Maps

CS 1044

Motivation

- Arrays and vectors provide fast access **by numeric position**
- Structures provide fast access **by name**, but only for a **fixed set** of names known at **compile-time**
- What if we want to look up an element using some other, more flexible, criteria?
- Consider a dictionary: You look up a definition based on the word itself, not by looking up the 3924th word

Maps

```
#include <map>
```

- maps support looking up items quickly using any type* of key, not just numeric positions
- The element used to perform the lookup is called the **key**
- The element retrieved by the lookup is called the **value**

* Any type that can be compared with *less-than* (<)

Declaring a Map Variable

- Recall: `vector`s must include their element type inside angle brackets: e.g., `vector<int>`
- Maps must include their **key type** and **value type**

Visualizing a Map

- Think of a map like a table, where each row represents a **key/value pair**
- A particular **key** can only appear **once**
- **Values** can appear **multiple times**

Key	Value
“Joe”	32
“Bob”	17
“Silvia”	26
“Jane”	32
“Frank”	65

Getting/Setting Values

```
map[key] = value;
```

- Inserts a value into the map with the specified key
- If a value for that key already exists, it is replaced

```
value = map[key];
```

- Looks up the item with the specified `key` and returns the value if found

Non-Existent Keys

- What if I do this...

```
map<string, string> dict;
dict["cat"] = "a pettable animal";
string def = dict["dog"];
```

...and there isn’t a value in the map for the key `"dog"`?

Non-Existent Keys

- For implementation reasons, the `[x]` notation on a map **must return a reference** to a value in the map
- There's no notion of a "reference to nothing"
- So, if the key isn't already in the map, it is **inserted into the map** and given a **default value** (0 for numeric values, "" for strings, etc.)
- Moral: Don't use `[]` if you need to check for existence

Checking for Existence

- To check if a key is in a map without the side-effect of inserting it, use the `find` function

```
map<string, string> dict;
map<string, string>::iterator it;
it = dict.find("dog");
```

- Returns `dict.end()` if the key was not in the map

Word Count

- We can use string streams and maps to count the words frequency of words in a file.

```
map <string, int> word_count;

// fin is an open ifstream variable
while(getline(fin, line)
{
  // line comes from somewhere.
  stringstream split(line);

  // Break a line down into words.
  while (split >> word)
  {
      word_count[word]++;
  }
}
```

- But how do we view the contents of the map?

Map Iterators

- For a vector, an iterator represents the location of a **single** piece of data, the element
- So, writing `(*it)` lets you access that element
- Each location in a map represents the location of a **pair of items**: the key and the value
- So, what does writing `(*it)` give us?

Map Iterators

- Writing `(*it)` returns a **structure** that has two fields named `first` and `second`
- `(*it).first` accesses the key
- `(*it).second` accesses the corresponding value
- Writing `(*x).y` is so common in C++ that we have a shortcut: `x->y`

Looping Over the Whole Map

```
map<K, V> m;
map<K, V>::iterator it;
for (it = m.begin(); it != m.end(); it++)
{
    K key = it->first;
    V value = it->second;
    // Do something with key and value...
}
```

Loop runs through the map in **ascending order** based on the sorting of the keys

Maps with Integer Keys

- What is the difference between this

```
vector<string> my_vec;
// ...
my_vec[5] = “hello”;
```

and this?

```
map<int, string> my_map;
// ...
my_map[5] = “hello”;
```

When Using Integer Keys...

- ...vectors are best for **dense** data
  - Elements occur in a **small range**
  - **Most or all** of the slots are used
- ...maps are best for **sparse** data
  - Elements occur in a **large range** of possible keys
  - **Relatively few** of the slots are used