Programming C++ Lecture 2

Howest, Fall 2012
Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

Function Templates

- We can do function overloading

```
int boxVolume(int side) {
    return side * side * side;
}
double boxVolume(double side) {
    return side * side * side;
}
```

- Why define 2 functions that look identical, but have different types?
- Overloading that is more compact and convenient = function templates. Only write it once!

Function Templates

- Template

```
template <class T>  //or template <typename T>
T boxVolume(T side) {
    return side * side * side;
}
```

- C++ compiler automatically generates separate function template specializations for each type the function is called with.
- T is placeholder for actual data type
- int result = boxVolume(3); double result = boxVolume(6.2);

Classes!

- Classes encapsulate objects
- Member variables
- Member functions
  - Getter and setter functions
  - Constructors and destructors
- Access specifiers
  - Public versus private

A Class

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

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

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

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

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

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

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

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

A Class

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

```
int main() {
    string nameOfCourse;
    Course myCourse;
    cout << myCourse.getCourseName();
    cout << endl;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    myCourse.setCourseName(
                    nameOfCourse);
    cout << myCourse.getCourseName();
    cout << endl;
    return 0;
}
```

Objects with Pointers, References

```
Course myCourse;  myCourse.getCourseName();

Course &courseRef = myCourse;  courseRef.getCourseName();

Course *coursePtr = &myCourse;  coursePtr->getCourseName();

Course* myCourse1 = new Course( "C++Programming" );

myCourse1->getCourseName();
```

- Inside your class, don't return a reference or pointer to private data members! BAD style.

A Class with a Constructor

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

```
int main() {
    Course myCourse1("C++
        Programming" );
    string nameOfCourse;
    cout << "Enter course name: ";
    getline( cin, nameOfCourse );
    Course myCourse2(nameOfCourse);
    cout << myCourse1.getCourseName();
    cout << endl;
    cout << myCourse2.getCourseName();
    cout << endl;
    return 0;
}
```

Constructors

- Special class methods with class name
- Cannot return anything
- Initialize state of object when created
- Usually public
- Called implicitly for every object creation
- Default constructor: no parameters, automatically created by compiler if no constructor
  - Calls default constructor of object data members of class

Destructors

- Similar to constructor: tilde followed by class name (~Course( ) { … })
- Receives no parameters, cannot return value.
- Only 1 destructor and must be public.
- Called implicitly when object destroyed (goes out of scope)
  - Does NOT release object's memory, but performs housekeeping.
- Compiler implicitly creates empty one if none exists.

Destructors

- When in particular are they useful?
- To see order in which constructors/destructors called, see

http://users.elis.ugent.be/~jsartor/howest/constructorDestructor.cpp

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
```

Compilation and Linking

- Compiler uses included interface .h files to compile .cpp file into object code
  - Course.h + testCourse.cpp -> testCourse.o
  - Course.h + Course.cpp -> Course.o
- Linker takes object code of testCourse.cpp and Course.cpp and STL and puts it together into an executable.
  - testCourse.o + Course.o + stl.o -> testC.exe

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 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.
- 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

Object =

```
Course myC++Course( “CS105: C++ Programming” );
Course myFavoriteCourse = myC++Course;
cout << myFavoriteCourse.getCourseName() << endl;
```

- Memberwise assignment of objects
- Assign each data member to corresponding data member
- Course& operator= (Course const &);

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)

Course(const Course &courseToCopy); //why "&"?

- How would we pass an object by reference??

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

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

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName(); //what does this print?
7. mPtr->print(); //and this?

Instantiating Objects Example

```
1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName();  //Jill
7. mPtr->print();                    //Jill
```

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary(); //result?
8. ePtr->print(); //what function does this call?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary(); //Jack 65000
8. ePtr->print(); //Employee.print which calls Member.print

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1; //is this ok? Base class pointer to derived class?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1; //Yes, valid; all Employees are Members
8. ePtr = &m1; //this valid? Derived class pointer to base class?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1; //Yes, valid; all Employees are Members
8. ePtr = &m1; //No, not all Members are Employees;
   //compiler error

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1; //yes, this is valid; all Employees are Members
7. cout << mPtr->getName(); //what does this print?
8. cout << mPtr->getSalary(); //this ok?
9. mPtr->print(); //what function does this call?

Instantiating Objects Example

```
1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;
7. cout << mPtr->getName();          //Jack
8. cout << mPtr->getSalary();        //compiler error
9. mPtr->print();                    //calls Member’s print: Jack
```

Operator Overloading

- Think of "+" – does different things based on the types that it is applied to.
- Can we apply "+" to objects – like the Date class?
- Can achieve same thing with function calls, but operator notation is often clearer and more familiar (in C++).
- Can't create new operators, but can overload existing ones so they can be used with user-defined types.

See Example

Overloading + does not implicitly overload +=

- Instead of myDate.add(otherDate), we do myDate + otherDate.
- Write a non-static member function or global function with function name as "**operator**<*symbol*>" (aka operator+)
- One argument of operator function must be user-defined (can't re-define meaning of operators for fundamental types)
- http://www.cs.utexas.edu/users/jbsartor/cs105/code/ArrayExample/

Global vs Member Functions

- Difference: member functions already have "this" as an argument implicitly, global has to take another parameter.
- "()" "[]" "->" or assignment has to be member function
- Leftmost operand
  - For member function: must be object (or reference to object) of operator's class.
  - Global function used when it is not user-defined object (overloading << and >> require left operand to be ostream& and istream&)
- Global operators can be made **friend** of class if needed.
- Global functions enable commutative operations

Overloading Restrictions

- To use an operator with class, operator ***must*** be overloaded with 3 exceptions (but these can be overloaded too):
  - Assignment (=) does member-wise assignment for objects. (overload for classes with pointer members)
  - The "&" and "," operators may be used with objects without overloading
- The following cannot be changed for operators:
  - Precedence
  - Associativity (left-to-right or right-to-left)
  - Arity (how many operands)
- Can't overload: "." ".*" "::" "?:"

Operators: Converting between Types

- Conversion constructor is a single-argument constructor that turns objects of other types (including fundamental types) into objects of a particular class.
- Conversion/cast operator converts object into object of another class or to a fundamental type
  - A::operator char *() const; //convert object of type A into char* object. "const" above means does not modify original object
  - A myA;
  - static_cast<char *>(myA); //CALLS myA.operator char* ()
- Conversion functions can be called implicitly by the compiler

Makefile

```
CC = g++ -Wall –Werror –g
testC: testCourse.o Course.o
    ${CC} –o testC testCourse.o Course.o
testCourse.o: testCourse.cpp Course.h
    ${CC} –c testCourse.cpp
Course.o: Course.cpp Course.h
    ${CC} –c Course.cpp
clean:
    rm –rf *.o
```

- Reusability!
- Written in different language
  - # denotes comments
- List of
  rule_name : *dependencies*
  <tab> *command*
- Can do "make" with any rule, or by itself for 1st rule

Example

- Look at
- http://www.cs.utexas.edu/users/jbsartor/cs105/code/
  - Makefile
  - constDest.h
  - constDest.cpp
  - testConstDest.cpp