UMassAmherst

CS197c: Programming in C++

Lecture 6

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197c/index.html

Syllabus

- ~~Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs~~
- ~~Lecture 2 : C libraries~~
- ~~Lecture 3 : Classes in C++, C++ I/O~~
- ~~Lecture 4 : Memory & Pointers~~
- ~~Lecture 5 : More Pointers~~
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Today’s lecture

Templates

&

The Standard Template Library (STL)

Template Types

- Function Templates
  - General Templates
  - Overloading
  - Explicit Specializations
- Class Templates
  - General
  - Explicit

So what do templates look like?

- There are function templates:

```
template <class T>
void quickPrint(T t) { cout << t << endl; }

quickPrint<float>(4.5);
quickPrint<string>("fourteen");
```

So what do templates look like?

- …and class templates

```
template <class T>
class Foo {
// declarations
   bool contains(T &item);
};

template <class T>
bool Foo<T>::contains(T &item)
{ //definition using T }
```

General Idea Behind the Magic...

`quickPrint<float>(4.5);` compiles to

```
void quickPrint(float t) { cout << t << endl; }
```

and

`quickPrint<string>(“fourteen”);` compiles to

```
void quickPrint(string t) { cout << t << endl; }
```

Function Templates

- Generic programming for functions:

```
template <class T>   // ← newer: <typename T>
void reverse(T arr[], int c) {
   T tmp;
   for (int i = 0; i < (c/2); i++){
         tmp = arr[i];
         arr[i] = arr[c-i-1];
         arr[c-i-1] = tmp;
   }
}
```

Specializing the Template:

- Given:

```
template <class T> void reverse(T arr[], int c)
```

- In action:

```
int nums[10];
for (int i = 0; i < 10; i++) nums[i] = i;
reverse(nums, 10);  // ← note the specialization is implicit
```

Function Templates (cont'd)

- We can overload as well…

```
template <class T> void reverse(T arr[], int c) {…}

template <class T> void reverse(T *arr, int c) {
    T tmp;
    for (int i = 0; i < (c/2); i++){
            tmp = arr[i];
            arr[i] = arr[c-i-1];
            arr[c-i-1] = tmp;
    }
}
```

Function Templates (cont'd)

- Explicit Specialization: special cases
- Given: `template <class T> void reverse(T arr[], int c)`

```
template <> void reverse<int>(int arr[], int c) {
    // do cool stuff specific to ints
}
```

Function Templates (cont'd)

- General Templates are familiar
  - Built for any appropriate type
- Explicit Specializations: specific behavior for one type
- Selection preferences:

1) non-template	void reverse(int arr[], int c);
2) exp. specialization	template <> void reverse<int>(int arr[], int c);
3) template	template <class T> void reverse(T arr[], int c);

Class Templates

- Generic programming for whole classes:

```
template <class T>
class Trie
{
   // declarations, members, etc.
   void add(T &item);
};

template <class T> void Trie<T>::add(T &item) {
   // def
}
```

Class Templates (cont'd)

- To use our class template:

```
#include "Trie.hpp"

Trie<int> myTrie;

// cool stuff here
```

- Note: specialization is explicit

Class Templates (cont'd)

- Further reading (if you're interested...):
  - Recursive templating
  - expression arguments
  - explicit specializations (similar to function templates)
  - partial specializations
  - templates as members
  - templates as parameters
  - and more...

Some Template Gotchas

- Templates are *not* defined functions and classes; they are meta-defintions
  - Don’t compile as usual
  - Separate copy for each specialization
  - Also means you may need to put the function “definitions” in the header file
    - cf `export`

- Can lead to code bloat

What is the STL?

- A library of template based reusable components
  - Implements common data structures and algorithms
- Uses *generic programming* based on *templates*
  - Similar to generics in Java

Why use STL at all ?

- Code has been tested for decades
  - Solid, efficient code base
- Provides interfaces which are easy to use
- Do not have to mess with internal mechanism of how the data structures are implemented

So what is the STL made up of ?

- Containers
  - Data structures that store objects of any type
- Iterators
  - tool used to traverse elements in containers
- Algorithms
  - common algorithms like searching and sorting

Containers

- Sequence containers
  - Linear data structures
  - E.g. lists, vectors, deque
- Associative containers
  - Non-linear containers
  - E.g. maps, multimaps, sets, multisets
- Container adapters
  - Constrained sequence containers
  - E.g., stacks and queues

Sequence containers

- Commonalities
  - assume an ordering of elements
  - allows sorting
- Examples:
  - vector (based on array)
  - queue (based on...queue)
  - deque (double-ended queue)
  - list (based on linked lists)

Some methods for sequences

- size()
- capacity()
- insert(iterator, const T & x)
- erase(iterator)
- pop_back()
- reserve()
- clear()

Vectors

- based on arrays
- Allow direct access to its elements using an overloaded ‘[]’ operator
- Insertion at the end is efficient
  - Use the `push_back` method
- Insertion and deletion in the middle is expensive
  - An entire portion of the vector needs to be moved

Example of using a vector

```
/* creating a vector of ints */
#include <iostream>
#include <vector>
using namespace std;
void main() {
  vector<int> v;
  vector<int>::iterator it;

   v.push_back(2);
   v.push_back(3);
   v.push_back(4);

 /* display the content of v */
 for(it = v.begin(); it != v.end(); it++) {
   cout<<(*it)<<endl;
  }

 cout << “the size of the vector is ”;
 cout << v.size() << endl;
 cout << “ the capacity of”;
 cout << “ the vector is ”;
 cout << v.capacity() << endl;
}
```

Lists

- Implemented using doubly linked list
- Insertion and deletion are efficient at any point of the list
- Bidirectional iterators are used to traverse the list in both directions

Deque *("deck")*

- Stands for double-ended queue
- Combines the benefits of list and vector
- Provides indexed access using indexes (not possible in lists)
- Provides efficient insertion and deletion (not efficient in vectors)

Associative containers

- Associative containers use keys to store and retrieve elements

- There are four types of associative containers
  - Set , multiset, map, multimap
  - All associative containers maintain keys in sorted order
  - All associative containers support bidirectional iterators

Mulitsets

- Multisets are implemented using red-black trees
- Multiset allows duplicate keys
- The ordering is determined by the STL comparator function object less<T>
- Keys sorted by less<T> must support the comparison operator

Sets

- Exactly like multisets
- But does not allow duplicate keys

Example of (multi)set

```
#include <iostream>
#include <set>

void main() {
  multiset<int, less<int> > ms;
  ms.insert(10);
  ms.insert(10);
  cout<<ms.count(10);
 multiset<int, less<int> >::iterator it = ms.find(10);
 if(it != ms.end()) cout<<"10 was found"<<endl;
}
```

Multimaps

- associate keys to values
- implemented using red-black trees
- insertion is done using objects of the class pair
- the ordering is determined by the STL comparator function object less<T>

Multimaps example

```
#include <iostream>
#include <map>

int main()
{
multimap<int, double> mp;
mp.insert(pair<int,double>(10,14.5));
multimap<int, double>::iterator it;
for(it=mp.begin(); it != mp.end();it++)
  cout<<it->first<<“ “<<it->second;
}
```

Container Adapters (Stacks)

- LIFO data structure
- They are implemented with vector, list, and deque(by default)
- Header file <stack>
- Examples
  - stack using vector : `stack<int, vector<int>> s1`
  - stack using list : `stack<int, list<int>> s2`
  - Stack using deque : `stack<int> s3`

Iterators

- Iterators are pointers to elements of sequence and associative containers
  - Type **const_iterator** defines iterator to a container element that *cannot* be modified
  - Type **iterator** defines iterator to a container element that *can* be modified
- All sequence and associative containers provide the **begin()** and **end()** methods
  - Returns iterators pointing to the first and one after the last element in the container

Iterators (cont..)

- If it points to a element in the container
  - it++ (++it) points to the next element in the container
  - (*it) is the actual element in the container

- The iterator resulting from end() can only be used to detect whether the iterator has reached the end of the container.

Iterators

- Iterators are pointers to elements of sequence and associative containers
  - Type **const_iterator** defines iterator to a container element that *cannot* be modified
  - Type **iterator** defines iterator to a container element that *can* be modified
- All sequence and associative containers provide the **begin()** and **end()** methods
  - Returns iterators pointing to the first and one after the last element in the container

Iterators (cont..)

- If it points to a element in the container
  - it++ (++it) points to the next element in the container
  - (*it) is the actual element in the container

- The iterator resulting from end() can only be used to detect whether the iterator has reached the end of the container.

STL Algorithms

- Some popular ones:
  - sort
  - binary_search
  - min/max
  - set_union / set_intersection
  - and more…

STL Algorithms (cont'd)

- sort example:

```
bool mycomp(int i, int j) { return (i<j);}
int nums[] = { 19, 2, 45, 6 };
vector<int> myvec(nums, nums+4);

sort(myvec.begin(), myvec.end(), mycomp);
```

STL Algorithms (cont'd)

- copy example:

```
vector<int> myvec();
myvec.push_back(7);
myvec.push_back(11);
vector<int> newvec(myvec.size());

copy(myvec.begin(), myvec.end(), newvec.begin());
/* newvec better have enough room! */
```

Next class

Reflection,

C++ Exception,

and C++11