COMP322 - Introduction to C++

Winter 2011

Lecture 06 - Classes & Overloading

Milena Scaccia

School of Computer Science
McGill University

February 8, 2011

Recall from last class

- Class Definition (Declaration and Implementation)
  - Constructor
  - Destructor
  - Copy Constructor
  - Member Functions

The `this` pointer

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

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

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:

```
complex complex::mul(const complex &y) {
  return complex(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.

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 complex &x);

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

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

```
  cout << complex::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 data members 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 = 1337; // Variable
```

- Data which is used for global accounting of resources:

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

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

~Example() {
  if (--lock == 0) {
    // Free resources
  }
}
```

Default arguments

We saw this for constructors; it also applies for member functions.

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.

Scoping issues

An issue arises when we wish to refer to a global object from within a class:

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

int count = 500;

class X
{
private:
  int count;
public:
  X(int a){
    count = a;
  }
  int getGlobalCount(){
    return ::count; //Here we use the ``unary'' form of the
                    //scope resolution operator, which means
                    //``use the global version of count''.
  }
};
```

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 &);
};
class matrix {
  //...
  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];
    }
  }
```

Friend classes

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

class Hound() {
  //...
  friend class Fox;    // Grant all of Fox access to Hound
};

class Poodle() {
  //...
  friend void Fox::f(); // Grant Fox::f() access to Poodle
};
```

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. Nested classes usually act as “helper classes” to 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
   }
};
```

Overloading

What is overloading?

http://www.codercaste.com/2011/01/09/what-is-function-overloading-how-to-use-it-to-write-better-code/

What is overloading?

Overloading refers to the programmer's ability to assign multiple new meanings to existing functions or operators.

- Function Overloading
  - Overloaded functions must have different argument lists, so the compiler can select the correct function.
- Operator Overloading
  - C++ allows us to overload operators as well. Operators get additional "power". Example: we can redefine the meaning of '+' for a new class.

Overloading functions

```
//First print
void print(int x, int radix = 10) {
  cout << setbase(radix) << x << endl;
}
//Second print
void print(double x, int precision = 6) {
  cout << setprecision(precision) << x << endl;
}

int main() {
  print(10,16); /* Calls the first print. If no second parameter
                   is specified for the base, then it takes the
                   default value of 10 */
  print(3.14159); /* Calls the second print. If no second
                     parameter is specified for precision,
                     then it takes the default value of 6 */
}
```

Restrictions on overloading

- The functions cannot differ by return type alone!

```
class example {
  double getval();
  int getval();     // Ambiguous!!
}
```

- Pointers and arrays are identical in argument lists, and the first array dimension is not significant:

```
double mean(int array[10]);
double mean(int array[20]); // Ambiguous
double mean(int *array);    // Also ambiguous
```

- Typedef names are not distinct

```
typedef int Int;
void f(int i);
void f(Int i);  // Ambiguous
```