Lecture 13c
The Standard Template Library (STL)
April 22nd, 2016

Announcements
• PA9 is due next Wednesday
• Reading Assignment
  – Chapter 10 for next Monday
  – Your last quiz for this course!
• Recitation next week
  – Installing & using google test (from scratch)
  – The last mandatory recitation

Templates Enable Vectors of Anything
```cpp
template<typename ELEMENT>
class vector {
public:
  vector(unsigned int sz = 0);
  ~vector();
  void push_back(ELEMENT element);
  ELEMENT& operator [] (unsigned int index);
private:
  unsigned int size;
  unsigned int capacity;
  ELEMENT* data;
};
```

Where to put templates?
• Put the entire template class – definition and implementation – in one .tpp file.
  – This is called the Borland convention
  – If the .tpp file is included in multiple .cpp files, it may cause the same functions to be multiply defined
  – In the Borland convention, the linker sorts this out
    • The Gnu linker support the Borland convention
    • So does Apple
    • The STL uses it too (except the .tpp filename)
  – Downside: slows the compiler
    • As templates are compiled many times

Up Next…
• The greatest strength of Java is its extensive library
  – Rarely build from scratch
  – Stand on the shoulders of giants!
• In C++, we have asked you to build from scratch
  But C++ has a library! It’s called the Standard Template Library (STL), and it consists of templated libraries and templated functions

STL
1. Containers
   – Abstractions of data structures
2. Iterators
   – Abstractions of pointers
   – Allow you to iterate through containers
3. Algorithms
   – Universal algorithms (e.g. sort)
     • Without reference to object type being acted on
     • Without reference to data structure holding the objects
Containers

- Containers are data structures
  - capable of holding objects of any data type.
  - But only a single object type per container
  - Of course, polymorphism still works
    - But be careful of splicing
- Containers are templated classes
  - Example: vector
- They hold what you tell them to
  - vector<int> != vector<int>* != vector<int&>

Containers defined by performance properties

- In the C++ specification, it doesn’t say how vectors should be implemented
- But it does describe their performance:
  - The cost of pushing n elements must be O(n \log(n))
  - If space is pre-reserved, the cost of pushing n elements must be O(n)
  - The cost of access must be O(1)
  - The cost of insertion must be O(n)
  - The cost of deletion must be O(n)

Vectors

- We have been using vectors since day 1, but how are they defined?
  - Elements accessed by at() or []
  - Extended via push_back()
  - Shortened via pop_back()
- Roughly the equivalent of Java’s ArrayList
- But this is a description more than a definition.

Deque

- Please don’t make me pronounce this
  - Rhymes with cheek
  - Or rhymes with deck
  - Or sounds like “D-Q”
- Conceptually, a double ended vector
  - Push_front() defined and O(log(n))
  - Pop_front() defined
  - Puts active part of vector in the middle of reserved space, instead of at the front
  - Same O() performance as vector…

List

- Doubly-linked list
- Different performance
  - Insert is now O(1), given location
  - Delete is now O(1), given location
  - Access is now O(n)
- Types of trade-offs you studied in CS200
  - Forward_list is similar, but singly linked
    - Takes up less memory
    - Can’t go backwards through the list
    - More when we get to iterators…
Sequence vs Non-sequence Containers

- array, vector, deque, list and forward_list are sequence containers
  - Objects stored in a fixed order
  - Objects accessed through indices
    - Via at() and []
- Set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map & unordered_multimap are non-sequence containers
  - Data values determine where they are stored

Set

- Acts like a set in mathematics
  - Cannot store the same value twice
  - Inserting an element that is already in the set is a no-op
- Basic methods:
  - Size() – number of elements in set
  - Insert() – add object to set
  - Erase() – remove element from set
  - Count() – number of times element appears (0 or 1)
  - Find() – returns iterator to element
- Performance
  - Insert, erase, count and find are all $O(\log n)$
  - Implemented as binary trees

Multisets & Unordered_sets

- Multisets are like set, but the same value can be stored multiple times
  - The Count() method is therefore more interesting
  - Same performance as set
- Unordered_sets are like sets, but stored through a radix or hash function
  - Constant time access
  - More memory used

Pairs

```cpp
template<typename FIRST, typename SECOND>
class Pair {
    public:
        inline FIRST& First() {return first;}
        inline const FIRST& First() const {return first;}
        inline SECOND& Second() {return second;}
        inline const SECOND& Second() const {return second;}
    protected:
        FIRST first;
        SECOND second;
};
```

Maps & Multimaps

- Maps are sets of <key, value> pairs
  - Still implemented as trees
- Data is stored and accessed according to the key value
  - Example <student_id, student_record>
    - Every key must be unique
- Multimaps allow multiple instances of key values
- Unordered maps are like maps, but implemented via radix or hash
- Unordered multimaps...