Programming C++ Lecture 3

Howest, Fall 2014

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)

A Class

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

```
//Course.cpp
#include “Course.h”

Course::Course( string name ) {
   setCourseName(name);
}
void Course::setCourseName(string name) {
   courseName = name;
}
string Course::getCourseName() {
   return courseName;
}
```

Test Program

```
//test program can be in another file – testCourse.cpp
#include <iostream>
using namespace std;
#include “Course.h”
int main() {
        Course myCourse1( “CS105: Programming in C++” );
        string nameOfCourse;
        cout << “Enter course name: “;
        getline( cin, nameOfCourse );
        Course myCourse2(nameOfCourse);
        cout << myCourse1.getCourseName();
        cout << endl;
        cout << myCourse2.getCourseName();
        cout << endl;
        return 0;
}
```

Preprocessor Wrapper

```
//Course.h
#include <string>
using namespace std;
#ifndef COURSE_H
#define COURSE_H
class  Course {
public:
      Course( string name );
      void setCourseName(
                string name);
      string getCourseName();
private:
      string courseName;
};
#endif
```

- Prevent header code from being included into same source code file more than once
- Use uppercase, usually file name with "." replaced by "_"

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

Constructors with Defaults

- Constructors can have default values.
- Specify them in .h

```
Course c1;

Course c2(54520);
```

- If multiple parameters, they are omitted right to left

```
//Course.h
#include <string>
using namespace std;
#ifndef COURSE_H
#define COURSE_H
class  Course {
public:
     Course( int num = 50000 );
     void setUniqueNum(
              string num);
     string getUniqueNum ( );
private:
     int uniqueNum;
};
#endif
```

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

A Class

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

```
//Course.cpp
#include “Course.h”

Course::Course( string name ) {
   setCourseName(name);
}
void Course::setCourseName(string name) {
   courseName = name;
}
string Course::getCourseName() const {
   return courseName;
}
```

const Objects

Object	Member Function	Allowed?
non-const	non-const	?
non-const	const	?
const	non-const	?
const	const	?

const Objects

Object	Member Function	Allowed?
non-const	non-const	YES
non-const	const	YES
const	non-const	NO
const	const	YES

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 of a class are 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( “C++ Programming” );

Course myGPUCourse(“GPU Programming with C++”);

Course myC++Course = myGPUCourse;
```

  - 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 myC++Course = myGPUCourse;  //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;
        return *this;
}
```

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.

Inheritance

Inheritance

- Software reuse – inherit a class's data and behaviors and enhance with new capabilities.
- Existing class = base class, inheriting class = derived class (no super/ subclass like Java)
- Derived class is more specialized than base class. Object instances of derived class are also object of base class (All cars are vehicles, but not all vehicles are cars.)
- There can be multiple levels of inheritance.

Inheritance Details

- class Circle : public Shape
  - What is base, what is derived here?
- Default = public inheritance (base member variables retain same access level in derived class), but there are other types
- When redefine something in derived class, use *<baseclassName>::member* to access base class's version.

Inheritance and Member Variables

- Derived class has all attributes of base class.
  - Derived class can access non-private members of base class.
  - **protected** members of base class are accessible to members and friends of any derived classes.
  - Derived does not inherit constructor or destructor of base.
  - Derived class can re-define base-class member functions for its own purposes, customizing base class behaviors.
- Size of derived class = non-static data members of derived class + non-static data members of base class (even if private)

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    ~Member();

    string getName() const;
    void setName(string name);
    void print() const;
private:
    string myName;
};
```

Derived Class Example

```
#include “Member.h”
class Employee : public Member {
public:
      Employee(string name, double money);
      Employee( Employee const &);
      Employee& operator= (Employee const &);
      ~Employee ();

      double getSalary() const;
      void setSalary(double money);
      void print() const;
private:
      double salary;
};
```

Employee Constructor

```
#include “Employee.h”
Employee::Employee( string name, double money )
    : Member(name)   //base class initializer syntax
{
    salary = money;
}
```

- C++ requires derived class constructor to call base class constructor to initialize inherited base class data members (if not explicit, default constructor would be called).

Employee’s print Function

```
void Employee::print() const
{
    cout << “Employee: “;
    Member::print();  //prints name from base class
    cout << “\nsalary: “ << getSalary() << endl;
}
```

Constructor/Destructor Order

- When we instantiate a derived class:
  1. Base class's member object constructors execute (if they exist)
  2. Base class constructor executes
  3. Derived class's member object constructors execute
  4. Derived class constructor executes
- Destructors called in reverse order.
- Base class constructors, destructors and overloaded assignment operators are not inherited by derived classes. However derived class can call base class's version of these.

Encapsulation

- Given a derived class can directly access and modify protected data members of base class, should base class member variables be protected? Or private?

Encapsulation

- Given a derived class can directly access and modify protected data members of base class, should base class member variables be protected? Or private?
  - + No overhead of function call in derived class
  - – Direct modification does not allow for error checking.
  - – If base class member variables names change, we have to change all derived classes use of them.

Kinds of Inheritance

Base Class Access (down)	Public inheritance	Protected inheritance	Private inheritance
public	public	protected	private
protected	protected	protected	private
private	private	private	private