COMP322 - Introduction to C++

Lecture 08 - Review of Classes plus OOP and Inheritance

Dan Pomerantz

School of Computer Science

5 March 2012

Why classes?

A *class* can be thought of as an abstract data type, from which we can create any number of *objects*.

A class in C++ allows us to do several useful things:

- Associate both code and data with an abstract data type.
- Hide implementation details from clients.
- Inherit functionality from one or more base (ancestor) classes, creating a class hierarchy.

We've already mentioned objects of stream and vector/list classes, as they are fundamental to doing anything useful in C++.

Declaring simple class

Here is the declaration of a very simple class for complex numbers, as might be found in a header file:

```
class fcomplex {
public:
  fcomplex(); // Default constructor
  fcomplex(float r, float i);  // Full constructor
  fcomplex add(const fcomplex &y);
  fcomplex sub(const fcomplex &y);
  fcomplex mul(const fcomplex &y);
  fcomplex div(const fcomplex &y);
  static float abs(const fcomplex &x);

  float realpart() const;
  float imagpart() const;

private:
  float real; // Real part
  float imag; // Imaginary part
};
```

Declaring simple class

Here is the declaration of a very simple class for complex numbers, as might be found in a header file:

Each declaration defines a *method* which will operate ON a particular element.

In the same way that we could write:

```
vector<int> foo;
foo.push_back(3);
```

and call the method *push_back* to add the element 3 to the vector foo, we will now be able to call the method `add` ON an `fcomplex` number, writing something like: (if number1, number2, and number 3 are all variables of type fcomplex)

```
fcomplex number3 = number1.add(number2);
```

Sometimes we will define our methods to change the existing object, other times to create a new one.

Implementing a simple class

Now let's implement some of these member functions:

```
fcomplex::fcomplex() { // Default constructor
  real = imag = 0.0;
}

fcomplex::fcomplex(float r, float i) {
  real = r;
  imag = i;
}

float fcomplex::realpart() const {
  return real;
}

fcomplex fcomplex::add(const fcomplex &y) {
  return fcomplex(real + y.real, imag + y.imag);
}

fcomplex fcomplex::mul(const fcomplex &y) {
  return fcomplex(real * y.real - imag * y.imag,
                  real * y.imag + imag * y.real);
}
```

Using our simple class

We can use the class as follows:

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

int main() {
  fcomplex a(1.0, 2.0);
  fcomplex b(2.0, 1.0);

  fcomplex c;

  c = a.mul(b);

  cout << c.realpart() << " + " << c.imagpart() << "i" << endl;
}
```

This code will print:

```
0 + 5i
```

Constructors

Each class can define one or more *constructors*. These are special functions which have the same name as the class, and have no defined return type.

The appropriate constructor is called automatically when an object is created.

The *default* constructor is the constructor with no arguments. It simply fills in a "reasonable" set of values.

In our example `main()` function, the declaration

```
fcomplex a(1.0, 2.0);
```

invokes the "full" constructor, while the declaration

```
fcomplex c;
```

invokes the default constructor.

Constructors with new

In cases that you wish to use a pointer, you can also use the new operator with the constructor:

```
fcomplex* a = new fcomplex(1.0,2.0);
```

Remember to delete a then!

Granting or denying access

We use the keywords `public`, `private`, and `protected` to specify how a member function or data object may be accessed:

- `public` - Globally visible.
- `private` - Visible only to other members of this very class.
- `protected` - Visible to this class and all of its descendants.

These restrictions can apply to any function or data member.

In our `main()` function we cannot access private members:

```
int main() {
  fcomplex a(1.0, 2.0);
  // ...
  a.imag = 1.0;     // Error! Not a public member.
}
```

Member functions

Unless specified otherwise, a member function is invoked by dereferencing a specific object:

```
c = a.mul(b);
```

The object through which we invoke the member function is an implicit parameter to the function. It may be accessed simply by using a member name:

```
fcomplex fcomplex::mul(const fcomplex &y) {
  return fcomplex(real * y.real - imag * y.imag,
                  real * y.imag + imag * y.real);
}
```

Destructors

A *destructor* is another “special” member function. The class destructor is called when an object of a given class is deleted. This gives an opportunity for the class to free memory or other resources.

The destructor always has the name $\sim \langle$*classname*$\rangle$:

```
class intStack {
  int top;
  int max;
  int *data;

  intStack(int max = 100) {  // Constructor
    Stack::max = max;
    data = new int[max];
  }
  @$\sim$@intStack() { // Destructor
    delete [] data;
  }

  int pop();
  void push(int);
};
```

What is inheritance?

- We've looked at classes in C++, which allow us to create abstract types with separate public declarations and private implementations.
- *Inheritance* refers to our ability to create a hierarchy of classes, in which *derived* classes (subclass) automatically incorporate functionality from their *base* classes (superclass).
- A derived class inherits all of the data and methods from its base class.
- A derived class may *override* one or more of the inherited methods.
- Most of the utility of classes and objects comes from the creation of class hierarchies.

Inheritance syntax

```
class A {       // base class
private:
  int x;        // Visible only within 'A'
protected:
  int y;        // Visible to derived classes
public:
  int z;        // Visible globally
  A();          // Constructor
  ~A();         // Destructor
  void f();     // Example method
};

class B : public A { // B is derived from A
private:
  int w;        // Visible only within 'B'
public:
  B();          // Constructor
  ~B();         // Destructor
  void g() {
    w = z + y; // OK
    f();        // OK
    w = x + 1; // Error - 'x' is private to 'A'
  }
};
```

Public inheritance

- The use of the `public` keyword is the norm although in some rare circumstances you will use `private` or `protected`.
- If you omit the `public` keyword, inheritance is `private`.

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

class B: A { // B inherits A privately
public:  void g();
};

int main() {
  A a;
  B b;
  a.f(); // OK
  b.g(); // OK
  b.f(); // Illegal
}
```

Overriding member functions

A derived class may *override* a function from its base class:

```
class A {
public:
  void f(int x) { cerr << "A::f(" << x << ")\n"; }
};

class B: public A {
public:
  void f(int x) { cerr << "B::f(" << x << ")\n"; }
};

int main() {
  A a;
  B b;
  a.f(1);
  b.f(2);
}
```

the `main()` program will print:

```
A::f(1)
B::f(2)
```

Calling the base class

Overridden functions do not automatically invoke the base class implementation. We have to do this explicitly:

```
class B: public A {
public:
  void f(int x) {
    A::f(x);          // Call the base class
    cerr << "B::f(" << x << ")\n";
  }
};
```

the prior `main()` would now print:

```
A::f(1)
A::f(2)
B::f(2)
```

Because of *multiple inheritance*, C++ does not offer the Java `super()` construct.

Inheritance and constructors

- Special provisions are made for inheritance of constructors and destructors.
- Constructors are inherited, and the constructors of base classes are automatically invoked before the constructor of the derived class.
- The same is true of destructors.
- This is not true of other methods, they are *not* invoked automatically from overridden functions.

Inheritance and constructors

```
class A {
public:
  A() { cerr << "A()\n"; }
  ~A() { cerr << "~A()\n"; }
  void f() { cerr << "A::f()\n"; } // Not special
};

class B: public A {
public:
  B() { cerr << "B()\n"; }
  ~B() { cerr << "~B()\n"; }
  void f() { cerr << "B::f()\n"; }
};

int main() {
  A a;
  a.f();
  B b;
  b.f();
}
```

Inheritance and constructors

This program:

```
int main() {
  A a;
  a.f();
  B b;
  b.f();
}
```

produces this output:

```
A()
A::f()
A()
B()
B::f()
~B()
~A()
~A()
```

Explicitly invoking the base constructor

The base class's default constructor is automatically used:

```
class A {
public:
  A() { cerr << "A()\n"; }
  A(int x) { cerr << "A()(" << x << ")\n"; }
  ~A() { cerr << "~A()\n"; }
};

class B: public A {
public:
  B(int x=2) { cerr << "B(" << x << ")\n"; }
  ~B() { cerr << "~B()\n"; }
};

int main() {
  B b(3);
}
```

produces this output:

```
A()
B(3)
~B()
~A()
```

Explicitly invoking the base constructor

We can explicitly invoke a non-default constructor:

```
class A {
public:
  A() { cerr << "A()\n"; }
  A(int x) { cerr << "A()(" << x << ")\n"; }
  ~A() { cerr << "~A()\n"; }
};

class B: public A {
public:
  B(int x=2) : A(x) { cerr << "B(" << x << ")\n"; }
  ~B() { cerr << "~B()\n"; }
};

int main() {
  B b(3);
}
```

produces this output:

```
A(3)
B(3)
~B()
~A()
```

A simple class hierarchy

- A classic example is a class hierarchy based on shapes.
- This might be useful in a graphics library.
- The root of the class hierarchy is very simple:

```
class shape {
public:
  shape();          // Constructor
  ~shape(); // Destructor
  double perimeter() const { return 0; }
  double area() const { return 0; }
};
```

A simple example - derived classes

```
class polygon : public shape {
protected:
  int nsides;       // Number of sides
  double *lengths; // Lengths of each side
public:
  polygon(double width=1.0, double height=1.0);
  polygon(int n, double *len);
  ~polygon() { delete [] lengths; }
  double perimeter() const { // Override base class
    double p = 0.0;
    for (int i = 0; i < nsides; i++) p += lengths[i];
    return (p);
  }
};

class rectangle: public polygon {
  // Constructor just calls the base class
  rectangle(double width, double length)
    : polygon(width, length) { }

  // Override base class
  double area() const { return lengths[0] * lengths[1]; }
}
```

A simple example - derived classes

```
class ellipse: public shape {
protected:
  double semimajor, semiminor;
public:
  ellipse(double smj=1.0, double smn=1.0) {
    semimajor = smj;
    semiminor = smn;
  }
  double area() const {
    return PI * semimajor * semiminor;
  }
};

class circle : public ellipse {
public:
  circle(double r=1.0) : ellipse(r, r) { }
  // Don't override area(), but provide perimeter()
  double perimeter() const {
    return 2*PI*semimajor; // ``semimajor'' is protected
  }
};
```

A simple example - derived classes

```
int main() {
  circle c1(1);
  rectangle r1(1, 1);

  cout << c1.area() << " " << c1.perimeter() << endl;

  cout << r1.area() << " " << r1.perimeter() << endl;
}
```

This program would produce the output:

```
3.14159 6.28319
1 4
```

Assignment compatibility

C++ considers objects of a derived class to be assignment compatible with objects of their base class. This just makes a copy, skipping members that aren't part of the base class.

```
class A {
protected:
  int x;
//...
};

class B: public A {
  int y;
//...
};

int main() {
  B b;
  A a;
  a = b;   // OK, but 'y' is not copied!
}
```

Assignment compatibility

However, we can't to the reverse and assign an object from a base class to a derived class. This could leave derived class members in an undefined state.

```
class A {
protected:
  int x;
//...
};

class B: public A {
  int y;
//...
};

int main() {
  B b;
  A a;
  b = a;   // Not OK - undefined value for 'y'
}
```

Assignment compatibility with pointers

The same rules apply with pointers. We can assign the address of an object of a derived class to an pointer to the base class, but not the opposite.

```
class A {
  // ...
};
class B: public A {
  // ...
};

int main() {
  A a, *pa;
  B b, *pb;
  pa = &b; // OK
  pb = &a; // Error!
}
```

However, since we are assigning pointers, the objects in these assignments *are not modified*, as opposed to the case when objects are copied. They retain their full contents.

Polymorphism

The ability to use base class pointers to refer to any of several derived objects is a key part of *polymorphism*.

Exploiting polymorphism requires additional effort:

```
class A {
public:
  void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f() { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();    // Which f() does this call?
}
```

This call invokes the base class, A::f()!

Virtual functions

The solution is to declare functions `virtual`. This causes the compiler to call the "right" function when a call is made through a base class pointer:

```
class A {
public:
  virtual void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f() { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();    // Now this will call B::f()!
}
```

Virtual functions

A virtual function in the derived class will override the base class only if the type signatures match.

```
class A {
public:
  virtual void f() { cerr << "A::f()" << endl; }
};
class B: public A {
public:
  void f(int x) { cerr << "B::f()" << endl; }
};

int main() {
  B b;
  A *pa = &b; // OK

  pa->f();    // Now this will call A::f()!
}
```

As with overloading, changing only the return type introduces an ambiguity and will trigger a compile-time error.

Virtual function details

- You do not need to use the `virtual` keyword in the derived classes, but it is legal.
- If you explicitly use the scope operator, you can override the natural choice of function.

```
class A {
public: virtual void f() { cerr << "A::f()\n"; }
};

class B : public A {
public: virtual void f() { cerr << "B::f()\n"; }
};

int main() {
  A *pa = new B();

  pa->A::f();    // Explicity invokes the base class
  pa->f();       // Invokes B::f()
}
```

Virtual constructors or destructors

- You *cannot* declare a constructor virtual.
- You can, and often *should*, declare a destructor virtual:

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

class B : public A {
private:
  int *mem;
public:
  B(int n=10) { mem = new int[n]; }
  ~B() { cerr << "~B()\n"; delete [] mem; }
};

int main() {
  A *pa = new B(100);

  delete pa;
}
```

Abstract classes

- In C++ , an *abstract* type or class is related to the Java "interface" construct.
- An abstract class explicitly leaves one or more virtual member functions unimplemented or *pure*.
- You can't create an object based on an abstract class, but you can use it to define derived classes.
- You *can* create pointers and references to an abstract class.

Abstract class syntax

```
class A {
public:
 virtual int f() = 0; // ``Pure'' (i.e. not implemented)
 virtual int g() = 0;
};

class B : public A {
public:
  int f() { return 1; } // Overrides f()
}

class C : public B {
public:
  int g() { return 2; } // Overrides A::g()
}
```

Both A and B are abstract classes, and we cannot create objects of either type. Only C is a concrete class that can be created.

Of course, you can't call a pure virtual function. Any attempt to do so will probably generate a linker error.