CSCI 262
Data Structures

12 – Maps

Introducing

MAPS

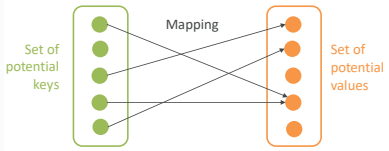
Map

- An abstract data type for associating *keys* with *values*
  - Keys must be unique, value can be anything
  - Similar to *sets* (and often built on them)
    - The map stores sets of *pairs* or *associations*
    - The pair first value is the key, determines uniqueness
- Also known as a Dictionary
- Also known as an *associative array*

For the Mathematically Inclined

Mathematically, a map is a *partial function*

- Relates keys in one domain to values in another domain
- Each key maps to one and only one value
- However, values can be mapped to multiple keys
- *Partial* because we don't map *all* possible keys



Example

A map of strings to strings, storing words → definitions.

Word	Definition
data	individual facts, statistics, or items of information
structure	anything composed of parts arranged together in some way; an organization
algorithm	a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor
…	…

Keys Values

Definitions from *Dictionary.com*

Example

Product database: a map of strings to *tuples*, storing SKUs (product id codes) → product descriptions, prices, etc:

SKU	Description	Color	Price	Unit
427-WHT-100-A	Widgets, white	White	47.99	box
437-RED-100-A	Thingamajigs, red	Red	47.99	box
5190-FOO-66X	Misc. doodads	Black	12.49	pack of 6
…	…	…	…	…

Keys Values

Types of Maps

Just like sets, we have two kinds:

- Ordered maps
  - Items are stored in key order, retrievable in key order
  - Keys must be *comparable*
  - Typically implemented using binary search trees
- Unordered maps
  - Items are stored in no particular order
  - Typically faster than ordered maps
  - Implemented using hashtables

Map Operations

A Map does all of these efficiently:

- **Get** a value associated with a key (if in map)
- **Put** a key/value pair into map
- Remove a key/value pair from map
- Update the value associated with a key
- Determine if the map contains a key

STL Maps (Ordered)

```
#include <map>
template <class K, class V> class map
```

Method summary:

```
at(K key)                  // get value associated with key; throws exception
                           // if not found
insert(pair<K,V> entry)    // put a key/value pair into map
emplace(K key, V value)    // put a key/value pair into the map
erase(K key)               // remove key/value pair from map
find(K key)                // get iterator to entry
count(K key)               // count matching entries
size()                     // number of entries
empty()                    // true if no entries
operator[](K key)          // get and put and update (returns a reference to
                           // value associated with key; creates default entry if
                           // not found)
```


STL Maps Methods

There's a lot to cover here.

We'll dive into the more important methods in a moment.

First…

Interlude

PAIR

STL Pair

`pair` is an STL template class designed for one purpose: to hold two objects.

```
#include <utility>
template <class A, class B> class pair
```

public **member variables** (not methods):

```
first    // first element
second   // second element
```

Pair Usage - Creation

Verbose:

```
pair<type1, type2> p;
p.first = obj1;
p.second = obj2;
```

Quicker:

```
auto p = make_pair(obj1, obj2);
```

Sneaky quick way, when a pair is expected, e.g. as arg:

```
{ obj1, obj2 }
```


Pair Usage – Extracting Values

```
void foo(pair<type1, type2> p) {
    type1 a = p.first;
    type2 b = p.second;
    …
}
```


Pair Example

```
void print_pair(pair<int, string> p) {
    cout << p.first << ": "<< p.second << endl;
}

int main() {
    auto p1 = make_pair(17, "hello");
    print_pair(p1);
    print_pair( {42, "goodbye"} );

    return 0;
}
```


Okay, back to

MAPS

STL Map Methods – Getting

You can get values associated with a key several ways:

- `at(key)`
  - Returns a reference to value, *if key exists in map*
  - Throws exception if key not in map
- `find(key)`
  - Returns an iterator to key, value pair if in map
  - Returns end iterator otherwise (compare with .end())
- `[key]`
  - **Always returns a reference to a value:**
    - Value associated with key, if already in map
    - If not in map, creates entry in map using default for value!
    - Should not be used to test for containment!!!

Could *handle* the exception, but we don't cover that in 262.

STL Map Getting Example

```
map<string, string> m = {
    {"cat", "meow"},
    {"dog", "woof"}
};

cout << m["dog"];          // output: woof
cout << m.at("cat");       // output: meow
cout << m["frog"];         // output: (blank)
                           // frog now in map!
cout << m.at("turtle");    // exception!
```

Note initializer list syntax for maps.

STL Map – Testing for Containment

Verbose but fast:

```
map<string, string>::iterator it = m.find("bunny");
if (it != m.end()) {
      cout << it->second << endl;
} else {
      cout << "No bunny!" << endl;
}
```

it->second is the same as (*it).second

Less verbose, but no access to iterator:

```
if (m.find("bunny") != m.end()) { … }
```

or:

```
if (m.count("bunny") > 0) { … }
```


STL Map - Putting

Again, several choices:

- `m.insert( { "snake", "hiss" } )`
  - Parameter is a pair object
  - **Will not** overwrite/update existing entry
- `m.emplace("snake", "hiss")`
  - More flexible parameters – pair or key, value
  - **Will not** overwrite/update existing entry
- `m["snake"] = "hiss"`
  - **Will** overwrite/update existing entry

Map Example

```
map<string, int> lengths;
lengths.emplace("apple", 5);
lengths.emplace("orange", 6);
lengths.emplace("pear", 4);
…
cout << "Ask me a word: " << endl;
string s;
cin >> s;
if (lengths.count(s) > 0) {
      cout << s << " has " << lengths[s];
      cout << " letters." << endl;
} else {
      cout << "I do not know the word \"" << s << "\".";
      cout << endl;
}
```


Another Map Example

```
map<string, int> frequencies;
ifstream fin("text.txt");
while (!fin.eof()) {
      string w;
      fin >> w;
      frequencies[w]++;
}

for (auto entry: frequencies) {
      cout << entry.first << ": ";
      cout << entry.second << endl;
}
```

Note: range-based for loop and iterators point to pair objects!

Performance Note

- Finding an entry in a map: very fast
- Getting/putting an entry: depends on size of key and value (copies are made!!!)
  - E.g.

```
map<int, vector<string>> m;
// fill m with entries, with long vector values

vector<string> v = m[42];        // copy made
v.push_back("expensive");
m[42] = v;                       // copy made

// better alternative
m[42].push_back("quick");        // no copies made
```


Up Next

- Reading for Monday: Chapter 15.3
- Friday, March 1
  - Lab 7 – Sets & Maps
  - APT 3 assigned
- Monday, March 4
  - Hashtables