COMP322 - Introduction to C++

Winter 2011

Lecture 09 - Templates and the STL

School of Computer Science

McGill University

March 15, 2011

Last Time - Function Templates

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
```

Example:

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

To call it:

```
int main(){
  int x=1,y=2,z;
  z = getMax<int>(x,y);
  return 0;
}
```

Class templates - Example 1

- Templates can also apply to classes.
- We can use templates to declare our stack class:

```
template <class T> class Stack {
private:
  T *storage;
  int max;
  int top;
 public:
  Stack(int n = 100);
  ~Stack();
  T pop();
  void push(T);
};
```

- We can use the template to create a stack of `string` objects:

```
int main() {
  Stack<string> sstack;
  sstack.push("bbq");
  sstack.push("omg");
  cout << sstack.pop(); // Print omg
}
```

Defining template members

- It's easiest to define member functions in the class:

```
template <class T> class Stack {
private:
  T *storage;
  int max;
  int top;
 public:
  Stack(int n = 100) {
    storage = new T[n];
    max = n;
    top = -1;
  }
  ~Stack() {
    delete [] storage;
  }
  T pop() {
    if (top >= 0) {
      return storage[top--];
    }
  }
  // ...
};
```

Defining template members

- Alternatively, member functions can be defined outside the class. This adds a bit of extra boilerplate to each definition:

```
template <class T> Stack<T>::Stack(int n) {
  storage = new T[n];
  max = n;
  top = -1;
}

template <class T> Stack<T>::~Stack() {
  delete [] storage;
}

template <class T> void Stack<T>::push(T v) {
  if (top < max - 1) {
    storage[++top] = v;
  }
}
```

Class Templates - Example 2

```
#include <iostream>
using namespace std;

template <class T> class MyPair {
    T a, b;
  public:
    MyPair (T first, T second)
      {a=first; b=second;}
    T getMax ();
};

template <class T> T MyPair<T>::getMax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  MyPair <int> myobject (100, 75);
  cout << myobject.getMax();
  return 0;
}
```

So many T's!

```
template <class T> T mypair<T>::getMax()
```

There are three T's in this declaration.

1. The first one is the template parameter.
2. The second T refers to the type returned by the function.
3. The third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template Specialization

- Suppose we want to define a different implementation for a template when a *specific type* is passed as template parameter
- We can declare a specialization of that template.
- Suppose we have a class `MyContainer` which
  - can store one element of any type
  - has one member function `increase()` which increases its value
- What if we want to store an element of type `char`?
  - more convenient to have a different implementation with member function `uppercase()` instead
- We declare a class template specialization for that type:

Template Specialization - Example

```
#include <iostream>
using namespace std;

// class template:
template <class T> class MyContainer {
    T element;
  public:
    MyContainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <> class MyContainer <char> {
    char element;
  public:
    MyContainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

Continued

```
int main () {
  MyContainer<int> myint (7);
  MyContainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

This would output:

8

J

Non-type parameters for templates

Templates can also have regular typed parameters (similarly to functions). Example:

```
#include <iostream>
using namespace std;

template <class T, int N> class MySequence {
    T memblock [N];
  public:
    void setMember (int x, T value);
    T getMember (int x);
};

template <class T, int N>
void MySequence<T,N>::setMember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N> T MySequence<T,N>::getMember (int x) {
  return memblock[x];
}
```

Continued

```
int main () {
  MySequence <int,5> myints;
  MySequence <double,5> myfloats;
  myints.setMember (0,100);
  myfloats.setMember (3,3.1416);
  cout << myints.getMember(0) << '\n';
  cout << myfloats.getMember(3) << '\n';
  return 0;
}
```

This would output:

100

3.1416

Default parameters

We may also set default values or types for class template parameters. Example:

```
template <class T=char, int N=10> class MySequence {..};
```

We could create objects using the default template parameters by declaring:

```
MySequence<> myseq;
```

Which would be equivalent to:

```
MySequence<char,10> myseq;
```

Templates and multi-file projects

- From the compiler's perspective, templates are not like regular classes or functions
- They are compiled on demand (upon instantiation with specific arguments)
- Usually the interface (.h file) and implementation (definition .cpp file) are separated
- Because templates are compiled when required, the full template definition must be accessible from any compilation unit (source file) that uses it.
- This forces a restriction for multi-file projects: the implementation of a template class or function must be in the same file as its declaration.

Source code issues

- Often, this means the entire template definition is placed in a ".h" file.
- This may expose the implementation, or require extra information to be included during compilation.

Source code issues - export keyword

- Alternatively, we can mark the template explicitly for export:

```
// min.h
template <class T> T min(T, T);

// min.cpp
export template <class T> T min(T x, T y) {
  return (x < y) ? x : y;
}

// client.cpp
#include "min.h"

// use min() as needed
```

- However, `export` is not implemented in many compilers.

Typename keyword

- Historically, C++ re-uses the `class` keyword to declare template parameters which may be any type.
- Arguably, this is confusing. More recent implementations have added the `typename` keyword to address this confusion:

```
template <typename T> class A {
  T *data;
  int sz;
  public:
  /* ... */
}
```

Two prototypes:

```
template <class identifier> declaration;
template <typename identifier> declaration;
```

Both expressions have exactly the same meaning and behave exactly the same way.

Templates and inheritance

- Inheritance is not preserved across templates:

```
template <typename T> class A { /* ... */ };
class B { /* ... */ };
class C : public B { /* ... */ };

int main() {
  B b;
  C c;
  A<B> ab;
  A<C> ac;
  b = c;   // legal
  ab = ac; // error!
}
```

Deriving templates from templates

- We can derive a class template from another template.
- Normally the template parameter will be used as the parameter of the base class:

```
template<class T> class A { /* ... */ };
template<class T> class B: public A<T> { /* ... */ };
```

- Or we could inherit from two different template classes:

```
template <class T> class E { /* ... */ };
template <class T> class F { /* ... */ };
template <class T, class X> class G: public E<T>, public F<X>
{/*... */};
```

Member templates

- A class or class template can contain templates as members:

```
template <typename T> class A {
  // ...
public:
  template <typename X> A(X &arg);
  // ...
};
```

- This syntax would allow us to construct an `A` from an object of an arbitrary type - although presumably a type with some well-known set of operations.

The Standard Template Library

- Templates in action
- Set of C++ template classes to provide common programming data structures and functions
- A generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template

Categories

- Containers and algorithms
  - classes for containing other objects and related algorithms for data manipulations such as searching, sorting, etc.
- Iterators
  - generalizations of pointers, which applies to any object stored in container classes

Containers

STL containers implement well-known abstract data types:

Template	Purpose	Header
Sequence containers		
`vector`	dynamically-sized array	`<vector>`
`deque`	double-ended queue	`<deque>`
`list`	doubly-linked list	`<list>`
Container adaptors		
`stack`	last-in, first-out	`<stack>`
`queue`	first-in, first-out	`<queue>`
`priority_queue`	queue by priority	`<queue>`
Associative containers		
`map`	associative array	`<map>`
`multimap`	multiple-valued map	`<map>`
`set`	set of keys	`<set>`
`multiset`	multiset of keys	`<set>`
`bitset`	array of booleans	`<bitset>`

Generic algorithms manipulate data stored in these container objects

STL vector

- A vector is implemented as a dynamically-sized array
- Elements are stored in contiguous memory
- Fast access by index
- Fast addition/deletion at end of list
- Slow addition/deletion at other locations
- Equivalent of ArrayList in Java

STL vector

A vector of strings

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main(){
  vector<string> SS;

  SS.push_back("Hello.");
  SS.push_back("Is there anybody in there?");
  SS.push_back("Just nod if you can hear me.");

  // Loop by index
  for (int ii=0; ii < SS.size(); ii++){
    cout << SS[ii] << endl;
  }
}
```

Iterators: Vector

Alternatively, we can loop using iterators. An *iterator* is an object that has the ability to iterate through elements within a range, using a set of operators (like the increment (++) and dereference (*) operators).

```
#include <iostream>
#include <vector>
#include <string>
#include <iterator>
using namespace std;

int main(){
  vector<string> SS;

  SS.push_back("Hello.");
  SS.push_back("Is there anybody in there?");
  SS.push_back("Just nod if you can hear me.");

  vector<string>::iterator cii;
  for(cii=SS.begin(); cii!=SS.end(); cii++){
    cout << *cii << endl;
  }
}
```

Sort: STL Style

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool cmp (int i,int j) { return (i>j); }

int main(){
  int array[] = {32,71,12,45,26,80,53,33};
  unsigned int array_size = 8;

  vector<int> myvector (array, array+8);
  // using default comparison (operator <):
  sort (myvector.begin(), myvector.end());
  // using cmp as comparison
  sort (myvector.begin(), myvector.end(), cmp);

  // sort the array
  sort(array, array+array_size);
}
```

Auto pointers

```
template <class X> class auto_ptr;
```

- Is a templated type
- Provides garbage collection: allowing pointers to have the elements they point to automatically destroyed when the auto_ptr object is itself destroyed.

```
#include <iostream>
#include <memory>
using namespace std;

int main(){
  auto_ptr<int> p1(new int(10)), p2;
  p2 = p1; // transfer ownership
  int *ptr = p2.get();
  cout<<*ptr<<endl;
  return 0;
}
```

More STL reference

- **STL book**: The C++ Standard Library - A Tutorial and Reference By Nicolai M. Josuttis
- **STL homepage**: `http://www.sgi.com/tech/stl`
- **STL tutorial**: `http://www.cs.brown.edu/people/jak/proglang/cpp/stltut`