Review from Lecture 13

- How to design and implement algorithms using three steps or stages:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right

Today’s Class — Associative Containers (Maps)

- Maps: associative containers for fast insert, access and remove
- Example: Counting word occurrences
- Pairs
- Map iterators
- Map member functions: `operator[]`, `find`, `insert`, `erase`.
- Efficiency
- Maps vs. vectors vs. lists

Reading: Ford & Topp, Sections 11.1 & 11.3; Koenig & Moo, Chapter 7.

14.1 Maps: Associative Containers

- Maps store pairs of “associated” values.

  - We will see several examples today, in lab tomorrow, and in Lecture 15:
    - An association between a string, representing a word, and an int representing the number of times that word has been seen in an input file.
    - An association between a string, representing a word, and a vector that stores the line numbers from a text file on which that string occurs (next lecture).
    - An association between a phone number and the name of the person with that number (tomorrow’s lab).
    - An association between a class object representing a student name and the student’s info (next lecture).

- A particular instance of a map is defined (declared) with the syntax:

  ```cpp
  map<key_type, value_type> var_name
  ```

  In our first two examples above, `key_type` is a string. In the first example, the `value_type` is an `int` and in the second it is a `vector<int>`.

- Entries in maps are `pairs`:

  ```cpp
  pair<const key_type, value_type>
  ```

- Map iterators refer to pairs.

- Map search, insert and erase are all very fast: $O(\log n)$ time, where $n$ is the number of pairs stored in the map.

Let’s see how this some of this works with a program to count the occurrences of each word in a file. We’ll look at more details and more examples later.
14.2 Counting Word Occurrences

- Here’s a simple and elegant solution to this problem using a map:

```c++
#include <iostream>
#include <map>
#include <string>

int main() {
    std::string s;
    std::map<std::string, int> counters; // store each word and an associated counter

    // read the input, keeping track of each word and how often we see it
    while (std::cin >> s)
        ++counters[s];

    // write the words and associated counts
    std::map<std::string, int>::const_iterator it;
    for (it = counters.begin(); it != counters.end(); ++it) {
        std::cout << it->first << '	' << it->second << std::endl;
    }
    return 0;
}
```

14.3 Maps: Uniqueness and Ordering

- Maps are ordered by increasing value of the key. Therefore, there must be an `operator<` defined for the key.
- Once a key and its value are entered in the map, the key can’t be changed. It can only be erased (together with the associated value).
- Duplicate keys can not be in the map.

14.4 Pairs

The mechanics of using `pairs` are relatively straightforward:

- `pairs` are a templated `struct` with just two members, called `first` and `second`. `Reminder: a struct is basically a wimpy class and in this course you aren’t allowed to create new structs. You should use classes instead.`
- To work with pairs, you must `#include <utility>`. Note that the header file for maps (`#include <map>`) itself includes utility, so you don’t have to include utility explicitly when you use pairs with maps.
- Here are simple examples of manipulating pairs:

```c++
std::pair<int, double> p1(5, 7.5);
std::pair<int, double> p2 = std::make_pair(8, 9.5);
p1.first = p2.first;
p2.second = 13.3;
std::cout << p1.first << " " << p1.second << std::endl;
std::cout << p2.first << " " << p2.second << std::endl;
p1 = p2;

std::pair<const string, double> p3 = std::make_pair(string("hello"), 3.5);
p3.second = -1.5;
// p3.first = string("illegal"); // (a)
// p1 = p3; // (b)
```

- The function `make_pair` creates a pair object from the given values. It is really just a simplified constructor, and as the example shows there are other ways of constructing pairs.
- Most of the statements in the above code show accessing and changing values in pairs.
• The two statements at the end are commented out because they cause syntax errors:
  – In (a), the first entry of p3 is const, which means it can’t be changed.
  – In (b), the two pairs are different types! Make sure you understand this.

• Returning to maps, each entry in the map is a pair object of type:

  pair<const key_type, value_type>

The const is needed to ensure that the keys aren’t changed! This is crucial because maps are sorted by keys!

14.5 Maps: operator[]

• We’ve used the [] operator on vectors, which is conceptually very simple because vectors are just resizable arrays. Arrays and vectors are efficient random access data structures.
• But operator[] is actually a function call, so it can do things that aren’t so simple too, for example:

  ++counters[s];

• For maps, the [] operator searches the map for the pair containing the key (string) s.
  – If such a pair containing the key is not there, the operator:
    1. creates a pair containing the key and a default initialized value,
    2. inserts the pair into the map in the appropriate position, and
    3. returns a reference to the value stored in this new pair (the second component of the pair).
    This second component may then be changed using operator++.
  – If a pair containing the key is there, the operator simply returns a reference to the value in that pair.
• In this particular example, the result in either case is that the ++ operator increments the value associated with string s (to 1 if the string wasn’t already it a pair in the map).
• For the user of the map, operator[] makes the map feel like a vector, except that indexing is based on a string (or any other key) instead of an int.
• Note that the result of using [] is that the key is ALWAYS in the map afterwards.

14.6 Map Iterators

• Iterators may be used to access the map contents sequentially. Maps provide begin() and end() functions for accessing the bounding iterators. Map iterators have ++ and -- operators.
• Each iterator refers to a pair stored in the map. Thus, given map iterator it, it->first is a const string and it->second is an int. Notice the use of it->, and remember it is just shorthand for (*it).

14.7 Exercise

Write code to create a map where the key is an integer and the value is a double. (Yes, an integer key!) Store each of the following in the map: 100 and its sqrt, 100,000 and its sqrt, 5 and its sqrt, and 505 and its sqrt. Write code to output the contents of the map. Draw a picture of the map contents. What will the output be?
14.8 Map Find

- One of the problems with \texttt{operator[]} is that it always places a key / value pair in the map. Sometimes we don’t want this and instead we just want to check if a key is there.

- The \texttt{find} member function of the map class does this for us. For example:
  
  \begin{verbatim}
  m.find(key);
  \end{verbatim}

  where \( m \) is the map object and \( \texttt{key} \) is the search key. It returns a map iterator:

  If the key is in one of the pairs stored in the map, \texttt{find} returns an iterator referring to this pair.
  If the key is not in one of the pairs stored in the map, \texttt{find} returns \( m.\text{end()} \).

14.9 Map Insert

- The prototype for the map \texttt{insert} member function is:
  
  \begin{verbatim}
  m.insert(make_pair(key, value));
  \end{verbatim}

  \texttt{insert} returns a pair, but not the pair we might expect. Instead it is pair of a map iterator and a bool:
  
  \begin{verbatim}
  pair<map<key_type, value_type>::iterator, bool>
  \end{verbatim}

- The \texttt{insert} function checks to see if the key being inserted is already in the map.
  - If so, it does not change the value, and returns a (new) pair containing an iterator referring to the \textit{existing pair} in the map and the bool value \texttt{false}.
  - If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the \textit{newly added pair} in the map and the bool value \texttt{true}.

14.10 Map Erase

Maps provide three different versions of the erase member function:

- void \texttt{erase(iterator p)} — erase the pair referred to by iterator \( p \).

- void \texttt{erase(iterator first, iterator last)} — erase all pairs from the map starting at \texttt{first} and going up to, but not including, \texttt{last}.

- size\_type \texttt{erase(const key\_type& k)} — erase the pair containing key \( k \), returning either 0 or 1, depending on whether or not the key was in a pair in the map

14.11 Exercise

Re-write the \texttt{word\_count} program so that it uses \texttt{find} and \texttt{insert} instead of \texttt{operator[]}.

14.12 Choices of Containers

- We can solve this word counting problem using several different approaches and different containers:
  - a vector or list of strings
  - a vector or list of pairs (string and int)
  - a map
  - ?

- How do these approaches compare? Which is cleanest, easiest, and most efficient, etc.?