Programming C++
Lecture 4

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

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.

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.

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

Templates

- Function and class templates – you specify with a single code segment an entire range of related (overloaded) functions or classes (function or class-template specializations).
- Generic programming!
- Templates are stencils of pretty shapes
- Template specializations are tracings we make of stencils – same shape but maybe different colors.

Class Stack Template Example

```
//Stack.h
template< typename T >
class Stack {
public:
    Stack(int = 10);
    ~Stack() { delete [] stackPtr; }
    bool push( const T & ); //push element
    bool pop( T & ); //pop element
    bool isEmpty() const {
        return top == -1;
    }
    bool isFull() const {
        return top == (size -1);
    }
private:
    int size;
    int top;
    T *stackPtr;
};
```

```
template< typename T >
Stack< T >::Stack(int s)    //constructor
    : size( s > 0 ? s : 10 ),
      top( -1 ),
      stackPtr( new T[size] ) { }
template< typename T >
bool Stack< T >::push(const T &pushValue) {
    if (!isFull()) {
        stackPtr[++top] = pushValue;
        return true;
    } return false;
}
template< typename T >
bool Stack< T >::pop(T &popValue) {
    if (!isEmpty()) {
        popValue = stackPtr[top--];
        return true;
    } return false;
}
```

Test Stack

```
#include <iostream>
using namespace std;
#include "Stack.h"
int main() {
    Stack< double > doubleStack(5);
    double doubVal = 1.1;
    while (doubleStack.push(doubVal))
        doubVal += 1.1;
    while (doubleStack.pop(doubVal))
        cout << doubVal << ' ';

    Stack< int > intStack;  //default size
    int intVal = 1;
    while (intStack.push(intVal ))
        intVal ++;
    while (intStack.pop(intVal ))
        cout << intVal << ' ';
    return 0;
}
```

- Testing double stack vs. int stack is very similar pattern.
- You could create a template function to test your template class!

Specifics of Templates

- You can have nontype template parameters too
  - template< typename T, int elements >   //compile time constant
  - Stack< double, 100 > mostRecentSalesFigures;
  - .h could contain member: T stackHolder[elements];
- Type parameter can specify default type
  - template< typename T = string >
  - Stack<> jobDescriptions;
- Explicit specialization for a particular type
  - template<>
  - class Stack< Employee > { … };
- Make sure overloaded operators used in template class are implemented if used with user-defined type!
  - Our Stack requires user-defined type to have default constructor and assignment operator.

Next: Kinda Important

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://users.elis.ugent.be/~jsartor/howest/arrayClassExampleCode/

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: "." ".*" "::" "?:"

Friends of Objects

- Classes sometimes need friends.
- Friends are defined outside the class's scope, but are allowed to access non-public (and public) data members.
  - Friend functions – see example
  - Friend classes: friend class ClassTwo; (if placed inside ClassOne definition, all ClassTwo is friend of ClassOne)
- Class must explicitly declare who its friends are.

Friend Example

```
#include <iostream>
using namespace std;
class Count {
    friend void setX(Count &, int);
public:
    Count() : x(0) { }
    void print() const {
        cout << x << endl;
    }
private:
    int x;
};
```

```
void setX( Count &c, int val ) {
    c.x = val; //accesses private data!
}
int main() {
    Count counter;
    counter.print();
    setX(counter, 8);
    counter.print();
    return 0;
}
```

Not As Important

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

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 1^{st} rule

Example

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