COMP322 - Introduction to C++

Lecture 06 - Introduction to C++ classes

Dan Pomerantz

School of Computer Science

12 February 2013

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);
}
```

The `this` pointer

Alternatively, we can explicitly reference the implicit parameter using the keyword `this`.

In any non-static member function, `this` is a pointer to the object through which the member was invoked:

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

It is rarely *necessary* to use the `this` pointer explicitly, but it may occasionally help clarify the intent of your code.

Const member functions

If a member function is declared `const`, by placing the keyword after the parameter list, this means the member function will not make any changes to the implicit parameter:

```
class fcomplex {
  // ...
  float imagpart() const;
  // ...
};

float fcomplex::imagpart() const {
  return imag;
}
```

In comparison, consider a function to set the imaginary part:

```
void fcomplex::imagpart(float i) {
  imag = i;
}
```

Static member functions

If a member function is declared `static`, it is not called through a specific object, and the `this` pointer is undefined:

```
// from the class declaration:
  static float abs(const fcomplex &x);

// Here is the actual function definition. Note that we must
// not re-use use the static modifier here:
float fcomplex::abs(const fcomplex &x) {
  return sqrt(x.real * x.real + x.imag * x.imag);
}
```

These static functions are `not` invoked through a specific object:

```
  cout << fcomplex::abs(c) << endl;
```

Static data members

Unlike structure definitions, data objects in a class can also be declared `static`.

This creates a single data field whose storage and value is shared among all instances of the class.

These are the only objects in a class which may be initialized:

```
class Example {
private:
  int data1;
  string data2;
  static int data3 = 5;
  //...
};
```

Applications of static data

Here are a couple of applications for static data members:

- Parameters that are common to all class objects:

```
  static const int N_TABLE = 100; // Fixed
  static int udp_port = 1336; // Variable
```

- Data which is used for global accounting of resources:

```
class Example {
  static int use_count = 0;
};

Example() {
  if (use_count++ == 0) {
    // Get resources
  }
}

~Example() {
  if (--use_count == 0) {
    // Free resources, e.g.
  }
}
```

Default arguments

Sometimes it is useful to specify default values for function parameters. In this way we can simplify the most commonly used form of a function call.

```
void sort(int *array, bool descending = false);
```

We can call this function in any number of ways:

```
int numbers[] = { 7, 9, 28, 5, 1 };
sort(numbers); // Sort in ascending order
sort(numbers, true); // Sort in descending order
sort(numbers, false);
```

Default arguments may be specified for any C++ function.

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];
  }
  ~intStack() { // Destructor
    delete [] data;
  }

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

More complex destructors

```
class Symtable {
private:
  Symbol *head;
  Symbol *find(string name) {
    for (Symbol *sp = head; sp != NULL; sp = sp->link)
      if (sp->name == name) return sp;
    return NULL;
  }
public:
  Symtable() { head = NULL; } // Empty
  ~Symtable() {
    while (head != NULL) {    // Free the list
      Symbol *sp = head->link;
      delete head;
      head = sp;
    }
  }
  void set(string name, int value);
  int get(string name) {
    Symbol *sp = find(name);
    return (sp == NULL ? 0 : sp->value);
  }
};
```

Issues with constructors

Consider the symbol table example we just gave. What happens if we initialize a new object with an old one?

```
int main() {
  Symtable st1;
  st1.set("apple", 1);
  st1.set("peach", 2);

  Symtable st2 = st1; // Make a copy

  cout << "apple=" << st2.get("apple") << endl;
  st1.set("apple", 3);
  cout << "apple=" << st2.get("apple") << endl;
}
```

Perhaps surprisingly, this prints:

```
apple=1
apple=3
```

By default, initialization and assignment do a naïve copy.

Copy constructor

The solution to this problem is to provide a *copy constructor*, which copies the entire data structure.

The most generic form of copy constructor is:

```
class X {
  X(const X& src);
  //...
};
```

For our symbol table example, it would be:

```
  Symtable(const Symtable &src) {
    head = NULL;
    for (Symbol *sp = src.head; sp != NULL; sp = sp->link) {
      Symbol *np = new Symbol;
      np->name = sp->name;
      np->value = sp->value;
      np->link = head;
      head = np;
    }
  }
```

Inefficiencies may arise from privacy

Suppose we have two classes, `matrix` and `vector`, with private data and public accessor functions:

```
vector multiply(matrix& m, vector& v)
{
  vector r;

  for (int i=0; i < m.rows(); i++) {
    r.elem(i) = 0;
    for (int j=0; j < m.cols(); j++) {
       r.elem(i) += m.elem(i,j) * v.elem(j);
    }
  }
  return r;
}
```

All these function calls may be inefficient.

Friend functions

The `friend` keyword can be used to alter the normal rules about the visibility of class members.

We add this line to both the `matrix` and `vector` classes:

```
class vector {
  //...
  friend vector multiply(matrix &, vector &);
};
```

our function can now be written more efficiently:

```
vector multiply(matrix& m, vector& v)
{
  vector r;

  for (int i=0; i < m.n_rows; i++) {
    r.data[i] = 0;
    for (int j=0; j < m.n_cols; j++) {
       r.data[i] += m.data[i][j] * v.data[j];
    }
  }
  return r;
}
```

Friend classes

The same idea can apply to member functions or entire classes:

```
class X {
  //...
  void f();
};

class Y() {
  //...
  friend void X::f(); // Grant X::f() access to Y
};

class Z() {
  //...
  friend class X;    // Grant all of X access to Z
};
```

Dealing with scoping issues

A number of confusing situations can arise:

```
class X {
  int a, b;
public:
  X(int a, int b) {
    // how can I refer to the class members rather
    // than the parameters?

    X::a = a;     // One option
    this->b = b; // Another option
  }
```

A common convention is to apply some prefix to all private data members:

```
class X {
  int m_a, m_b;
public:
  X(int a, int b) {
    // no confusion now
  }
};
```

Dealing with scoping issues

Another situation arises when we wish to refer to a global object from within a class:

```
class vector {
  double *data;
  int length;
public:
  //...
  void pow(double y) {
    for (int i = 0; i < length; i++)
      data[i] = ::pow(data[i], x);
  }
}
```

Here we use the “unary” form of the scope resolution operator, which means “use the global version of `pow`”.

Nesting classes

A class can contain one or more classes:

```
class X {
  int x;
  class Y {
    // ...
  };
  class Z {
    // ...
  };
};
```

The enclosed classes are not visible outside of the scope of the enclosing class.

Initializing class members

When a class contains objects of another class, the constructors of the components can be called in the constructor of the containing class.

A new syntax is necessary to allow parameters to be passed to the constructor of objects allocated within the structure.

```
class matrix {
public:
  matrix(int rows, int cols) {
    // ...
  }
};

class something {
   matrix m1;
   matrix m2;
public:
   something(int n, int m)
     : m1(n, m), m2(n, m) {
    // initialize other members of something
   }
};
```