

# CSCI-1200 Data Structures — Spring 2018

## Lecture 14 – Associative Containers (Maps), Part 1 (and Problem Solving Too)

**HW6 NOTE:** Do not use the STL `map` or STL `pair` for HW6. (It's okay to use them for the contest.)

### Today's Class — Associative Containers (STL Maps)

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

### 14.1 STL Maps: Associative Containers

- STL maps store pairs of “associated” values.
- We will see several examples today, in lab 9, 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 (next 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:

```
std::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 `std::vector<int>`.

- Entries in maps are *pairs*:

```
std::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.
- Note: The STL `map` type has similarities to the Python dictionary, Java `HashMap`, or a Perl hash, but the data structures *are not the same*. The organization, implementation, and performance is different. In a couple weeks we'll see an STL data structure that is even more similar to the Python dictionary.
- Map search, insert and erase are  $O(\log n)$ . Python dictionaries are  $O(1)$ .

First, 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:

```
#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 << "\t" << 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.

map<string, int> counters

first	second
"run"	1
it → "see"	2
"spot"	1

## 14.4 STL Pairs

The mechanics of using `std::pairs` are relatively straightforward:

- `std::pairs` are a templated `struct` with just two members, called `first` and `second`. *Reminder: 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:

```
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 `std::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:

```
std::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];

++(counters.operator[](s)) //equivalent
```

- 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 `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 `find` member function of the map class does this for us. For example:

```
m.find(key);
```

where `m` is the map object and `key` is the search key. It returns a map iterator:

If the key is in one of the pairs stored in the map, `find` returns an iterator referring to this pair.

If the key is not in one of the pairs stored in the map, `find` returns `m.end()`.

## 14.9 Map Insert

- The prototype for the map `insert` member function is:

```
m.insert(std::make_pair(key, value));
```

`insert` returns a pair, but not the pair we might expect. Instead it is pair of a map iterator and a bool:

```
std::pair<map<key_type, value_type>::iterator, bool>
```

- The `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 *existing pair* in the map and the bool value `false`.
  - If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the *newly added pair* in the map and the bool value `true`.

## 14.10 Map Erase

Maps provide three different versions of the erase member function:

- `void erase(iterator p)` — erase the pair referred to by iterator `p`.
- `void erase(iterator first, iterator last)` — erase all pairs from the map starting at `first` and going up to, but not including, `last`.
- `size_type 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
- In C++11, the first two versions instead return an iterator pointing to the next valid element, just like vector or list erase would.

## 14.11 Exercise

Re-write the `word_count` program so that it uses `find` and `insert` instead of `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.?

### 14.13 Generating and Evaluating Ideas

- Most importantly, play with examples! Can you develop a strategy for solving the problem? You should try any strategy on several examples. Is it possible to map this strategy into an algorithm and then code?
- Try solving a simpler version of the problem first and either learn from the exercise or generalize the result.
- Does this problem look like another problem you know how to solve?
- If someone gave you a partial solution, could you extend this to a complete solution?
- What if you split the problem in half and solved each half (recursively) separately?
- Does sorting the data help?
- Can you split the problem into different cases, and handle each case separately?
- Can you discover something fundamental about the problem that makes it easier to solve or makes you able to solve it more efficiently?
- Once you have an idea that you think will work, you should evaluate it: will it indeed work? are there other ways to approach it that might be better / faster? If it doesn't work, why not?

### 14.14 Mapping Ideas Into Code

- How are you going to represent the data? What is most efficient and what is easiest?
- Can you use classes to organize the data? What data should be stored and manipulated as a unit? What information needs to be stored for each object? What operations (beyond simple accessors) might be helpful?
- How can you divide the problem into units of logic that will become functions? Can you reuse any code you're previously written? Will any of the logic you write now be re-usable?
- Are you going to use recursion or iteration? What information do you need to maintain during the loops or recursive calls and how is it being "carried along"?
- How effective is your solution? Is your solution general? How is the performance? (What is the order notation of the number of operations)? Can you now think of better ideas or approaches?
- Make notes for yourself about the logic of your code as you write it. These will become your *invariants*; that is, what should be true at the beginning and end of each iteration / recursive call.

### 14.15 Getting the Details Right

- Is everything being initialized correctly, including boolean flag variables, accumulation variables, max / min variables?
- Is the logic of your conditionals correct? Check several times and test examples by hand.
- Do you have the bounds on the loops correct? Should you end at  $n$ ,  $n - 1$  or  $n - 2$ ?
- Tidy up your "notes" to formalize the invariants. Study the code to make sure that your code does in fact have it right. When possible use assertions to test your invariants. (Remember, sometimes checking the invariant is impossible or too costly to be practical.)
- Does it work on the corner cases; e.g., when the answer is on the start or end of the data, when there are repeated values in the data, or when the data set is very small or very large?

## 14.16 Problem Solving Strategies

Here is an outline of the major steps to use in solving programming problems:

1. Before getting started: study the requirements, carefully!
2. Get started:
  - (a) What major operations are needed and how do they relate to each other as the program flows?
  - (b) What important data / information must be represented? How should it be represented? Consider and analyze several alternatives, thinking about the most important operations as you do so.
  - (c) Develop a rough sketch of the solution, and write it down. There are advantages to working on paper first. Don't start hacking right away!
3. Review: reread the requirements and examine your design. Are there major pitfalls in your design? Does everything make sense? Revise as needed.
4. Details, level 1:
  - (a) What major classes are needed to represent the data / information? What standard library classes can be used entirely or in part? Evaluate these based on efficiency, flexibility and ease of programming.
  - (b) Draft the main program, defining variables and writing function prototypes as needed.
  - (c) Draft the class interfaces — the member function prototypes.

These last two steps can be interchanged, depending on whether you feel the classes or the main program flow is the more crucial consideration.
5. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
6. Details, level 2:
  - (a) Write the details of the classes, including member functions.
  - (b) Write the functions called by the main program. Revise the main program as needed.
7. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
8. Testing:
  - (a) Test your classes and member functions. Do this separately from the rest of your program, if practical. Try to test member functions as you write them.
  - (b) Test your major program functions. Write separate “driver programs” for the functions if possible. Use the debugger and well-placed output statements and output functions (to print entire classes or data structures, for example).
  - (c) Be sure to test on small examples and boundary conditions.

The goal of testing is to incrementally figure out what works — line-by-line, class-by-class, function-by-function. When you have incrementally tested everything (and fixed mistakes), the program will work.

### Notes

- For larger programs and programs requiring sophisticated classes / functions, these steps may need to be repeated several times over.
- Depending on the problem, some of these steps may be more important than others.
  - For some problems, the data / information representation may be complicated and require you to write several different classes. Once the construction of these classes is working properly, accessing information in the classes may be (relatively) trivial.
  - For other problems, the data / information representation may be straightforward, but what's computed using them may be fairly complicated.
  - Many problems require combinations of both.