CSCI-1200 Data Structures — Fall 2025
Lecture 16 – Associative Containers (Maps), Part 1

Review from Lecture 15

- Error handling strategies
- Basic exception mechanisms: `try/throw/catch`
- Functions & exceptions, constructors & exceptions

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

16.1 STL Maps: Associative Containers

- STL maps store pairs of "associated" values.
- We will see several examples today, in the next lab, and in next lecture:
  - 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:

  ```
  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.

16.2 Counting Word Occurrences

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

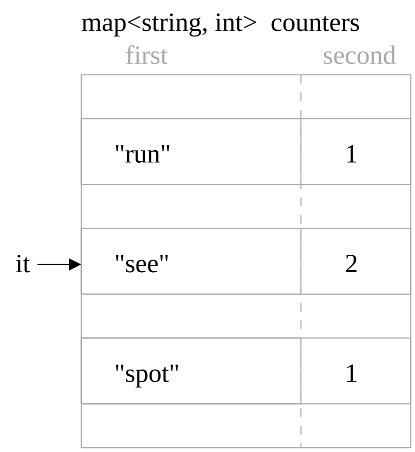
```
#include <iostream>
#include <map>
#include <string>

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;
}
```

16.3 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.
- Note: Python dictionaries are *not* ordered!

16.4 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: a* `struct` *is basically a wimpy* `class` *and in this course you aren't allowed to create new structs. You should use classes with private variables instead – to practice good object-oriented programming design.*
- 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!

16.5 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 `operator[]` is that the key is ALWAYS in the map afterwards.

16.6 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).`

16.7 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?

16.8 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()`.

16.9 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`.

16.10 Map Erase

Maps provide three different versions of the erase member function:

- `iterator erase(iterator p)` — erase the pair referred to by iterator `p`, and return an iterator pointing at the next pair (or end).
- `iterator erase(iterator first, iterator last)` — erase all pairs from the map starting at `first` and going up to, but not including, `last`. Also returns iterator pointing at the next pair (or end).
- `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

16.11 Exercise

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

16.12 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.?

16.13 A "Homework 6" from the Past: Inverse Word Search

Let's flip the classic word search problem and instead *create the board* that contains the specified words! We'll be given the grid dimensions and the set of words, each of which must appear in the grid, in a straight line. The words may go forwards, backwards, up, down, or along any diagonal. Each grid cell will be assigned one of the 26 lowercase letters. We may also be given a set of words words that should *not* appear anywhere in the grid. Here's an example:

```
4 4
+ arts
+ arid
+ east
+ rest
- ear
- at
- sit
```

r	e	a	a
e	a	r	r
s	s	i	t
t	t	d	s

e	r	a	a
a	e	r	r
s	s	t	i
t	t	s	d

In the middle above, is an *incorrect* solution. Though it contains the 4 required words, it also contains two of the forbidden words. The solution on the right is a fully correct solution. This particular problem has 8 total solutions including rotations and reflections.

```
5 3
+ echo
+ baker
+ apt
+ toe
+ ore
+ eat
+ cap
```

e	c	h	o	o
b	a	k	e	r
a	p	t	o	e

```
3 3
+ ale
+ oat
+ zed
+ old
+ zoo
```

```
7 5
+ avocado
+ magnet
+ cedar
+ robin
+ chaos
+ buffalo
+ trade
+ lad
+ fun
- ace
- coat
```

16.14 Generating Ideas

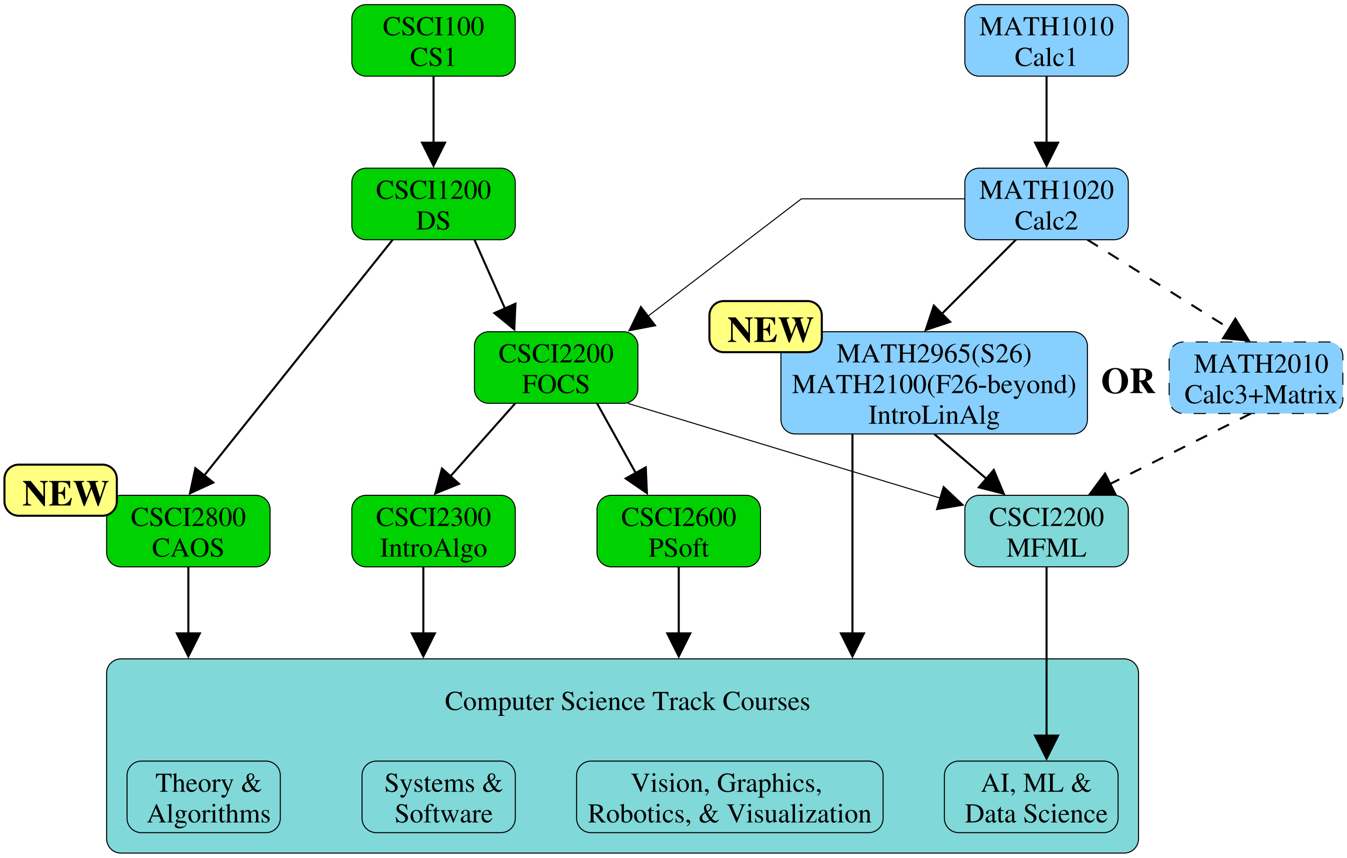
- If running time & memory are not primary concerns, and the problems are small, what is the simplest strategy to make sure all solutions are found. Can you write a *simple* program that tries *all possibilities*?
- What variables will control the running time & memory use of this program? What is the big O notation in terms of these variables for running time & memory use?
- What incremental (baby step) improvements can be made to the naive program? How will the big O notation be improved?

16.15 Mapping Ideas to Code

- What are the key steps to solving this problem? How can these steps be organized into functions and flow of control for the main function?
- What information do we need to store? What C++ or STL data types might be helpful? What new classes might we want to implement?

16.16 Getting the Details Right

- What are the simplest test cases we can start with (to make sure the control flow is correct)?
- What are some specific (simple) corner test cases we should write so we won't be surprised when we move to bigger test cases?
- What are the limitations of our approach? Are there certain test cases we won't handle correctly?
- What is the maximum test case that can be handled in a reasonable amount of time? How can we measure the performance of our algorithm & implementation?


CSCI100
CS1
MATH1010
Calc1
CSCI1200
DS
MATH1020
Calc2
CSCI2200
FOCS
NEW
MATH2965(S26)
MATH2100(F26-beyond)
IntroLinAlg
OR
MATH2010
Calc3+Matrix
NEW
CSCI2800
CAOS
CSCI2300
IntroAlgo
CSCI2600
PSoft
CSCI2200
MFML
Computer Science Track Courses
Theory &
Algorithms
Systems &
Software
Vision, Graphics,
Robotics, & Visualization
AI, ML &
Data Science