Programming C++ Lecture 5

Howest, Fall 2014
Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

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.

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);

Class Stack Template Example

```
//Stack.h
template< typename T >
class Stack {
public:
    Stack(int = 10);
    ~Stack() { delete [] stackPtr; }
//need operator= and copy constructor also
    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 > { … };

More Details

- Because a compiler compiles template classes on demand, it requires the definition (usual .cpp) to be in the same file as the declaration (usual .h).
- http://www.cplusplus.com/doc/tutorial/templates/
- Make sure 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.

Friends

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;
}
```

Makefile

Compiling with g++

- g++ basic.cpp (creates “a.out” executable)
- g++ -o program basic.cpp (“program” is executable)

  ./program

- Flags that are good practice
  - g++ -Wall -o program basic.cpp (print all warnings)
  - g++ **-Wall -Werror** -o program basic.cpp (treat warnings as compilation errors)

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

Compilation and Linking

- Compiler uses included interface .h files to compile .cpp file into object code
  - g++ -Wall -Werror –c testCourse.cpp DOES Course.h + testCourse.cpp -> testCourse.o
  - g++ -Wall -Werror –c Course.cpp DOES 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.
  - g++ -Wall –Werror –o testC testCourse.o Course.o DOES testCourse.o + Course.o + stl.o -> testC.exe

Example

- For example Makefile, see
  - http://users.elis.ugent.be/~jsartor/howest/MemberAndDate/

Operator Overloading

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?
  - Instead of myDate.add(otherDate), we do myDate + otherDate.
- 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.

Operator Overloading

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/ArrayClass/

Global vs Member Functions

- Leftmost operand
  - For member function: must be object (or reference to object) of operator's class.
    - myDate + otherDate; => myDate.operator+(Date &otherDate)
    - Defined inside Date class
  - Global function used when it is not user-defined object (overloading << and >> require left operand to be ostream& and istream&)
    - cout << myDate; => operator<<(ostream &cout, Date &myDate)
    - Defined outside Date class

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
- 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

Why References, Why Pointers?

- References
  - invoke functions implicitly, like copy constructor, assignment operator, **other overloaded operator**
  - 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

Overloading ++ and --

- Prefix (++x)
  - Member function: Array &operator++();
  - Global: Array &operator++(Array &);
  - Returns incremented reference to object (lvalue)
- Postfix (x++)
  - Member function: Array operator++(int);
  - Global: Array operator++(Array &, int);
  - Use dummy int (0) to distinguish prefix from postfix
  - myA++ translates to myA.operator++(0)
  - Returns temp object that contains original value before increment (rvalue instead of lvalue)
    - Save: Array temp = *this. Then do your increment, then return (unmodified) temp.

Overloaded Function Call Operator

- Use ( ) operator
- String operator()(int index, int subLength) const;
  - Returns a substring for class String starting at index, of length subLength
  - String s1("Hello"); cout << s1(1,3) << endl;

Pointers to Functions

Function Pointers

- A pointer to a function contains the address of the function in memory
- Name of a function is actually starting address in memory of the code (like array name!)
- Function pointers can be
  - Passed to and returned from functions
  - Stored in arrays
  - Used to call the underlying function

Function Pointers

- See http://users.elis.ugent.be/~jsartor/howest/functionPointers.cpp
- See http://users.elis.ugent.be/~jsartor/howest/arrayFunctionPointers.cpp

Functor

- Where a pointer to a function is required – can instead put object of a class that overloads operator ( ) (function call).
- Object like that is called function object, and can be used like a function or function pointer.
- Call operator ( ) by using object name plus parentheses with arguments inside.
- Functor = function object + function.

Functor Example

```
class AddNum{
   int num;
   public:
     AddNum (int m) : num(m) {}
     int operator()(int x) { return num + x;}
}
AddNum add44(44);
int newNum = add44(8);   //newNum == 52
```

Other Topics

Enum

- enum Mood { HAPPY, FROWNY, NEUTRAL};
- Mood current = HAPPY;
- if (current == FROWNY) current = NEUTRAL;
- In reality: HAPPY = 0, FROWNY = 1, NEUTRAL = 2;
- enum Months {JAN = 1, FEB, MAR, APR, MAY, …, DEC};

const_cast< T > (v)

- Adds or removes **const** or **volatile** modifiers
- Single cast removes all modifiers
- Result is an rvalue unless T is a reference
- Types cannot be defined within **const_cast**

```
const int a = 10;
const int* b = &a;
int* c = const_cast< int* > (b);
*b = 20;  //compiler error
*c = 30;  //OK
```

Namespace

- Namespace defines a scope in which identifiers and variables are placed.
  - Try to help with naming conflicts.
- To use a namespace member, need *MyNameSpace*::*member* or using declaration/directive.
- Using declaration (using std::cout;) brings 1 name into scope where declaration is (therefore no need to do std::cout every time).
- Using directive (using namespace std;) brings all names from namespace into scope.

Namespace Example

```
#include <iostream>
using namespace std;
int integer1 = 98;
namespace Example {
    const double PI = 3.14159;
    int integer1 = 8;
    void printValues();
    namespace Inner {
        enum Years{ FISCAL1 =
        1990, FISCAL2 };
    }
}
namespace {
    double doubInUnnamed = 3.2;
}
```

```
int main() {
    cout << doubInUnnamed << endl;
    cout << integer1 << endl;
    cout << Example::PI << " " <<
        Example::integer1 << " " <<
        Example::Inner::FISCAL2 <<
        endl;
    Example::printValues();
    return 0;
}
void Example::printValues() {
    cout << integer1 << " " << PI << " " <<
    doubInUnnamed << " " << ::integer1 <<
    " " << Inner::FISCAL2 << endl;
}
```

Exceptions

- Exception Handling
  - try { … } catch(Exception &e) { cout << e.what(); }
  - Can make derived classes from base exception classes to create your own types of exceptions.
  - Deals with errors and can keep execution of program going.

Exceptions

```
//DivideByZeroException.h
#include <stdexcept>
using std::runtime_error;

class DivideByZeroException :
    public runtime_error {
public:
    DivideByZeroException()
    : runtime_error("div by zero")
    {}
};
```

```
#include "DivideByZeroException.h"
double quotient(int numer, int denom)
{
    if (denom == 0) { throw
        DivideByZeroException();
    } …
}
int main {
    try {double result = quotient(3, 0); }
    catch (DivideByZeroException &d)
    { cout << d.what() << endl; }
    //after exception, execution continues
    return 0;
}
```

Exceptions

```
//DivideByZeroException.h
#include <stdexcept>
using std::runtime_error;

class DivideByZeroException :
    public runtime_error {
public:
    DivideByZeroException()
    : runtime_error("div by zero")
    {}
};
```

```
#include "DivideByZeroException.h"
double quotient(int numer, int denom)
    throw ( DivideByZeroException )
    // above is exception specification
{
    if (denom == 0) { throw
        DivideByZeroException();
    } …
}
int main {
    try {double result = quotient(3, 0); }
    catch (DivideByZeroException &d)
    { cout << d.what() << endl; }
    //after exception, execution continues
    return 0;
}
```

Exception Specification

- This is a guarantee that the function will throw only exceptions listed in specification (or classes derived from those)
- You can specify a comma-separated list of types
- A function with no exception specification allows ALL types of exceptions
- A function that has an empty list such as: throw( ) does NOT allow any exceptions

Threads

- Pthreads
  - http://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm
  - http://codebase.eu/tutorial/posix-threads-c/
- Lots of examples, including C++ thread class
  - http://stackoverflow.com/questions/266168/simple-example-of-threading-in-c
  - http://www.cplusplus.com/reference/thread/thread/
  - http://www.codeproject.com/Articles/540912/Cplusplus-11-Threads-Make-your-multitasking-life-e

Optional

Scope

```
1.  int x = 1;                          //file scope
2.  void useStaticLocal();              //function prototype scope
3.  void useGlobal();                   //function prototype scope
4.  int main() {
5.     int x = 5;                       //block scope
6.     { int x = 7; }                   //block scope
7.     useStaticLocal ();
8.  }
```

```
1.  void useStaticLocal () {
2.     static int x = 83;  //block scope
3.     x++;
4.  }
```

Scope

```
1.  int x = 1;
2.  void useStaticLocal();
3.  void useGlobal();
4.  int main() {
5.      int x = 5;
6.      { int x = 7; }
7.      //how do we access global x?
8.  }
```

Scope

```
1.  int x = 1;
2.  void useStaticLocal();
3.  void useGlobal();
4.  int main() {
5.     int x = 5;
6.     { int x = 7; }
7.     cout << ::x << endl;
8.  }
```

Unary scope resolution operator ::

Only use with global variables, not locals in outer block

Not good style to have global and local variables with same name!

Volatile/Mutable/const_cast

- Keyword **volatile** means variable could be modified by hardware not known to the compiler. Key to tell compiler not to optimize it.
- Cast **const_cast** adds or removes **const** and **volatile** modifiers
  - Useful when get const char* back from function, and you need to modify it.
- Keyword **mutable** is an alternative to const_cast.
  - mutable member variable is always modifiable even with const member function or const object of that class.