UMassAmherst

CS197c: Programming in C++

Lecture 3

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197c/index.html

Syllabus

- ~~Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs~~
- ~~Lecture 2 : C libraries~~
- Lecture 3 : Classes in C++, C++ I/O
- Lecture 4 : Memory & Pointers
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

C++ Classes

- Conceptually similar to Java classes
- Mostly syntax differences:

```
class Foo {
   // member declarations here
};

// method definitions out here
```

C++ Headers and Classes

- Usually only function declarations, type defs

Foo.h:

```
class Foo {
private:
    int id;
    string name;
public:
    Foo(string n);                  //constructor
    ~Foo();                         //destructor
    string getName() { return name; }  //inlined!
};
```

C++ Headers and Classes

- Source contains function/method definitions:

Foo.cpp:

```
#include “Foo.h”

Foo::Foo(string n) { name = n; }

Foo::~Foo() {}

string Foo::getName() { return name; } // or define it here
```

Inheritance

- Sharing an interface among a group of related classes
- One class is the base class
- Remaining classes are derived classes (or subclasses) of the base class and inherit the base class's interface

Differences from Java

- C++ allows multiple inheritance
  - In reality: *Hideous* idea. Not needed
  - Java goes too far the other way
  - Newer languages use *mixins*

- No interfaces explicitly
  - Pure virtual (abstract) classes take this role

Syntax for inheritance

- To derive a class from another class
  - class derivedclass : access baseClassname{…}
  - Example :
    - Class Rectangle : public GraphicalObject { … }

Access issues in inheritance

- When base class is inherited public
  - class Rectangle : public Graph {..}
  - public in base class => public in derived class
  - protected in base class => protected in derived class
- When base class inherited as protected
  - Class circle : protected Oval { …}
  - Public in base class => protected in derived
  - Protected in base class => protected in derived
- When base class inherited as private
  - Class square : private Rectangle { … }
  - Public in base class => private in derived class
  - Protected in base class => private in derived class

Overriding/dominated inherited members

- If Y is a subclass of X, then Y object inherits all the protected and public member data
- You might want to define a local version of an inherited member

```
class X {
    public :
        void f() { cout<<“exec”;}
 };
```

```
class Y : public X {
    public :
        void f() {cout<<“no”;}
};
```

Parent Constructors

```
class X {
   public : ~X()
   { cout<<"X:X()";}
  };

class Y: public X {
  public : ~Y()
   { cout<<"Y:Y()";}
  };

class Z : public Y {
   public: ~Z()
   { cout<<"Z:Z()"; }
};
```

Call order:

```
X:X()
Y:Y()
Z:Z()
```

Parent Destructors

```
class X {
   public : ~X()
   { cout<<"X:X()";}
  };

class Y: public X {
  public : ~Y()
   { cout<<"Y:Y()";}
  };

class Z : public Y {
   public: ~Z()
   { cout<<"Z:Z()"; }
};
```

Call order:

Z:Z()
Y:Y()
X:X()

Virtual methods

```
class X {
  public :
    void f() {
      cout <<“X:f()” << endl;
    }
};
```

```
class Y : class X {
    public :
      void f() {
        cout<<“Y:f()”<<endl;
    }
};
```

```
main () {
   X x;
   Y y;
   X *p = &x;
   p -> f(); p = &y; p->f();
}
```

How do you only declare a method?

```
class X {
   public : virtual void f() { cout <<"X:f()";}
  };
```

what is the output now?

The methods signature in the derived class must exactly match the method's signature in the base class

You can never alter the return type of a virtual method

Pure Virtual Classes

- Predecessor to abstract classes in Java
- Cannot be instantiated
- Definition is incomplete

```
class X {
  public :
    virtual void f() = 0;
};
```

(see examples virtual1.cpp and virtual2.cpp)

Inner Classes

- Example: iterator

```
class X {
   public:   // ← what if private?
        class iterator {
              // definition in here
        };
};
```

Overloading Operators : Basics

- What : redefine operators on a per-class basis
- Why : C++ likes to make objects look like primitives
- Which operators? Almost all of them:
  - Relational (==, != , < , >)
  - Arithmetic/Bitwise/Logical (+, -, /,*,--, ++, ||, &&, |, &)
  - Assignment (=, += , -=)
  - Unary (+, -)
  - I/O (<<, >>)
  - Access ((),[], ->, ->*, , (the comma))
  - Memory (new , delete, new[], delete[])

Overloading operators: Basics

- Off limits:
  - sizeof
  - membership (., .*)
  - ::
  - ?:
  - typeid, const_cast, dynamic_cast, static_cast (reflection)

Overloading example

```
Person p1, p2;

cout<<p1;
```

What is really going on here?

Overloading example

```
Person p1, p2;

cout<<p1;
```

What is really going on here?

`cout<<p1` → `operator<<(cout,p1)`

More about overloading

- Operator has different definitions depending on which object it is operating on

- Operators should take care to release memory if they are replacing one piece of dynamic memory with another

Overloaded operator syntax

```
T& operator=(const T&);
```

- Return type is reference to an object of the same class T
- Thus function must return the object that is being assigned
- How do you implement this?

Example of overloaded arithmetic operator

- Conventional way of doing it

```
Ratio product (Ratio x, Ratio y)
{
    Ratio z(x.num*y.num, x.den*y.den);
    return z;
}
    call :  Z = product(x,y);
```

- Operator overloading

```
Ratio operator*(Ratio x, Ratio y)
{
    Ratio z(x.num*y.num, x.den*y.den);
    return z;
}
   call :  Z = x * y;
```

Non-member function

- Previous example :

  product is a non member function (vs member)

- Product requires access to private members. Make use of friend modifier

```
class Ratio
{
    friend Ratio operator* (const Ratio &, const Ratio &);
 }
```

Member vs. Non-member

- Member
  - `Ratio &operator*(Ratio &other)` ...
  - Modifies object in place

- Non-Member
  - `Ratio operator*(Ratio x, Ratio y)`...
  - Takes 2 arguments and returns some `Ratio`

Post and Pre increment

- (Post increment)++
  - Ratio Ratio::operator++(int)
  - How would you implement it?

- ++ (Pre increment)
  - Ratio Ratio::operator++()
  - How is it different from post increment?
  - Dummy argument (int)

Subscript operator []

- How would you implement in the Ratio class
- Say for a ratio 22/7 1st index is the numerator and 2nd index is the denominator
  - e.g. `r[0]` = 22, `r[1]` = 7

- How might you implement this?

LET'S TAKE A LOOK

(blobs.cpp)

Namespaces: Back to C++ code

- Hello world !!

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

int main(int argc, char * argv[]) {
     cout<<“Hello world !”<< endl;
      return 0;
}
```

Specifies use of a given namespace

Namespaces

- Similar to Java's package scheme
- Used to avoid name collisions

```
namespace cs197c {
    Person teacher;
    Person[] students;
...
}

cs197c::teacher = new Person();

using cs197c;
teacher = new Person();
```

w/o using

with using

I/O of C++

- std::cin is standard input
- std::cout is standard output
- std::cerr is standard error

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

cout << "Beam me up, Scotty.\t" << 5 << 3.425654 << endl;
```

contains declarations

puts in scope

produces newline

calls can be chained, different operands

I/O in C++

- Standard input (cin)

```
int value, input;
cin >> value >> input;
```

- Standard output (cout)

```
cout << "Hello world!" << endl;
```

- Standard error (cerr)
  - Same behavior as cout, but separate stream

I/O in C++

- Available stream implementations:
  - <iostream>: cout, cin, cerr
  - <fstream> is the equivalent, but for files
  - <stringstream> transfers stream functionality to a string.

- Let's get into some detail...

Setting flags on a stream

- Flags affect the behavior of streams
  - Example: `cout.setf(ios_base::showpos);`

  will show plus signs in front of positive numbers

- Types of flags
  - numerical formatting
  - filling spaces
  - etc…

Manipulators

- Instead of using setf() all day long...
- Convenience functions for the streams
  - endl
  - flush
  - hex, dec, oct
  - fixed, scientific...many more

Example (printing in hexadecimal):

```
int i = 45;
cout << hex << i << flush;
```

Raw Output

- Say you just want to put 8 bytes on the stream...

- write method to the rescue!

```
ostream &write( const char *s,
                streamsize n);
```

- streamsize is really an integer
- Paraphrase: “write n bytes starting at s”

Raw Output (cont'd)

- Example:

```
double d = 2.718281828;

cout.write((const char *) &d, sizeof(double));
```

- Also:

```
char c = 'A';

cout.put( c );
```

Reading input

- Simple line-at-a-time interface: getline

```
fstream iofile("input.txt",ios::in);
if(!iofile.fail()) {
    while(!iofile.eof()) {
      char data[MAX];
      iofile.getline(data,MAX);
      // Manipulate 'data' variable here
    }
}
iofile.close();
```

- cin is the preferred method of input processing in C++

Reading input (cont'd)

- Use cin:

```
for (cin >> num; cin.good(); cin >> num) {
   sum += num;
}
// cin failed to read since the loop ended.
// Need to clear the stream to read more...

cin.clear();
cin >> num;
```

Raw Input

- Similar to cout's raw methods:
  - get

- Single-byte input

```
char c;
cout.get( c );
```

- Multibyte input

```
long l;
cout.get((char *) &l, sizeof(long));
```

Programming Assignment 2

- Implement a Linked List of integers

- Provided
  - Makefile (won't compile yet)
  - Header file (`LinkedList.h`)
  - PDF of assignment

- You provide
  - `LinkedList.cpp`

Programming Assignment 2

- Will exercise knowledge of defining classes
- Also needs pointers (next week)
- Examples are available
- Each individual method is *not* complicated

Next class

Memory Management

&

Pointers