Lecture 13a
More Templates
November 14th, 2017

Announcements

• ACM (Wed. @ 6 in CSB130)
  – Representation Development for Faster Deep Reinforcement Learning
  – Prof. Chuck Anderson
• ACM-W (Tue @ 5 in CSB130)
  – Travelport Developer panel, here to talk about working at Travelport, the industry, and answer any questions!
• Reading Assignment
  – Chapter 10 (with Quiz)
  – Last reading assignment of semester!
• PA8 due Thursday
  – Any questions?

An Example Container...

```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;
};
```

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<short>

Container Types

• array (not dynamic) (C++11)
• vector
• deque (two-ended dynamic arrays)
• forward_list (singly linked) (C++11)
• list (doubly linked)
• set (binary tree)
• map (key/value binary tree)
• multimap (key/value binary tree with repeated keys)
• unordered_set (hash table, no repeated elements) (C++11)
• unordered_multiset (hash w/ repeated elements) (C++11)
• unordered_map (key/value hash table) (C++11)
• unordered_multimap (key/value hash table w/ repeated keys) (C++11)
• queue
• stack
• heap

To access the C++11 standard, compile with the `-std=c++11` flag
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, not a definition.

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 $O(n)$

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 $O(n)$
- The 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 (abstract pointer) to element
- Performance
  - Insert, erase, count and find are all $O(\log n)$
  - Implemented as binary trees
Set Example (Part 1)

```cpp
void store(set<int>& number_set) {
    int number;
    while (!cin.fail()) {
        cout << "Input number to store (or 'q' to quit):");
        cin >> number;
        if (!cin.fail()) number_set.insert(number);
    }
    char clear_ch;
    cin.clear();
    cin >> clear_ch;
}
```

Set Example (Part 2)

```cpp
void retrieve(set<int>& number_set) {
    int number;
    while (!cin.fail()) {
        cout << "Input number to retrieve (or 'q' to quit):");
        cin >> number;
        if (!cin.fail()) {
            if (number_set.count(number))
                cout << number << " is present in set" << endl;
            else cout << number << " is missing from set" << endl;
        }
    }
```

Set Example (Part 3)

```cpp
int main(int argc, char* argv[]) {
    set<int> number_set;
    store(number_set);
    retrieve(number_set);
    return 1;
}
```

Multisets & Unordered_sets

- Multisets are like sets, but the same value can be stored multiple times
  - The Count() method is therefore more interesting
  - Same O() performance as set
- Unordered_sets are like sets, but stored through a radix or hash function
  - O(1) access
  - More memory used

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...

Map Example (Part 1)

```cpp
void store(map<string, char>& grade_map) {
    string name;
    char grade;
    bool quitp = false;
    while (!quitp) {
        cout << "Input name and grade (or 'q' to quit):");
        cin >> name;
        if (name.compare("q") == 0) {
            quitp = true;
        } else {
            cin >> grade;
            grade_map[name] = grade;
        }
```
Map Example (Part 2)

```cpp
void retrieve(map<string, char>& grade_map) {
    string name;
    bool quitp = false;
    while (!quitp) {
        cout << "Input name to retrieve (or 'q' to quit):";
        cin >> name;
        if (0 != name.compare("q")) {
            cout << name << " has a grade of "
            << grade_map[name] << endl;
        } else quitp = true;
    }
}
```

Let's discuss PA9...

- What containers should you use?
  - Starting point:
    - For every input document:
      - Create vector of strings (returned by `>>`)
      - Copy to vectors of parsed strings ("a.b" -> "a", ",", ".")
      - Resolve capitalization
      - Compute reading level
      - Copy to vector of stemmed strings ("gets" -> "get")
      - Sort vector
      - Create vector of <string, count> pairs
    - For corpus
      - Create vector of <string, document_count> pairs
    - For every input document:
      - If within reading level range, compute TFIDF to target document
    - Select max TFIDF
  - How do you make this faster?

What do Containers share?

- Common methods:
  - `begin()`, `end()`, `cbegin()`, `cend()`
  - `size()`, `max_size()`, `empty()`
  - `front()`
  - `emplace()`, `insert()`, `erase()`, `swap()`, `clear()`
- More specific methods:
  - `rbegin()`, `rend()`, `rcbegin()`, `rcend()`
    - If backwards is supported
  - `reserve()`, `resize()`
    - Containers with contiguous memory
  - `at()`, `[]`
    - For indexed containers (arrays, vectors, deques, maps, ...)
  - `find()`
    - For non-sequential containers