Programming C++ Lecture 3

Howest, Fall 2013
Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

Interface vs. Implementation

- Interface defines and standardizes way to interact – says what services are available and how to request them.
- Implementation – how services are carried out.
- Separate them: interface = *.h, implementation = *.cpp
- *.h includes function prototypes and data members
- *.cpp defines member functions (use **::** binary scope resolution operator to tie functions to class definition)

Abstraction and Encapsulation

- Abstraction = creation of a well-defined interface for object
- Encapsulation = keep implementation details private
  - Data members and helper functions private
- Promotes software reusability
- Can change class data representation and/or implementation without changing code that uses class
- Good software engineering

const Objects

- const Course courseOne("Intro to CS");
- const objects can *only* call const member functions – even if function does not modify object
- Member function that is const cannot modify data members
- Member function that is const cannot call non-const member functions
- Constructors and destructors cannot be const

Member Initializer Syntax

```
#ifndef INCREMENT_H
#define INCREMENT_H
class Increment {
public:
    Increment(int c = 0, int i = 1);
    void addIncrement() {
        count += increment;
    }
    void print() const;
private:
    int count;
    const int increment;
};
#endif
```

```
#include <iostream>
using namespace std;
Increment::Increment(int c, int i)
    : count(c), increment(i) {
    //empty body
}
void Increment::print() const
   cout << "count = " << count <<
     ", increment = " << increment <<
     endl;
}
```

Member Initializer Syntax

- All data members can be initialized with this
- const data members and data members that are references **must** be initialized with this
- After constructor's parameter list and before left brace, put ":" then *dataMemberName(initialValue)*
- Member initializer list executes before constructor body
- Member objects either initialized with member initializer or member object's default constructor

Static Data Members

- Classes have only 1 copy of static data members whereas object instances each have their own copy of non-static data members
  - Object instance size determined by non-static members
  - Static member initialization
    - Initialized only *once*, only static members can be initialized in class definition (.h)
    - Static members with fundamental types initialized by default to 0.

Static and Scope

- We now have another scope: class scope
  - Inside class scope, data members accessible by all member functions
  - Outside, public data members referenced through object handle
  - Static data members have class scope
- Access using *className::staticDataMemberName* (can use a particular object instance name if any exist)

A Class

```
//Course.h
#include <string>
using namespace std;
class  Course {
public:
     Course( string name );
     void setCourseName(
               string name);
     string getCourseName() const;
     static int getCount();
private:
     string courseName;
     static int count;
};
```

```
//Course.cpp
#include "Course.h"
int Course::count = 0;  //no static here!
int Course::getCount() {//no static here!
     return count;
}
Course::Course( string name ) {
   setCourseName(name);
   count++;
}
void Course::setCourseName(string name) {
   courseName = name;
}
string Course::getCourseName() const {
   return courseName;
}
```

Using static Data Members

- Course::getCount(); //don't need objects of class to exist to access static data member
- Course *myCourse = new Course("CS105 C++");
- myCourse->getCount(); //but you can use them if they exist

this

- Every object has access to its own address through pointer called *this* (C++ keyword)
- **this** pointer passed by the compiler as implicit argument to each object's non-static member functions
- **this** pointer's type is const pointer to type of class (i.e. Course * const)
- In Course class, accessing data member "courseName" implicitly uses **this**. Or: this->courseName or (*this).courseName

Tricky Things with Objects

- What happens if you…
  - Set one object equal to another?

Course myC++Course(“CS105: C++ Programming”);

Course myFavoriteCourse = myC++Course;

  - Pass an object to a method as a parameter?

void myMethod(Course myCourse);

Tricky Things with Objects

- What happens if you…
  - Set one object equal to another?
    - Object =
  - Pass an object to a method as a parameter?
    - Object copy
- Both assignment operator and object copy are provided by default, and do member-wise assignment
  - However, if you have pointer member variables, you have to write your own!

Object Copies

- When objects are passed to functions or returned, they are by default passed by value; a copy needs to be created
- How: copy constructor (default provided by compiler) that does member-wise copying of object (assign each member variable)

```
Course( const Course &courseToCopy ) {   //why "&"?
    courseName = courseToCopy.courseName;
}
```

Object =

- When one object is set to equal another object

```
Course myFavoriteCourse = myC++Course;  //example
```

- How: object assignment method (default provided by compiler) that does member-wise assignment of each member variable

```
Course& operator= (Course const &otherCourse) {
        courseName = otherCourse.courseName;
}
```

Member Initializer Example

```
Employee::Employee(const char* const first, const char* const last,
    const Date &dateOfBirth, const Date &dateOfHire)
    : birthDate( dateOfBirth ),
      hireDate( dateOfHire ) {
    /*above initializers each call
    copy constructor of Date class*/
    //here use first & last to initialize members
    .....
}
```

```
//from Employee.h
class Employee {
private:
    char firstName[25];
    char lastName[25];
    const Date birthDate;
    const Date hireDate;
};
```

Why References, Why Pointers?

- References
  - invoke functions implicitly, like copy constructor, assignment operator, other overloaded operators
  - Can pass large objects without passing address
  - Don't have to use pointer semantics
- Pointers
  - Good for dynamic memory management
  - Ease of pointer arithmetic
  - Provides level of indirection in memory

Tidbits about Classes

- Copy constructor and overloaded assignment operator (=) have to be provided when you have member variables that are dynamically allocated
  - Destructor also should be provided
- To prevent one object from being assigned to another, declare assignment operator as private member function.
- To prevent objects from being copied, make both overloaded assignment operator and copy constructor private.