Maps, Lexicons, and Sets

Written by Julie Zelenski and Jerry Cain.

All of the containers we've studied so far—**Vector**, **Grid**, **Queue**, **Stack**—are examples of **sequential containers**: They're sequential in the sense that the client has complete or near complete control over the order in which the items are stored. Our **Map** template, on the other hand, is an example of an **associative container**: We control how we associate keys with values, but we have less control over how the key-value pairs are stored on our behalf.

Map Primer

```
template <typename K, typename V>
class Map {
public:
   Map();
  ~Map();

   bool isEmpty();
   int size();

   void put(K key, V value);
   bool containsKey(K key)
   V get(K key);
   V& operator[](K key);
   void remove(K key);
};
```

Finish reading through Chapter 5 by the end of Wednesday, as your second assignment, which also goes out on Wednesday, will exercise all things **Queue**s, **Stack**s, **Map**s, **Lexicon**s, and **Set**s.

Our **Map**—the one defined in the CS106 libraries—operates much like the Java **HashMap**, although there are some key differences:

- The second template parameter can be set to virtually anything at all, but the first template parameter must respond to infix **<**. Primitive types like **int** and **double**, of course, can be compared using **<**, so they're perfectly acceptable **Key** types. And the C++ **string**, even though it's a class, responds to infix **<**, so you'll often see **string** used as keys as well. In the rare situation where you'd like to key your **Map** on a custom data type, your custom data type needs to define **<** (via **operator<**). We'll soon see how to do that.
- The **get** method returns a **copy** of the value attached to the provided key, but **operator[]** returns a **reference** to the associated value. The latter method overloads array indexing so that statements like **sunetIDs.put(4041554, "poohbear")** and **sunetIDs[4041554] = "poohbear"** do the same thing (the motivation being that array notation supports traditional assignment via **operator=**, and is therefore easier to understand.) The distinction with **operator[]**—which is the method the **Map** class must implement to overload the

meaning of **[]**—is a subtle one, in that it returns a reference to the very **string** behind the scenes where **"poohbear"** should be assigned.

- The **get** method returns the **V** 's default value if the key isn't present (and that default value is **0**, **0.0**, **false**, **'\0'** for primitives, and default-constructed objects for classes). If the default value can't be uniformly treated as a sentinel—that is, a if a return value of 0 doesn't tell you whether the key is present or absent, you might guard against the call to **get** by first calling **containsKey**. **operator[]** operates much the same way when the supplied key is missing. More details soon.

Here is a simple function that compiles a histogram of all the words appearing in the referenced **ifstream** and stores those words and their frequencies in the referenced **Map**:

```
static void countWordsInFile(Map<string, int>& map, ifstream& infile) {
   while (true) {
      string word;
      infile >> word; // skips whitespace
      if (infile.fail()) break;
      if (map.containsKey(word)) // if we've seen this word
         map.put(word, map.get(word) + 1); // incr value by 1, update
      else
         map.put(word, 1); // add first occurrence of this word
   }

   cout << "Found " << map.size() << " unique words." << endl;
}
```

The above intentionally avoids the use of array notation so you understand how to code without it. But because the **Map** overloads **[]**, it's possible to rewrite **countWordsInFile** this way:

```
static void countWordsInFile(Map<string, int>& map, ifstream& infile) {
   while (true) {
      string word;
      infile >> word;
      if (infile.fail()) break;
      map[word]++;
   }

   cout << "Found " << map.size() << " unique words." << endl;
}
```

When word already appears in map, **++** is levied directly against a reference to the relevant value. Even though we don't know where the associated value lives, we're momentarily granted access to it, through a reference, so that we can increment it in place without any copying or reinsertion.

When a word is encountered for the very first time, there is, of course, no entry in the map. If **operator[]** notices that the supplied key isn't present, it creates an entry for that key, maps it to a default-constructed value (in this case, a 0) and returns a reference to that 0.

Execution then proceeds as if the key were present all along, and the **++** would promote the newborn 0 to a 1. That's what we want to happen.

Map Iteration

It's easy to lookup a value if we know the key, but nothing we've discussed so far allows us to iterate over all of a **Map**'s key-value pairs. Fortunately, the **Map** class defines some additional directives so that it plays well with the C++ **for** loop construct. If, for instance, we need to publish the contents of our frequency map, as compiled by the **countWordInFile** function above, to a text file, we could do this:

```
static void publishMapToFile(Map<string, int>& map, ofstream& outfile) {
   for (const string& key: map) {
      outfile << key << ": " << map[key] << endl;
   }
}
```

As far as examples go, it's a pretty short one, but it illustrates the general idiom you'll use from time to time when you need to search or other access the data within a **Map** when the search can't be framed in terms of a specific key. If, for instance, you need to search the same **Map** to return all those words that appear a certain number of times or more, you might go with the following:

```
static Vector<string> getCommonWords(Map<string, int>& map, int threshold) {
   Vector<string> commonWords;
   for (const string& key: map) {
      if (map[key] >= threshold)
         commonWords.add(key);
   }

   return commonWords;
}
```

One feature of **Map** iteration worth mentioning: Iteration visits the keys in natural increasing order. The **Map** relies on the key's behavior around **<** to internally organize key/value pairs, and a nice side effect of that organization is that keys are surfaced in natural order (increasing order for numbers, in lexicographical order for **string**s, etc.) during iteration.

Larger Example: Precompiling Anagram Sets

Here's a larger example (courtesy of Keith Schwarz) that uses a **Map** to partition all of the English words into sets—where each set contains those words that are anagrams of one another. And because all words in any given anagram set have the same letter distribution, we elect to key these sets on the sorted character distribution, also expressed as a **string**. The process of compiling the anagram information will reveal, for example, that **"nastier"** and **"retinas"** are in the same set, and the **Vector<string>** containing both will be keyed by **"aeinrst"**.

The program I have in mind is console-based, and interacts with the user is a way that's consistent with the following output:

```
Grouping all words that are anagrams of one another... [done]
Enter some characters [or just hit return to quit]: retains
Anagrams of "retains":

   1.) "anestri"
   2.) "nastier"
   3.) "ratines"
   4.) "retains"
   5.) "retinas"
   6.) "retsina"
   7.) "stainer"
   8.) "stearin"

Enter some characters [or just hit return to quit]: insert
Anagrams of "insert":

   1.) "estrin"
   2.) "inerts"
   3.) "insert"
   4.) "inters"
   5.) "niters"
   6.) "nitres"
   7.) "sinter"
   8.) "triens"
   9.) "trines"

Enter some characters [or just hit return to quit]: preamble
Anagrams of "preamble":

   1.) "preamble"

Enter some characters [or just hit return to quit]: trace
Anagrams of "trace":

   1.) "caret"
   2.) "carte"
   3.) "cater"
   4.) "crate"
   5.) "react"
   6.) "recta"
   7.) "trace"
```

There are a few ways to implement this. My approach here is to pre-compute all anagram classes, catalog them into a **Map<string, Vector<string> >**, and then consult this **Map** as the user enters words like **"insert"**, **"preamble"**, and **"trace"**.

Let's first concern ourselves with the population of the **Map**. Here's my **compileAnagramMap** function, which accepts a reference to a presumably empty **Map** and iterates over a **Lexicon** of all of the English words. Each word is marshaled and appended to the correct **Vector<string>** value, the key of which is produced by a helper routine called **characterSort**, which is really just a selection sort on **string**s—

e.g. **characterSort("nastier")** and **characterSort("retinas")** each return "**aeinrst**".

```
static string characterSort(string word) { // the copy is intentional
    for (size_t lh = 0; lh < word.size(); lh++) {
        int smallest = lh;
        for (size_t rh = lh + 1; rh < word.size(); rh++) {
            if (word[rh] < word[smallest])
                smallest = rh;
        }
        swap(word[lh], word[smallest]); // swap is in <algorithm>
    }

    return word;
}

static void compileAnagramMap(Map<string, Vector<string> >& anagrams) {
    cout << "Grouping all words that are anagrams of one another... ";
    Lexicon english("EnglishWords.dat");
    for (const string& word: english) {
        string key = characterSort(word);
        anagrams[key].add(word);
    }

    cout << "[done]" << endl;
}
```

You'll notice I'm relying on a **Lexicon** class, even though this is the first you're seeing it. Fortunately, the **Lexicon** class is trivial, and you can intuit how one interacts with a **Lexicon** (exported by "**lexicon.h**") by just looking at some code that uses it. (One note: one can iterate over the words in the **Lexicon**, and doing so surfaces all of its words in alphabetical order. The data structure backing the **Lexicon** is a fascinating one, and we'll discuss its implementation in a few weeks.)

The code that prompts the user to enter a collection of characters (English word or not) and retrieve anagrams is more straightforward:

```
static void printAnagrams(const string& letters, Vector<string>& anagrams) {
    cout << "Anagrams of \"" << letters << "\":" << endl << endl;
    for (size_t i = 0; i < anagrams.size(); i++) {
        cout << "   " << (i + 1) << ".) \"" << anagrams[i] << "\"" << endl;
    }
    cout << endl;
}

static void queryAnagramMap(Map<string, Vector<string> >& anagrams) {
    while (true) {
        string letters = trim(
            getLine("Enter some characters [or just hit return to quit]: "));
        if (letters.empty()) return;
        string key = characterSort(letters);
        if (anagrams.containsKey(key)) {
            printAnagrams(letters, anagrams[key]);
        } else {
```

```
            cout << "There are no anagrams of \"" << letters << "\"" << endl;
        }
    }
}
```

And, as was the case with the **queen-safety** example, we declare the master copy of our data structure—in this case a **Map<string, Vector<string> >**—in the **main** function, and share a reference to that master copy with each of **compileAnagramMap** and **queryAnagramMap**.

```
int main() {
    Map<string, Vector<string>> anagrams;
    compileAnagramMap(anagrams);
    queryAnagramMap(anagrams);
    return 0;
}
```

As an added exercise, if we'd like to identify the largest anagram set, we can just iterate over the keys of the map, keeping track of the largest set ever encountered along the way. If there are multiple largest sets, we can return any single one of them.

```
static Vector<string>
getLargestAnagramSet(Map<string, Vector<string> >& anagrams) {
    Vector<string> largestAnagramSet;
    for (const string& key: anagrams) {
        if (anagrams[key].size() > largestAnagramSet.size()) {
            largestAnagramSet = anagrams[key];
        }
    }
    return largestAnagramSet;
}
```

Set Primer

The only significant container class left to discuss is the **Set**, which does its darndest to emulate the notion of a mathematical set. On the one hand, the **Set** is like a **Map** with keys but no companion values, and that analogy gets you very far. But it also overloads some common operators (**+**, **-**, *****, **+=**, **-=**, ***=**) to support set addition, union, subtraction, and intersection. The **Set**, like the mathematical set, is unordered and doesn't allow duplicates. Since it doesn't need to preserve an order, it's free to **impose** one and store our elements in such a way that insertion, deletion, and search can run very, very quickly.

Like the **Map**, the **Set** requires its keys cooperate when compared to each other using **<**. Iteration is supported via the **for** loop and the **Set**'s members are presented in logically increasing order (again, not because it's required to, but because it's electing to.)

Here is the condensed interface for the **Set** template:

```
template <typename T>
class Set {
public:
   Set();
  ~Set();

   bool isEmpty();
   int size();

   void add(T elem);
   void remove(T elem);
   bool contains(T elem);
   bool isSubsetOf(Set& other);

   Set operator+(T elem)
   Set operator+(Set& other);
   Set operator*(Set& other);
   Set operator-(T elem);
   Set operator-(Set& other);
   Set& operator+=(T elem);
   Set& operator+=(Set& other);
   Set& operator*=(Set& other);
   Set& operator—=(T elem);
   Set& operator-=(Set& elem);
};
```

Don't let the syntax in place to overload **+**, **+=**, and so forth intimidate you. The syntax is just quirky C++ needed to overload operators to work on custom data types. As you study the forthcoming examples, you'll come to understand that the operators are just a clean, intuitive way to express set union, intersection, and subtraction.

Set Primer: Happy Numbers

Let's use a **Set** to determine whether a number is **happy**. In recreational mathematics, a number is happy if zero or more manipulations of its digits eventually generate a 1. If a number is 1, then it's happy. Otherwise, a number is happy if the sum of the squares of its digits is itself a happy number.

For example, 44 is happy, as the associated sequence demonstrating happiness is:

$$4^2 + 4^2 = 32$$
$$3^2 + 2^2 = 13$$
$$1^2 + 3^2 = 10$$
$$1^2 + 0^2 = 1$$

Similarly, 139 is happy because

$$1^2 + 3^2 + 9^2 = 91$$
$$9^2 + 1^2 = 82$$

$$8^2 + 2^2 = 68$$
$$6^2 + 8^2 = 100$$
$$1^2 + 0^2 + 0^2 = 1$$

And lest you believe all numbers are happy, consider the number 4:

$$4^2 = 16$$
$$1^2 + 6^2 = 37$$
$$3^2 + 7^2 = 58$$
$$5^2 + 8^2 = 89$$
$$8^2 + 9^2 = 145$$
$$1^2 + 4^2 + 5^2 = 42$$
$$4^2 + 2^2 = 20$$
$$2^2 + 0^2 = 4$$

The digit manipulation of 4 leads through a collection of mutually unhappy numbers whose own digit manipulations leads back to the original 4. These numbers are collectively trapped in a cycle of misery, and are all unhappy numbers.

We can leverage the above and use the CS106 **`Set`** to help identify integer happiness. Here we go:

```
static int computeSquareOfDigitSum(int n) {
   int sum = 0;
   while (n > 0) {
      int digit = n % 10;
      sum += digit * digit;
      n /= 10;
   }

   return sum;
}

static bool isHappy(int n) {
   if (n <= 0) return false; // how can negative numbers be happy?

   Set<int> previouslySeen;
   while (n > 1 && !previouslySeen.contains(n)) {
      previouslySeen.add(n);
      n = computeSquareOfDigitSum(n);
   }

   return n == 1;
}
```

The above implementation assumes that every number, when manipulated as described, eventually leads to 1 if it doesn't get trapped in a cycle like the one we saw with the number 4.

As it turns out, all unhappy numbers are eventually led into the above cycle of unhappiness (according to a **http://mathworld.wolfram.com/HappyNumber.html**). An alternate implementation would just collect all of the base numbers that inform us of a number's happiness into a **Set**. We can manipulate the original number and its successors as needed, knowing they'll eventually lead to either a 1 or one of the numbers in the unhappy number cycle. This second version illustrates a slick use of the **+=** operator and the fact that the comma operator is lower precedence and evaluated from left to right:

```
static bool isHappy(int n) {
    if (n <= 0) return false;
    Set<int> endpoints;
    endpoints += 1, 4, 16, 37, 58, 89, 145, 42, 20;
    while (!endpoints.contains(n)) {
        n = computeSquareOfDigitSum(n);
    }

    return n == 1;
}
```

The **endpoints += 1, 4, 16, 37, 58, 89, 145, 42, 20;** line a shorthand that populates a **Set** with a small number of explicitly listed entries. Save for the 1, all of the numbers are those contained in the unhappy number loop. Our approach here is to just transform the current number over and over until it becomes some number in held by **endpoints**. Then we know the number is happy if and only if it evolved into a 1.

Larger Example: Generating Farey Series

The Farey series of order n, denoted by F_n, is the sorted sequence consisting of all reduced fractions between 0 and 1 with denominators less than or equal to n. The Farey series of order 6, for example, is

$$\frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}$$

We can model a fraction with a simple record type (using the **struct** keyword) as follows:

```
struct fraction {
   int numerator;
   int denominator;
};
```

In C++, records are like classes in that they aggregate related information to help model a more complex type. The one key difference is that everything within a **struct** definition is **public**, whereas everything within a class is **private** unless exposed via the **public** access modifier. We'll have much more to say about records and classes in a few weeks, but this example calls for a crash course in the record and how to define one.

Let's implement

```
Vector<fraction> generateFareySeries(int n);.
```

We'll add all relevant **fraction**s to a **Set<fraction>**, and then iterate over it to construct a sorted **Vector<fraction>** with exactly the same information.

For this to work, we need to overload **operator<** to work with **fraction**s. Because the **fraction** is a custom data type, there's no way support for **<**. We can, however, define it ourselves.

We need to make use of a rule we all learned in 7th grade:

$$\frac{a}{b} \text{ is less than } \frac{c}{d} \text{ if and only if ad is less than bc}$$

We overload **<** by defining a top-level **operator<** function that looks like this:

```
static bool operator<(const fraction& one, const fraction& two) {
    return one.numerator * two.denominator < two.numerator * one.denominator;
}
```

The function defining a new flavor of **<** must be called **operator<**, must take two arguments, and *should* return a **bool**. Note that we accept references to **fraction**s! **fraction**s are large enough that we shouldn't make deep copies of them. And we mark the **fraction**s as **const** to ensure **operator<** won't change the **fraction**s being shared with them. (The **const** and the **&** are technically optional, but good programming practice, and professional C++ programmers would use both here.)

While we're overloading operators, I confess that it's also possible to overload **<<** so we can print **fraction**s just as we print **int**s and **string**s. Those who designed C++ wanted developers to be able to integrate their own types into the language as fully as possible, which is why C++ supports operator overloading to the extent that it does.

The implementation of **operator<<** looks like this:

```
static ostream& operator<<(ostream& os, const fraction& f) {
    os << f.numerator; // assume for our example that fractions are positive
    if (f.denominator > 1) {
        os << "/" << f.denominator;
    }
    return os;
}
```

The prototype, if it's to print **fraction**s at all, must have this prototype. The **ostream&** return value is required if we're to be able to daisy chain calls to **<<**, as with:

```
fraction pi = { 22, 7 };
cout << "The constant pi is equal to " << pi << "." << endl;
```

Our implementation constructs **all** fractions (with a denominator of n or less) between 0 and 1—both reduced and unreduced—and inserts them in such an order that only the reduced fractions remain. One subtle feature of the **`Set`**'s **`add`**, **`operator+`** and **`operator+=`** operations is that requests to insert a previously inserted item are ignored. So, if you insert 2/3 first and 4/6 later, the request to insert 4/6 is ignored.

```
Vector<fraction> generateFareySeries(int n) {
   Set<fraction> fractions;
   for (int denominator = 1; denominator <= n; denominator++) {
      for (int numerator = 1; numerator < denominator; numerator++) {
         fraction f = { numerator, denominator };
         fractions += f; // could have called add, but illustrating += instead
      }
   }

   Vector<fraction> series;
   for (const fraction& f: fractions) series.add(f);
   return series;
}
```

A simple possible test framework might looks like this:

```
int main() {
    for (int n = 1; n <= 6; n++) {
        cout << "F(" << n << ") is ";
        cout << generateFareySeries(n) << endl;
    }
    return 0;
}
```

`main`'s output is always:

```
F(1) is {}
F(2) is {1/2}
F(3) is {1/3, 1/2, 2/3}
F(4) is {1/4, 1/3, 1/2, 2/3, 3/4}
F(5) is {1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5}
F(6) is {1/6, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6}
```

(Our **`Vector`** template can be printed in its entirety using **`<<`**, and the implementation of **`operator<<`** relevant to **`Vector<fraction>`** itself relies on the **`operator<<`** presented earlier.)