Templates, Basic STL, Lambdas

ITP 435 – Spring 2016
Week 2, Lecture 2
Lecturer: Sanjay Madhav

USC Viterbi School of Engineering
University of Southern California

Multiple Classes, Separate Files

- Suppose there are two classes: ClassA and ClassB. ClassB has-an instance of ClassA.
- You normally might do:

ClassA.h	ClassB.h

`#pragma once`
`class ClassA`
`{`
`   // ...`
`};`
	`#pragma once`
	`#include "ClassA.h"`
	
	`class ClassB`
	`{`
	`private:`
	`   ClassA myClassA;`
	`};`

Multiple Classes, Separate Files

ClassA.h	ClassB.h

`#pragma once`
`class ClassA`
`{`
`   // ...`
`};`
	`#pragma once`
	`#include "ClassA.h"`
	
	`class ClassB`
	`{`
	`private:`
	`   ClassA myClassA;`
	`};`

- ***Problem:*** What happens if "ClassA.h" has a lot of other includes in it, too?

Circular Dependecy

- What if ClassA "has-a" ClassB, too?!

ClassA.h	ClassB.h

`#pragma once`
`#include "ClassB.h"`
 	
`class ClassA`
`{`
`private:`
`   ClassB myClassB;`
`};`
	`#pragma once`
	`#include "ClassA.h"`
	
	`class ClassB`
	`{`
	`private:`
	`   ClassA myClassA;`
	`};`

- (This won't compile...)

Forward Declarations

- A ***forward declaration*** solves both of these problems...

ClassA.h	ClassB.h

`#pragma once`
 	
`class ClassA`
`{`
`private:`
`   class ClassB* myClassB;`
`};`
	`#pragma once`
	
	`class ClassB`
	`{`
	`private:`
	`   class ClassA* myClassA;`
	`};`

Forward Declarations, Cont'd

ClassA.h	ClassB.h

`#pragma once`
 	
`class ClassA`
`{`
`private:`
`    class ClassB* myClassB;`
`};`
	`#pragma once`
	
	`class ClassB`
	`{`
	`private:`
	`    class ClassA* myClassA;`
	`};`

- There are three main aspects of using a forward declaration:
    1. Don't include the header that defines the class you're forward declaring
    2. When you use the custom type, add the "class" keyword in front of it – this is the actual forward declaration
    3. A forward declared type can only be used as a pointer or reference, since they have a known, constant size – otherwise C++ doesn't know how much space to reserve

Basic Template Syntax (classes)

```
template <class T>
class List
{
    // ...
};
```

Or

```
template <typename T>
class List
{
    // ...
};
```

Basic Template Syntax (functions)

```
template <class T>
T max(T a, T b)
{
    return ((a > b) ? a : b);
}
```

or

```
template <typename T>
T max(T a, T b)
{
    return ((a > b) ? a : b);
}
```

Compiler Generation

- If you use this template in code as such:

```
max(1, 2); // Type not specified, compiler attempts substitution
max<char>('a', 'b'); // Type specified (optional)
```

- Compiler will generate two versions of our function:

```
int max(int a, int b)
{
    return ((a > b) ? a : b);
}

char max(char a, char b)
{
    return ((a > b) ? a : b);
}
```

Template Specialization

- Suppose you want to do something specific in max when the type is std::string
- You can then specify a specialization:

```
template <>
std::string max<std::string>(std::string a,
  std::string b)
{
     // Code specific for this case
}
```

Non-type templates

- Templates don't have to be only based on type, C++ allows you to use the following as template parameters:
  - Constant integral expressions (int, bool, enum, char)
  - pointers to global functions with external linkage
  - pointers to static member functions
  - pointers to global constants with external linkage
  - pointers to static member variables

Non-type templates, cont'd

- Example:

```
template <typename T, int size>
class A
{
	// ...
};

// Use later...
A<int, 5> my_A;
```

Non-type templates, cont'd

- Note that they just have to be constant *expressions*. If the compiler can evaluate it constantly, it will work.
- Suppose you have this:

```
template <int size>
class A
{
    int get_size() { return size; }
};
```

- You could declare two instances of A like this, and they would both have a template parameter "size" 2:

```
A<2> a1;
A<6/3> a2;
```

Why you can't use floats

- Suppose you could use floats:

```
// Not valid template
template <float value>
class A
{
};
```

- These are both constant expressions we might want to use the same template, but they may not due to floating point errors

```
A<3.0f/1.0f> my_A1;
A<3.0f> my_A2;
```

Default Parameter

- You can specify a default template parameter

```
template <typename T, int size = 20>
class A
{
};
```

- This would default to a size of 20:

```
A<int> my_A;
```

Second use of typename

- If you have this:

```
template <typename T>
class A
{
    class X { };
};
```

- The following is an error because the compiler doesn't know X is necessarily a member type:

```
template <typename T>
class B : class A<T>
{
    A<T>::X m_inst; // Error, doesn't know if A<T>::X is a type
};
```

Second use of typename, cont'd

- In order to fix the error, we can tell the compiler it's definitely a type:

```
template <typename T>
class B : class A<T>
{
    typename A<T>::X m_inst; // Works b/c of typename
};
```

Auto keyword

- Auto tells the compiler to deduce the type:

```
// long any annoying
std::vector<int>::iterator myIter = myVect.begin();

// short and sweet
auto myIter = myVect.begin();
```

- Added in C++11, supported in every current compiler

STL Containers

- Sequence Containers
  - vector
  - array (C++11 only, in VS 2010)
  - deque
  - list
  - forward_list (C++11 only, in VS 2010)
- Container Adaptors
  - queue
  - priority_queue
  - stack

STL Containers, Cont'd

- Associative Containers
  - map and multimap (map is a red-black tree)
  - set and multiset (set is a red-black tree)
  - unordered_set, unordered_multiset, unordered_map, unordered_multimap (C++11 only; hash tables)
- pair and tuple (tuple is C++11 only)

std::vector

- Dynamically sized array

```
#include <vector>

// Create vector of ints
std::vector<int> v;
// Add element
v.push_back(0);
// Access element at index 0 (no bounds check is performed)
v[0] = 5;
// Get back element
int i = v.back(); // i = 5
// Remove back element
v.pop_back();
// Increase maximum size to 50
v.resize(50);
```

std::array

- Statically sized array (C++11 only, works in VS 2010)

```
#include <array>

// Create array of 20 ints
std::array<int, 20> a;
// Access element at index 0, no bounds checking
a[20] = 5; // Bad code!!
// Get back element
int i = a.back(); // i = undefined
```

std::deque

- Double ended queue
- Sort of like a vector, but you can add both to the front and back
- Not guaranteed to be in a contiguous block of memory, so you can't use pointer arithmetic

```
#include <deque>

// Create deque of ints
std::deque<int> d;
// Add element to back
d.push_back(0);
// Add element to front
d.push_front(10);
// Access element at index 0
d[0] = 5;
```

USC Viterbi School of Engineering

University of Southern California

std::list

- Doubly-linked list…so you can add to front and back
- Random access sucks (since it's a linked list)

```
#include <list>

// Create list of ints
std::list<int> l;
// Add element to back
l.push_back(0);
// Add element to front
l.push_front(10);
```

USC Viterbi
School of Engineering

University of Southern California

std::forward_list

- Singly-linked list…so you can only add to the front
- You can only iterate from the front
- C++11 only (supported in VS 2010)

```
#include <forward_list>

// Create list of ints
std::forward_list<int> f;
// Add element to front
f.push_front(10);
```

Container Adaptors

- These are conceptual containers which are implemented in terms of one of the other containers
- queue – By default uses deque, but you can also use list

```
#include <queue>
// This uses deque
std::queue<int> q1;
// This uses list
std::queue<int, std::list<int>> q2;
```

- priority queue – By default uses vector, but can also use deque
- stack – By default uses deque, but you can also use vector or list

std::map

- An *ordered* map of key, value pairs
- In map, keys have to be unique. In multimap, they don't have to.

```
#include <map>
#include <string>
// Key is a std::string, and the value is a pointer
// Note using a string as a key is slow
std::map<std::string, Card*> m;
// Add element to map with key "Joker"
m["Joker"] = new Card();
// Alternative, potentially more efficient way to add element
m.emplace("King", new Card());
// Find an element...returns an iterator to that element
auto iter = m.find("Joker");
// Call Draw on the Joker’s card
(iter->second)->Draw();
```

std::unordered_map

- Like std::map except it's a hash table (C++11 only)

```
#include <unordered_map>
#include <string>
// Key is a std::string, and the value is a pointer
// Using std::string as a key is fine because it's hashed
// using std::hash functions (also C++11!)
std::unordered_map<std::string, Card*> u;
// Add element to map with key "Joker"
u["Joker"] = new Card();
// Alternative way to add element
u.emplace("King", new Card());
// Find an element...returns an iterator to that element
auto iter = u.find("Joker");
// Call Draw on the Joker's Card
(iter->second)->Draw();
```

USC Viterbi School of Engineering

University of Southern California

This is basically a hash table

Set versions

- A std::set is exactly like std::map except the key and value are the same exact thing

```
#include <set>
#include <string>
// Create a set of strings
std::set<std::string> s;
```

- Ditto for std::multiset, std::unordered_set, ...

Pair

- std::pair is a pair of elements, it's not restricted to just maps
- You could make a vector like this, for example:

```
#include <vector>
// This vector stores pairs of month numbers and names
std::vector<std::pair<int, std::string>> v;
// Add January
v.push_back(std::pair<int, std::string>(1, "January");
// This would output 1,January
std::cout << v[0].first << "," << v[0].second;
```

Tuple

- std::tuple is like pair but you can have an unlimited number of elements

```
#include <vector>
#include <tuple>
// This vector stores tuples with month num, short name, long name
std::vector<std::tuple<int, std::string, std::string>> v;
// Add January
v.push_back(std::make_tuple(1, "Jan", "January"));
// This would output 1,Jan,January
std::cout << std::get<0>(v[0]) << "," << std::get<1>(v[0]) << "," <<
   std::get<2>(v[0]);
```

<algorithm>

- Many useful functions:
- random_shuffle
- for_each
- sort
- stable_sort

- And lots, lots more…
- I won't cover everything, but just know <algorithm> has your back

std::for_each

- Normally, you could iterate through something like this:

```
// Assume I have a list<int> called myList
// This would output each element
for (auto i = myList.begin();
     i != myList.end();
     ++i)
{
     std::cout << *i << std::endl;
}
```

- I could instead do this with std::for_each
- std::for_each takes three parameters:
  - Iterator to beginning of range
  - Iterator to end of range
  - Function pointer OR function object OR lambda expression

Using a Function

- If we want to just use a function pointer, we could do this:

```cpp
// This function would be declared elsewhere
void myFunc(int i)
{
    std::cout << i << std::endl;
}

// Later on...

// Assume I have a list<int> called myList
// This would output each element
std::for_each(myList.begin(), myList.end(), myFunc);
```

- Problems??

USC Viterbi School of Engineering

University of Southern California

We don’t retain any state from element to element

The result of the for_each isn’t clear. You have to go look at another function to see what happens

Retaining State

- Function Object (aka Functor)
- Suppose we want to track the number of elements we output:

```cpp
// This is our function object
struct Functor
{
   int count;
   Functor(): count(0) { }
   void operator()(int i) { std::cout << i << std::endl; count++; }
};
// Later on...
// Create the functor, use it, then output count at the end
Functor myFunc;
myFunc = std::for_each(myList.begin(), myList.end(), myFunc);
std::cout << myFunc.count << std::endl;
```


This is kind of ugly

And also we can’t determine what the for_each does at a glance

Lambda Expressions

- The solution is the magical syntax of Lambda Expressions (C++11 to the rescue!)

```
int count = 0;
std::for_each(myList.begin(), myList.end(), [&count](int i)
{
   std::cout << i << std::endl;
   count++;
});

std::cout << count << std::endl;
```

- In other languages, this may be called anonymous functions (like in Javascript)
- Let's explain this crazy syntax!

Lambda Expression Syntax

Capture Clause

Function Parameters

Return Value

Body

```
[&count](int i)    -> int
{
    std::cout << i << std::endl;
    count++;
}
```

Capture Clause

- This is used to “capture” variables that exist outside the lambda expression, and bring them into the lambda expression
- Variables can be captured by value or reference:

```
// Capture ALL local variables by value (not recommended)
[=]
// Capture ALL local variables by reference (not recommended)
[&]
// Capture x by value and y by reference
[x, &y]
// Capture count by reference, and all other locals by value
[=, &count]
// Capture this by value (can't be captured by reference)
// If you want to use any member functions or variables in the
// lambda, you have to capture this.
[this]
```

Lambdas stored in variables

- You're allowed to store lambdas in a variable
- Instead of figuring out the type, use auto
- Our previous example could be:

```
int count = 0;
auto myLambda = [&count](int i)
{
   std::cout << i << std::endl;
   count++;
};
std::for_each(myList.begin(), myList.end(), myLambda);
std::cout << count << std::endl;
```

- Note that since count is captured by reference, subsequent calls to myLambda would keep the new state of count

Range-Based for

- An even simpler for syntax:

```
std::vector<int> vec;
vec.push_back(10);
vec.push_back(20);

for (int i : vec )
{
   std::cout << i << std::endl;
}
```

Range-based for with "auto"

- Can be combined with auto:

```
std::vector<int> vec;
vec.push_back(10);
vec.push_back(20);

for (auto i : vec ) // NOT automatically & or const
{
   std::cout << i << std::endl;
}
```

USC Viterbi School of Engineering

University of Southern California

Note that auto does not automatically add reference or const. If you want it to be so, you have to say auto& or const auto&

Range-based for with "auto"

- More correct in this case:

```
std::vector<int> vec;
vec.push_back(10);
vec.push_back(20);

for (const auto& i : vec )
{
   std::cout << i << std::endl;
}
```