https://isocpp.org/wiki/faq/virtual-functions#virtual-ctors
- https://isocpp.org/wiki/faq/abcs#copy-of-abc-via-clone
- https://isocpp.org/wiki/faq/virtual-functions#virtual-ctor-rationale