

More ADTs: Scanners, Maps, and Sets

Written by Julie Zelenski.

Scanner

The **Scanner** class provides functionality for dividing a string into separate words or tokens. A scanner comes in quite handy when doing any sort of string or file processing. Here's a few ideas to get you thinking about its uses:

- breaking apart a command entered at the console (e.g. "**turn left**", or "**find price < 5**")
- dividing a document into component words for textual analysis, such as a word frequency count, preparing an index, or building a concordance
- separating a file pathname into its components (e.g. the sub-directories within **/usr/class/cs106x/www/handouts/01-Course-Information.pdf**)
- evaluating an arithmetic expression, such as **3 + 4 * 7**

The basic public interface of the **Scanner** class is listed below:

```
class Scanner {  
    public:  
        Scanner();  
        ~Scanner();  
  
        void setInput(string str);  
        void setInput(ifstream& infile);  
        string nextToken();  
  
        bool hasMoreTokens();  
        void saveToken(string token);  
  
        enum spaceOptionT { PreserveSpaces, IgnoreSpaces };  
        void setSpaceOption(spaceOptionT option);  
        spaceOptionT getSpaceOption();  
        // other advanced options excerpted for clarity  
};
```

The idiomatic pattern for using a **Scanner** is to create a new object, set the string to be scanned, and then enter a loop that calls **nextToken** while **hasMoreTokens** returns **true**. Here's a simple example that uses the **Scanner** to reports the number of tokens entered in a user's response:

```

void CountTokens()
{
    cout << "Please enter a sentence: ";
    string response = GetLine();

    int count = 0;
    Scanner scanner;
    scanner.setInput(response);

    while (scanner.hasMoreTokens()) {
        scanner.next_token();
        count++;
    }

    cout << "You entered " << count << " tokens." << endl;
}

```

The next example shows finding the longest token read from a file. Note that the Scanner can also be constructed around an **ifstream** object and tokenizes the stream of characters within the referenced file.

```

string FindLongestToken(ifstream& infile)
{
    string longest = "";
    Scanner scanner;

    scanner.setInput(infile);
    while (scanner.hasMoreTokens()) {
        string token = scanner.next_token();
        if (token.length() > longest.length()) {
            longest = token;
        }
    }

    return longest;
}

```

There are various options that you can set on a scanner to determine how it handles certain special tokens. Most of these options are only needed for advanced use, but one that you commonly manipulate is the **SpaceOption**, which controls whether white space tokens are returned from **ReadToken** or skipped over. The default is for spaces to be returned, but you can cause them to be ignored by changing the option:

```

scanner.setSpaceOption(Scanner::IgnoreSpaces);

```

This is handy when you are interested in processing only the non-space tokens. Note that the **spaceOptionT enum** is defined within the Scanner class, so when referring to those values as a client, we fully specify the name including the scope qualification **Scanner::**. This tells the compiler to look for the name within the **Scanner** class instead of assuming it is top-level entity. This is like the **string::npos** constant.

Associative Containers

So far we've discussed the **Vector**, **Grid**, **Stack**, and **Queue** storage classes. The classes are sometimes collectively referred to as *sequential containers*, because elements are stored and retrieved in order of the sequence in which they were inserted. While these containers are quite useful, we also need containers that are a bit fancier. How about being able to quickly lookup an element, automatically maintaining the elements in sorted order, or being able to easily combine two collections into a new result? The *associative containers*, such as **Map** and **Set**, use information about the value of the elements being stored to arrange them in a manner to support these operations very efficiently. Sound good? Read on!

Map

The **Map** class is an incredibly nifty container. It provides the abstraction of a *map*, which is the dorky-computer-scientist name for a collection of key-value pairs. The client associates a value with a key and can lookup by key to get the associated value back. Other names used for this data structure include *symbol table*, *dictionary*, or just plain *table*. One cool feature of a map is that adding an entry and looking up by key are typically implemented very efficiently, so that even when the map has thousands or millions of entries, these operations are practically instantaneous. As a client, we might scratch our heads and marvel at this, but even with no clue how it works, we can still repeat the benefits of this stellar performance.

You can do lots of handy things with a map, here are just a few ideas:

- A dictionary: words are the keys, associated value is the word's definition
- Access: keys are student id numbers, value is student's transcript
- A document index: words are keys, value is a set of pages where the word is referenced
- A symbol table for a compiler: variable name mapped to memory location
- DMV: key is license plate number, value is the registration information for the vehicle

The public interface of the **Map** class template is outlined below. Note that the keys are always of type **string** (there is a good reason for this restriction, we will discuss this when we talk about implementation), but the values can be of whatever type the client needs to store. The interface is written using the template placeholder **ValueType** that is filled by the client when declaring and allocating a map.

```

template <typename ValueType>
class Map {

    public:
        Map(int sizeHint = DefaultStartingSize);
        ~Map();

        int size();
        bool isEmpty();

        ValueType get(string key);
        void put(string key, ValueType value);
        void remove(string key);
        bool containsKey(string key);
        Iterator iterator();
};

```

When calling the **put** member function, the client provides the key and value and the map stashes the pair for later access. Attempting to add a value for key that already has an entry in the map will replace the previous value with the newly added one. Calling the **remove** member function will remove the entry for a key.

The **get** member function allows the client to retrieve the value associated with a given key. This function has a slightly unusual design with regard to its return value. In the case where **get** finds the key in the map, it can return the associated value easily enough. However, what if the key being looked up isn't in the map? What, then, should the function return? Ideally, the returned value would some sort of "not found" sentinel that reports to the client that the key isn't contained in the map. But that sentinel needs to be of type **ValueType**, which is just a placeholder.

The map is written as a template and doesn't make any assumptions about the true nature of **ValueType**. It might be a string, an integer, a structure, a pointer, or something else. Each of these types has its own unique value for what might make a good sentinel and it isn't the same for all of them. There is no easy way for a generic version of **get** to decide which is the right kind of sentinel value to return when the key is not found. Thus, the member function is written to raise an error if the client attempts to retrieve the value for a key that doesn't exist. It becomes the client's responsibility to first call **containsKey** to verify that a key does indeed have an entry in the map before attempting to get its value.

Here is a sample use that uses a map to record the frequencies of words occurring in a file:

```
void MapTest(istream & in)
{
    Map<int> map;
    string word;
    while (true) {
        in >> word;
        if (in.fail()) break;
        if (map.containsKey(word)) // if already have 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 map also provides a shorthand syntax for setting and retrieving values by overloading the square brackets operator, e.g. **map[key]**. The argument within the square brackets is not an integer index, but a string key, and the expression evaluates to the value associated with that key in the map. If there is no entry for the key in the map, using this expression creates a new entry for key in map. The use of square brackets returns the value by reference, which allows you to use the expression to either get the value or to change it. The code below is a rewrite of the previous example, using square brackets instead of **get/put**.

```
void MapTest(istream & in)
{
    Map<int> map;
    string word;
    while (true) {
        in >> word;
        if (in.fail()) break;
        if (map.containsKey(word)) {
            map[word]++;
        } else {
            map.put(word, 1);
        }
    }

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

```

Although you may find the use of square brackets a little bit funky at first, it is fast becoming the de facto syntax for accessing maps (in C++ as well as other programming languages) and once you become accustomed to it, you may even find it tidy and convenient.

You'll note that the map assumes that keys are of **string** type. This can be seen as a bit of a restriction, but conveniently, it's not difficult to convert most data into a string representation to make it suitable for use as a key. For example, consider the problem of

storing student information for the Axxess system. The desired key is the student id number, which is of integer type, so you might think you were out of luck. However, one quick call to the **StringToInteger** function converts the id to a string equivalent, which makes a fine key:

```
struct student {
    string first, last;
    int idNum;
    string emailAddress;
};

void AddToMap(Map<student>& map, student student)
{
    // convert id number to string to use as key in map
    string idAsString = IntegerToString(student.idNum);
    map.put(idAsString, student);
}

string GetNameForId(Map<student>& map, int idNum)
{
    string idAsString = IntegerToString(idNum); // convert to lookup
    if (map.containsKey(idAsString)) {
        student found = map.get(idAsString);
        return found.first + " " + found.last;
    } else {
        return "";
    }
}
```

You might also see it is limiting that there can be only one value associated with a given key. Attempting to enter another value for an existing key replaces the previous value. However, there is an easy way to get a one-to-many mapping if you desire it: simply store a vector as the value for each key and add values to that vector. For example, consider a map that manages the index for a document. Each key in the map is a word and the associated value is the pages where that word appears. The list of pages can be stored using a vector of integers. Here is a bit of code that shows how you would add a new entry into such a map:

```
// This map represents a book index. The keys are words in the book,
// the associated value is vector of page numbers where that word appears
// For example, the word "binky" might appear on pages 3, 5, 8, and 10.
// This function is given a partially complete index, a word, and
// a page # and updates the index to show that this word is referenced
// on that page.

void AddReference(Map<Vector<int> > & index, string word, int pageNum)
{
    index[word].add(pageNum);
    // wow, this little one-liner does a lot!
    // the [] retrieves either the existing vector already contained
    // in map, or creates a new empty vector and adds to map under key
    // that vector is then asked to add a new page number
}
```

Note that use of a nested template—a map where the values are vectors of integers. The syntax is a little crufty (be careful about the need for the space between the two closing angle brackets), but hopefully the expressive power that enables building these sophisticated structures makes it all worthwhile!

Iterating over a map

A map provides key-value access. If you have the key, you can look up the associated value, or enter a new value for that key. However, it's not obvious how to browse the map entries when you don't know the key you want. For example, consider finding the word with the most occurrences in the frequency map or identifying all students with last names beginning with Z or just printing all the entries in any map. The techniques that worked for the sequential containers don't apply here—the map is not a linear collection, the elements are not stored in order, and you cannot run a loop over the collection's size calling **get** or **dequeue** to get each element in turn.

Instead, the map allows the client to get access to the entries by providing an iterator. An **iterator** is a helper object that works in conjunction with a container object (in this case, the map) to step through the elements contained in the collection. When you want to browse the entries in a map, you ask the map to create an iterator for you, and you use that **iterator** object to visit the keys from the map one by one in a loop. Because the **iterator** is tightly integrated with its container, it is defined as a nested class, i.e. within the scope of the collection class. The name for the map's iterator is **Map<ValueType>::Iterator**. An **Iterator** has two public member functions: a predicate **hasNext** that returns true if there are more elements remaining in the iteration and a **next** function that returns the next element in the iteration. The idiomatic iteration loop is similar to using a scanner: while there are still elements remaining, retrieve the next one. Consider this loop that prints the keys within the map:

```
void PrintKeysFromMap(Map<int> & map)
{
    Map<int>::Iterator itr = map.iterator();
    while (itr.hasNext())
        cout << itr.next() << endl;
}
```

Each key in the map will be accessed once and only once in an iteration, but the keys can be visited in any order (e.g. the map iterator does not guarantee to return keys in alphabetical order or in the order they were inserted).

You'll notice that the iterator returns only the key. If you also need access to the value, it's just a member function call away. For example, to print the frequency entries, this code does the trick:

```
void PrintFrequencies(Map<int> & map)
{
    Map<int>::Iterator itr = map.iterator();
```

```

while (itr.hasNext()) {
    string key = itr.next();
    cout << key << " = " << map[key] << endl;
}
}

```

Set

Next up, we have our **Set** class. The **Set** data structure isn't covered in Chapter 4 like the rest of the containers are, but it's a useful enough data structure—particularly in the context of the Word Ladder assignment—that I'm including it in this handout.

A set is used to store a collection of unique elements and serves as the analog to the sets you know from mathematics. Some of its key features are:

- Unlike a vector, set elements are not stored linearly, and are not assigned/retrieved by index. The sequence in which elements are added is not of consequence.
- Sets do not store duplicate elements. Attempting to add an element that the set already contains is a no-op.
- A set can quickly determine if it contains a given element.
- Sets provide nifty set-level operations like equals, subset, and union/intersect/subtract that do comparisons and combinations of one entire set with another. These operations are typically implemented quite efficiently.
- The set needs to compare elements to see if they are identical, to check for membership, enforce non-duplication, and so on. It will attempt to use the built-in relational operators to compare elements, but if that is not legal or appropriate for your needs, you have the option of supplying a comparison callback function when creating the set. (More on how this works later in this handout)

What are sets good for? All sorts of things! The set of synonyms for a given word, the set of email addresses on a mailing list, the set of requirements to graduate, the set of options on a car, you name it! You can throw a bunch of items into a set just to coalesce duplicates. The set is especially snazzy when you need set-level operations, like intersecting the pizza topping preferences of your friends to find a pizza they would all enjoy.

The public interface of the **Set** class is outlined below. The function pointer passed to the constructor used for the comparison callback is discussed in more detail below.

```

template <typename ElemType>
class Set {

public:
    Set(int (cmpFn)(ElemType, ElemType) = DefaultCmp);
    ~Set();

    int size();
    bool isEmpty();

```

```

    void add(ElemType element);
    void remove(ElemType element);
    bool contains(ElemType element);
    bool equals(Set& otherSet);

    bool isSubsetOf(Set& otherSet);
    void unionWith(Set& otherSet);
    void intersect(Set& otherSet);
    void subtract(Set& otherSet);

    Iterator iterator();
};

```

First, off, here is a trivial example that shows creating the set of unique characters from a string:

```

void SetTest(string str)
{
    Set<char> set;
    for (int i = 0; i < str.length(); i++)
        set.add(str[i]);
    cout << "There are " << set.size() << " unique chars.";
}

```

Now, let's use sets to help ferret out just how random the random library is. The idea is to count how long a sequence of random numbers you can generate before you get a repeat. We use a set to store the random sequence and check each number for membership in the set to determine if it has previously been used.

```

void RandomTest()
{
    Set<int> seenSoFar;
    while (true) {
        int num = RandomInteger(1, 100);
        if (seenSoFar.contains(num)) break;
        seenSoFar.add(num);
    }

    cout << seenSoFar.size() << " unique numbers before repeat.";
}

```

Similar to the map, the set provides an iterator to give access to the elements. You ask the set to create an iterator and then use the standard iteration loop to walk through the elements one by one.

Here is a function that prints a set of doubles using the iterator:

```

void PrintSet(Set<double>& set)
{
    Set<double>::Iterator itr = set.iterator();
    while (itr.hasNext())
        cout << itr.next() << " ";
}

```

The **Set** iterator guarantees that it will visit the set elements in sorted order, where "sorted" is determined by the natural ordering of the elements (e.g. numeric or alphabetic order) or according to client's wishes specified by the comparison function (discussed below).

Sets really shine when your algorithm requires the high-level operations that work on entire sets, such as testing for set equality or combining two sets to compute the intersection or difference. Want to schedule a meeting for everyone in your project group? Intersect the set of times each member has available. Want to check if a student can graduate? Verify that the set of required courses is a subset of the set the student has taken. Compound database queries are naturally expressed using set operations. You want to find people who like skiing or Bartok but not Thai food and who live in your dorm? Use intersection, union, and difference to construct the result.

Here's a simple example that is given two people and their set of friends and enemies and will use set operations to construct a guest list for a joint part that contains all friends and excludes either's enemies:

```
struct person {
    string name;
    Set<string> friends, enemies;
};

Set<string> MakeGuestList(person& one, person& two)
{
    Set<string> result = one.friends; // start by copying one's friends
    result.unionWith(two.friends); // join with two's friends
    result.subtract(one.enemies); // take out one's enemies
    result.subtract(two.enemies); // and two's enemies, too
    return result;
}
```

The **Set** class is unique in that it needs more information about the type of element being stored than the other template containers. Vectors, stacks, queues, and maps just take the client's data and stash it somewhere to be returned later. Those containers never examine the elements or do any operations on them. But a set needs to be able to compare two elements to determine if they are the same (this is needed to avoid entering duplicates, to check if an element is contained, to compute the intersection, and so on). But the set is written as a template, when means the element type is merely a placeholder, and the "real" element type isn't known until the client allocates the set. So how then can the **Set** class do its job without the full information?

As a default, the set tries to use the built-in relational operations (e.g. **==**) to compare two elements. For some element types (such as **int** or **string**), this will work just fine. However, let's say the client tries to create a set containing student **structs**, something like this:

```
struct student {
    string first, last;
```

```

    int idNum;
    string emailAddress;
};

int main()
{
    Set<student> students; // compile error!
}

```

The above code will generate a compile error that is reported something like this:

```

Error : illegal operands 'student' == 'student'
(point of instantiation: 'main()')
(instantiating: CmpUsingBuiltinOps<student>(student, student))
cmpfn.h line 25 if (one == two) return 0;

```

The compiler tries to instantiate the set of students using the default set comparison which attempts to compare two student **structs** using `==`, yet `==` does not have a valid meaning for structure types.

What's happening here is that there isn't a specification for what it means for two student records to be the same. Does every field have to match or just the name or id? The client who defined the **struct** is the only one who truly knows what it means for two of those **structs** to be considered equal. How can the client inform the set how to compare two student **structs**? The client writes a function that takes two **structs** and returns the result of comparing the two. The client then passes that function to the set constructor. The new set will call the client's function it needs to compare two elements to determine their equality. Here's how to fix the non-compiling example from above:

```

// comparator function for two student structs
// compares the id field to determine equality/ordering
int CompareStudents(student one, student two)
{
    if (one.idNum < two.idNum) return -1;
    else if (one.idNum == two.idNum) return 0;
    else return 1;
}

int main()
{
    Set<student> students(CompareStudents);
}

```

The client defines the function **CompareStudents** that takes two **student structs** and returns the comparison result based on the id number. A comparison function is expected to return `-1` (or some other negative number) if the first parameter is "less than" the second, `0` if the two are equal, and `+1` (or some other positive number) if the first is "greater than" the second.

This function **CompareStudents** is then passed to the constructor when the set is created. This configures the set to skip the default comparison (e.g. `==`) and instead call

the client's supplied function whenever the set needs to compare two elements. This kind of function is called a *callback* function because the set calls back out to the client.