STL Maps

Maps in STL are one of the most useful components within the STL. If this template is the only component of STL that you use, it will improve your programming vocabulary significantly. I’m not saying this so you ignore the other parts of STL, but to merely demonstrate how important maps are. In other contexts, maps are known as hash tables or hash maps.

Before we get into a discussion of what maps are and exactly how to use them, let’s discuss the general concept in a simplified context.

C Arrays

Almost all programming languages have some concept of an array, which is comprised of the same or related kind of elements. We will discuss them in the context of C because they are virtually identical to arrays in C++, a language that we have some knowledge. STL has the vector template, which has many advantages over arrays, so most C++ programmers will use this template in most situations.

Successful C programmers know that the ‘metal’ of the computer is lurking only a layer or two underneath their code. This brings an intuitive sense of what their code will be doing at the assembly or even the machine code level. C++, by using various kinds of abstraction (classes or specialization using templates), makes it possible to use higher level constructs without having to know what is happening at lower levels. It is true that many very good C++ programmers are aware of what is happening at a lower level, but it is not necessary – for example, you do not need to know the details of how vectors are implemented to use them effectively.

C arrays are guaranteed to be contiguous in memory. For example, consider the following C declaration:

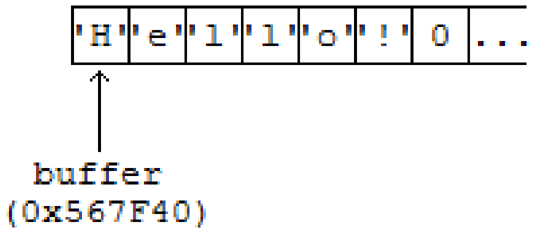
```
char buffer[512];
```

This allocates space enough for 512 characters that are side by side in memory. From an assembly/machine code point of view, this allows for very efficient manipulation of the array because access can be accomplished through pointer arithmetic.

If we set the contents of the array using the following line of code:

```
strcpy(buffer, “Hello!”);
```

…we have set up the memory in the following way:



This supposes that the address at which the compiler has set aside the memory starts at `0x567F40`. Every C programmer intuitively knows that this is going on when an array is declared. He or she is very comfortable manipulating the array through the use of the pointer value of the first element and applying pointer arithmetic.

When we take a look at the bigger picture, we have set up what is known as an **association**: we can look upon this as set of value pairs an unsigned integer (the index of the array) to a character (the value of array at the index). Using array notation this is clear:

```
buffer[0] == ‘H’;
buffer[1] == ‘e’;
buffer[2] == ‘l’;
buffer[3] == ‘l’;
buffer[4] == ‘o’;
buffer[5] == ‘!’;
buffer[6] == 0;
```

The associations are:

```
0 <--> ‘H’
1 <--> ‘e’
2 <--> ‘l’
3 <--> ‘l’
4 <--> ‘o’
5 <--> ‘!’
6 <--> 0
```

On an abstract level, C arrays are associations with some limitations:

1. The first of the two values is always an unsigned integer. You can use negative numbers as indexes but this is invariably a sign of very poor and crash-prone software (or hacking).
2. The values of the index are contiguous because they are closely tied to the memory layout. If you want to set out an association between 15 and a character, you must set up a relationship between 0 and 14, even if you don't assign the values in these pairs immediately.
3. In the most common usage, the size of an array is set at compile time. This gives the careless programmer the chance to manipulate values that are beyond the bound of the array. This is the biggest source of program crashes and security vulnerabilities in software. The careful and secure use of arrays is a skill that not every C programmer possesses, and one of the reasons why there is so much crash-prone and insecure software in existence.
4. Because arrays are contiguous in memory, they are inefficient when dealing with operations like insertion and deletion.

In C++ we can build something more abstract and flexible.

Key-Value Pairs

Ideally, we want to link to pieces of data together into what are known as **key-value pairs**. In the case of C arrays, we associated an unsigned integer to another data type, so we could look up a **value** based on an index, or the **key**. What we really want, and what we get with C++, is the ability to map pairs based on keys of a wide variety of types. In the STL, we have two types of maps: the `map` and `multimap` template, and in the upcoming C++ standard, two additional maps: `unordered_map` and `unordered_multimap`.

The basis of these classes is the container template, `std::pair`. When defining a pair, we specify the key data type (or class) and the value data type (or class). In this declaration, we can map the names of the students in the class to a number that represents that student's grade:

```
#include <string>
#include <map> //pair template defined here
#include <utility> //needed for make_pair(…)


//Using the values constructor:
std::pair<std::string, unsigned int> cpp_grade1(“Fred Flintsone”, 47);

//Using the make_pair(…) function
std::pair<std::string, unsigned int> cpp_grade2;
cpp_grade2 = std::make_pair(“Barney Rubble”, 97);
```

In this example, we have two pairs constructed by using the `std::pair` constructor and by the template function `make_pair(…)`. We can access the values of the pairs by using the two members: `first` (the key) and `second` (the value):

```
cout << cpp_grade1.first << “ earned “ << cpp_grade1.second <<
        “ in C++.\n”;

cout << cpp_grade2.first << “ earned “ << cpp_grade2.second <<
        “ in C++.\n”;
```

This code outputs:

```
Fred Flintstone earned 47 in C++.
Barney Rubble earned 97 in C++.
```

This template is the basis of all map classes in STL.

std::map

This template stores pairs with a unique key, and the value of the keys can be placed in an order. This is because the values are stored in a data structure that keeps track of the order of the keys. In the simplest form, the key is orderable, like a number or a string, and the declaration looks like this:

```
std::map<std::string, unsigned> grades;
```

As long as a pair is of the same classes or data types, we can insert pairs into a map by using the insert(…) member function:

```
grades.insert(cpp_grade1);
```

We can also add to the map by using something resembling C array notation:

```
grades[“Barney Rubble”] = 95;
```

This notation will look very strange to a C programmer for two reasons: the argument to the array notation (square brackets) is a `std::string` and it has not been initialized prior to its usage. If the map does not contain a pair with the included key, one is silently added to the map and if the key does exist, the value can be retrieved or updated:

```
grades[“Wilma Flintstone”] = 85; //creates a new entry in the map
                                 //with the key “Wilma Flintstone”

grades[“Wilma Flintstone”] = 95; //updates the value in the pair
                                 //with the key “Wilma Flintstone”
```

If we try to read a value with a key that does not exist in the map, a pair is added to the map with the key and a default value.

```
//the key “Betty Rubble” does not exist in the map
cout << grades[“Betty Rubble”] << endl;

//…and now it does, since we have tried to read it.
```

In order to insert pairs into a map, the key values must be orderable: numbers come with a natural order and strings have the notion of alphanumeric order which is implemented in the string class. If you try to key a map based on your own class and it is not orderable, the insertions in the map will fail:

```
class MyClass
{
//various data members and operation
//but no sense of ordering.
};


std::map<MyClass, int> map_test;

MyClass obj;
map_test[obj] = 986; //Compile error!  Cannot order MyClass objects.
```

You can create an ordering for objects of your classes and including this in your declaration of the map as an argument to the template, but this is a somewhat advanced technique that you can research in a good STL reference.

Another possible source of unpredictable behaviour in the map template using `float`s, `double`s or `long double`s as key values. These data types in most implementations of C++ are approximations and two values that are notionally equal might not be in practice. Another issue that can cause problems is that these data types have the concept of `NaN` – or **N**ot **a N**umber. This happens when an overflow or underflow condition has occurred in a floating point type variable.

These issues will lead to unpredictable results when using a floating point data type as a key value in a map.

STL Iterators

Because using array notation will silently add elements in to a map, it is impractical to use this method to find all the entries in a map. This is where iterators come in handy. The iterator object can be dereferenced with the same syntax as with pointers, to access the `pair` values of the map entry:

```
for(std::map<std::string, unsigned>::iterator it = grades.begin();
    it != grades.end(); ++it)
{
      // the it dereferenced is a reference to
      //a std::pair< std::string, unsigned>

      cout << it -> first << “ earned “ << it -> second
           << “ in C++.\n”;
}
```

This loop will output the map entries in ascending order of the keys.

Iterators are the recommended way of accessing the pairs in maps. They are also used by the STL template functions and member functions. The most commonly used member functions include:

`end()`	Returns an iterator to one past the end.
`rbegin()`	Returns a reverse iterator to the reverse beginning.
`rend()`	Returns a reverse iterator to one past the reverse end.
`empty()`	Tests whether container is empty.
`size()`	Returns the container size.
`max_size()`	Returns maximum size.
`operator[]`	Accesses a pair or creates a new pair if it does not exist.
`insert()`	Inserts an element.
`erase()`	Erase elements in a map.
`swap()`	Swaps content between two maps.
`clear()`	Clears all content in a map.
`find()`	Gets the iterator to an element.

Consult a good STL reference for the template functions that are available.

std::multimap

The std::multimap template is an ordered container, very similar to the std::map, except that there can be pairs with duplicated keys. For this reason the array type access is not implemented for this class and the `find()` function returns an iterator to a multimap of all pairs where the key matches the argument to the call to `find()`.

std::unordered_map and std::unordered_multimap

The latest and soon to be ratified C++ standard (C++0X) include these two templates that are very similar to the `std::map` and `std::mutimap` templates. The difference between these two new classes and the existing one is that there is no concept of ordering of the key values. This allows the programmer to set up maps where there is no inherent notion of order for the keys.

In order to use these templates, you need to ensure that the compiler you are using support these new classes. Check the documentation for your compiler.

	Ordered?	**Duplicated Keys?**
`std::map`	Yes	No
`std::multimap`	Yes	Yes
`std::unordered_map`	No	No
`std::unordered_multimap`	No	Yes