CS 106B
Lecture 6: Sets and Maps

Friday, April 14, 2017

Programming Abstractions
Spring 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 5.4-5.6
Today's Topics

• Logistics:
  • Tiny Feedback: some responses!
  • Not enough motivation for why we care about ADTs: good point!
  • More interactive classes: I'll see what I can do!
  • KEY pages (🔑)

• Late credits change: *up to two calendar days equals one late credit.* You get three automatic late credits per quarter.

• the "const" qualifier
• Postfix refresher
• Structs (details will come later!)
• Sets
• Maps
• When we pass variables by reference into a function, we do so for a couple of reasons:
  • We don't want to make copies of big objects
    • As it turns out (thanks to the person who put a note on sayat.me/chrisgregg who reminded me to mention this), C++ has new functionality that allows us to return big objects in some cases without lots of copying (but the Stanford libraries don't have that functionality yet)

    and / or

• We need to modify an object in place (we will do this a great deal with recursion)
• What if we want to pass a variable by reference, but we know we won't modify it?

• We could just have self-control and not modify it.

• Or, we could make the compiler keep us honest. To do this, we use the keyword `const`.
• `const` allows a programmer to tell the compiler that the object passed cannot be changed in the function. E.g.,

```cpp
void printLifeGrid(Grid<char> const &lifeGrid);
```

• There is no need for the `printLifeGrid()` function to change the `lifeGrid`, but we would rather pass the grid by reference to avoid big copies.
• We use `const` to tell the compiler to give us an error if we do try to modify a const-declared variable in a function.

• This *also* tells someone reading our code that we are guaranteeing that the object will be the same when the function ends as when it began.
What does the following postfix (RPN) computation equal?

\[ \begin{align*}
10 & \quad 3 & \quad 5 & \quad * & \quad 9 & \quad 4 & \quad - & \quad / & \quad + \\
\end{align*} \]

Feel free to use our stack algorithm:

Read the input and push numbers onto a stack until you reach an operator. When you see an operator, apply the operator to the two numbers that are popped from the stack. Push the resulting value back onto the stack. When the input is complete, the value left on the stack is the result.

Answer: 13

How would our stack-based RPN know that we had made an error, e.g.,

\[ \begin{align*}
10 & \quad 3 & \quad 5 & \quad * & \quad - & \quad + & \quad 9 & \quad 4 & \quad - \\
\end{align*} \]

Answer: the stack is empty when we try to pop two operands
Brief Introduction to Structs

Recall that in C++, we can only return one value from a function. We have overcome this in the past by using references:

```c
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

Recall that in C++, we can only return one value from a function. We have overcome this in the past by using references:

```c
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

Recall that in C++, we can only return one value from a function. We have overcome this in the past by using references:

```c
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```
There is another way we can return variables by packaging them up in a type called a "struct"

- Structs are a way to define a new type for us to use.
- Once we define a struct, we can use that type anywhere we would normally use another type (e.g., an `int`, `double`, `string`, etc.)

```c
struct Roots {
    double root1;
    double root2;
};
```

- new type name
- struct variables, referred to with dot notation
- don't forget the semicolon
Brief Introduction to Structs

• Let's re-write our quadratic equation solver to use the Roots struct.
Brief Introduction to Structs

• Let's re-write our quadratic equation solver to use the Roots struct.

```c
struct Roots {
    double root1;
    double root2;
};

/*
 * Solves a quadratic equation ax^2 + bx + c = 0,
 * storing the results in output parameters root1 and root2.
 * Assumes that the given equation has two real roots.
 */
Roots quadratic(double a, double b, double c) {
    Roots roots;
    double d = sqrt(b * b - 4 * a * c);
    roots.root1 = (-b + d) / (2 * a);
    roots.root2 = (-b - d) / (2 * a);
    return roots;
}
```
Sets

- Collection of elements with no duplicates.

Maps

- Collection of key/value pairs
- The key is used to find its associated value.

Sets and Maps
Sets

- **set**: a collection of elements with no duplicates.

Operations include **add**, **contains**, and **remove**, and they are all fast.

Sets *do not* have indexes.

```java
set.contains("to")  // true
set.contains("be")  // false
```
Sets: Simple Example

```cpp
Set<string> friends;
friends.add("chris");
friends.add("anton");
cout << friends.contains("voldemort") << endl;
for(string person : friends) {
    cout << person << endl;
}
```
## Set Essentials

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int set.size()</code></td>
<td>Returns the number of elements in the set.</td>
</tr>
<tr>
<td><code>void set.add(value)</code></td>
<td>Adds the new value to the set (ignores it if the value is already in the set).</td>
</tr>
<tr>
<td><code>bool set.contains(value)</code></td>
<td>Returns true if the value is in the set, false otherwise.</td>
</tr>
<tr>
<td><code>void set.remove(value)</code></td>
<td>Removes the value if present in the set. Does not return the value.</td>
</tr>
<tr>
<td><code>bool set.isEmpty()</code></td>
<td>Returns true if the set is empty, false otherwise.</td>
</tr>
</tbody>
</table>

Sets also have other helpful methods. See the online docs for more.
Looping Over a Set

```cpp
for(type currElem : set) {
    // process elements one at a time
}
```

(can't use a normal `for` loop and get each element `[i]`

```cpp
for(int i=0; i < set.size(); i++) {
    // does not work, no index!
    cout << set[i];
}
```
Types of Sets

Set
- Iterate over elements in *sorted* order
- **REALLY FAST!** $O(\log n)$ per retrieval
- Implemented using a "binary search tree"

HashSet
- Iterate over elements in *unsorted (jumbled)* order
- **REALLY, RIDICULOUSLY FAST!** $O(1)$ per retrieval
- Implemented using a "hash table"
Set Operands

Sets can be compared, combined, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 == s2</td>
<td>true if the sets contain exactly the same elements</td>
</tr>
<tr>
<td>s1 != s2</td>
<td>true if the sets don't contain the same elements</td>
</tr>
<tr>
<td>s1 + s2</td>
<td>returns the union of s1 and s2 (all elements in both)</td>
</tr>
<tr>
<td>s1 * s2</td>
<td>returns intersection of s1 and s2 (elements must be in both)</td>
</tr>
<tr>
<td>s1 - s2</td>
<td>returns difference of s1, s2 (elements in s1 but not s2)</td>
</tr>
</tbody>
</table>
Count Unique Words
**map**: A collection of pairs \((k, v)\), sometimes called **key/value** pairs, where \(v\) can be found quickly if you know \(k\).

a.k.a. dictionary, associative array, hash

a generalization of an array, where the "indexes" need not be ints.
A map allows you to get from one half of a pair to the other.

store an association from "Jenny" to "867-5309"

```java
/* key       value
   */
// m["Jenny"] = "867-5309"; or
m.put("Jenny", "867-5309");
```

What is Jenny's number?

```java
/* string ph = m["Jenny"] or
   string ph = m.get("Jenny")
   "206-685-2181"
```

Maps are Everywhere

key = title, value = article

key: "Yosemite National Park"
value:

Yosemite National Park
From Wikipedia, the free encyclopedia

"Yosemite" redirects here. For other uses, see Yosemite (disambiguation).

Yosemite National Park ([yosˈməni] (listen)) is a national park spanning portions of Tuolumne, Mariposa and Madera counties in Northern California. The park, which is managed by the National Park Service, covers an area of 747,886 acres (1,186,691 sq km; 362,687 ha; 3,086.87 km²) and reaches across the western slopes of the Sierra Nevada mountain range. About 4 million people visit Yosemite each year, most spend the majority of their time in the seven square miles (18 km²) of Yosemite Valley. Designated a World Heritage Site in 1984, Yosemite is internationally recognized for its granite formations.

key: "Mariana Trench"
value:

Mariana Trench
From Wikipedia, the free encyclopedia

The Mariana Trench or Mariana Deep[1] is the deepest known part of the world's oceans. It is located in the western Pacific Ocean, to the east of the Mariana Islands. The trench is about 2.550 kilometres (1,580 mi) long with an average width of 69 kilometres (43 mi). It reaches a maximum-known depth of 10,996 metres (36,070 ft) (±40 metres (130 ft) at its southwestern end).[2] Although some unverified measurements place the deepest portion at 11,034 metres (36,201 ft),[3] at the bottom of the trench the water remains deep.
Creating Maps

Requires 2 type parameters: one for keys, one for values.

```cpp
// maps from string keys to integer values
Map<string, int> votes;

// maps from double keys to Vector<int> values
Map<string, Vector<string>> friendMap;
```
## Map Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.clear();</code></td>
<td>removes all key/value pairs from the map</td>
</tr>
<tr>
<td><code>m.containsKey(key)</code></td>
<td>returns true if the map contains a mapping for the given key</td>
</tr>
<tr>
<td><code>m[key]</code> or <code>m.get(key)</code></td>
<td>returns the value mapped to the given key; if key not found, adds it with a default value (e.g. 0, &quot;&quot;)</td>
</tr>
<tr>
<td><code>m.isEmpty()</code></td>
<td>returns true if the map contains no key/value pairs (size 0)</td>
</tr>
<tr>
<td><code>m.keys()</code></td>
<td>returns a Vector copy of all keys in the map</td>
</tr>
<tr>
<td><code>m[key] = value;</code> or <code>m.put(key, value);</code></td>
<td>adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one</td>
</tr>
<tr>
<td><code>m.remove(key);</code></td>
<td>removes any existing mapping for the given key</td>
</tr>
<tr>
<td><code>m.size()</code></td>
<td>returns the number of key/value pairs in the map</td>
</tr>
<tr>
<td><code>m.toString()</code></td>
<td>returns a string such as &quot;{a:90, d:60, c:70}&quot;</td>
</tr>
<tr>
<td><code>m.values()</code></td>
<td>returns a Vector copy of all values in the map</td>
</tr>
</tbody>
</table>
Map Example

Map<string, string> wiki;

// adds name / text pair to dataset
wiki.put("Neopalpa donaldtrumpi", articleHTML);
Map Example

Map<string, string> wiki;

// adds name / text pair to dataset
wiki.put("Neopalpa donaldtrumpi", articleHTML);

// returns corresponding articleHTML
cout << wiki.get("Yosemite National Park");
Map Example

Map<string, string> wiki;

// adds name / text pair to dataset
wiki.put("Neopalpa donaldtrumpi", articleHTML);

// returns corresponding articleHTML
cout << wiki.get("Yosemite National Park");

// removes the article
wiki.remove("Britain in the E.U.");
Types of Maps

Map

Iterate over elements in sorted order

**REALLY FAST!**

$O(\log n)$ per retrieval

Implemented using a "binary search tree"

HashMap

Iterate over elements in unsorted (jumbled) order

**REALLY, RIDICULOUSLY FAST!**

$O(1)$ per retrieval

Implemented using a "hash table"
Map Example: Tallying Votes

count votes:
// (M)ilk, (S)tokes, (R)ogers
"MMMMRSMSSSMMMMMMRMMMMRRRRMMMM"

key:
"M"  "S"  "R"

value:
17  7  3

*In 1976 Harvey Milk became the first openly gay elected official in the US
Tallying Words
Map<string, double> gpa = load();
for (string name : gpa) {
    cout << name << "'s GPA is ";
    cout << gpa[name] << endl;
}

*The order is unpredictable in a HashMap*
Recap

• **Structs**
  • Used to define a type that holds multiple other types.
  • Useful for returning more than one value, or keeping things together (e.g., a coordinate could be an x,y and it is nice to keep them together:)
    ```
    struct coordinate {
        double x, y;
    }
    ```
  • Uses dot notation to access elements.

• **Sets:**
  • Container that holds non-duplicate elements
  • O(log n) behavior per element access (**HashSet**: O(1), but unordered)

• **Map:**
  • Container that relates keys to values.
  • Needs two types when defining: **Map<keyType, valueType>**
  • O(log n) behavior per element access (**HashMap**: O(1), but unordered)
References and Advanced Reading

**References:**
- Stanford Map reference: [stanford.edu/~stepp/cppdoc/Map-class.html](http://stanford.edu/~stepp/cppdoc/Map-class.html)

**Advanced Reading:**
Extra Slides