Programming with Haiku

Lesson 2

Written by DarkWyrm

In our first lesson, we learned about how to generalize type handling using templates and some of the incredibly flexible data containers in Standard Template Library. The list, deque, and vector containers are good at storing data for sequential access, but there is more to the STL than merely these three. We will study the other major containers and the differences and similarities among them. There will also be some coverage of a portable string class which makes string manipulation much easier than with the standard C functions. There is a lot of information in this lesson. Go slowly and don't be afraid to re-read it once or twice, if necessary.

C++ Strings

Before we tackle these other containers, it is necessary to know about the Standard C++ library, which uses concepts from the STL but provides other useful development tools. One of these is a general purpose string class.

Working with strings using the standard C functions such as `strcpy()` and `strstr()` is a lot of hassle. The wonderful BString class in the Support Kit eliminates this hassle, but so does the C++ `string` class. Most of the time you will find the BString class preferable because it is generally faster, can do far more, and is integrated into the Haiku API, but there are a few methods which the C++ strings have for which BString has no equivalent.

```
string substr(size_t pos = 0, size_t n = npos);
```

`substr()` returns a substring starting with the character at `pos` and ending at `n`. `npos` is a static value which equals the greatest size the string could be. Needless to say, it's silly to call this and use both of the default values because this just returns the entire string.

```
size_t find_first_not_of (const string &str, size_t pos = 0) const;
size_t find_first_not_of (const char *s, size_t pos, size_t n) const;
size_t find_first_not_of (const char *s, size_t pos = 0) const;
size_t find_first_not_of (char c, size_t pos = 0) const;
size_t find_last_not_of (const string& str, size_t pos = npos) const;
size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
size_t find_last_not_of (const char* s, size_t pos = npos) const;
size_t find_last_not_of (char c, size_t pos = npos) const;
```

These two functions search the string object for the first (or last) occurrence of a character that is not one of the ones specified in the first parameter of the call. Here is an example of how it can be used.

```
std::string myString = "/boot/home/config/settings";
size_t pos = myString.find_first_not_of('/');
```

In this example, the `char` version of find_first_not_of() is used to find the first character which is not a slash. `pos` in this example has a value of 1. Used in series, this could be a way to break up a file path into a series of folder names without resorting to using `strtok()`.

C++ strings are also in the `std` namespace and may be referred to as `std::string` to differentiate them from regular C-style strings. They are used with the header `<string>`. We won't be using C++ strings very much, but it's best to know about their existence because they are used fairly often on other platforms.

Associative Containers

The associative containers provided to us by the Standard Template Library are necessary when it is slow or impossible to use integers to look up data in a container. While it is possible to manually search arrays, vectors, or other sequential containers using a loop, it is potentially slow to do so. For an occasional lookup, this is just fine, but if you have to repeatedly look up information this way, this can impose a serious performance hit in your code.

map

Header: `<map>`

The `map` container centers around what is known as a key-value pair, i.e. a lookup value and the data that goes with it. Declaring a `map` object involves the types for both the key and value, like this:

```
map<BString, int32> myMap;
```

This declaration creates a map which uses strings to look up integers. It is common to see maps being used to look up data using strings as keys. By using `std::string` or BString as the key type, it is possible to use a regular C-style string to get the value.

```
printf("The value for %s is %d\n", "Some value", myMap["Some value"]);
```

The only caveat to using a `map` is that the key values are expected to be unique. The items in the `map` are actually another STL container: `pair`. The `pair` container merely associates two types together. Accessing the two types paired together is done via the properties `first` and `second`.

set

The `set` container is a similar to a `map` except that the values are also the keys and they are sorted. Like `map`, all items in a `set` are expected to be unique.. It is not often used because there are other containers which are more flexible. The main purpose for using a `set` is guaranteed fast insertions and fast lookups. The implementation of the `set` container is normally pretty complicated so it can provide these guarantees.

multimap, multiset

These are versions of `set` and `map` which do not require the keys to be unique. Calling the `find()` method still returns only one item with the key given to it, but these containers have an additional method, `equal_range()` which returns two iterators to a range that provides all of the elements with the specified key value.

Container Adapters

In addition to the containers provided by the Standard Template Library, there are a few container adapters. These use a regular STL container to do the heavy lifting for a specialized interface.

queue

A `queue` adapter normally sits on top of a `deque` container. Conceptually, items go in the back of the queue and come out the front, much like getting into the ticket line at the movies. This is sometimes called FIFO, or first-in, first-out processing. It provides the methods `front()`, `back()`, `push_back()`, and `pop_front()`.

priority_queue

The `priority_queue` adapter works a lot like the `queue` adapter with one significant difference: the first item out is not the first item in. Instead, the first item out is the one with the highest priority. The two containers that can be used as the backend – to do all of the heavy lifting – are the `vector`, which is the default, and the `deque`.

stack

The `stack` can sit on top of a `deque`, `vector`, or `list`. It is used for LIFO processing, as in last-in, first-out. It can be likened to a stack of trays in a cafeteria: the last tray placed on the stack is also the first one to be taken off.

Common STL Container Methods

There are a lot of different containers provided by the STL, and sometimes it's hard to remember which is which. Thankfully, there is a common set of methods for all of them.

```
iterator begin();
const_iterator begin() const;
```

> Returns an iterator which refers to the first element in the container. Because the associative containers sort their elements in ascending order, `begin()` will return the item with the lowest value.

```
iterator end();
const_iterator end() const;
```

> Returns an iterator which refers to an element which is past the end of the elements in the container – this is not the same as the last element. This method is most often used in loops, particularly `for` loops.

```
iterator rbegin();
const_iterator rbegin() const;
iterator rend();
const_iterator rend() const;
```

> These methods do the same job as `begin()` and `end()` but starting at the end of the container and working toward the beginning. `rbegin()` returns the last element of the container and `rend()` returns an iterator which is before the first element. These methods correspond with using a `reverse_iterator` in a loop instead of a regular one.

```
size_type size() const;
```

> This is the number of objects the container holds.

```
size_type max_size() const;
```

`max_size()` returns the maximum number of objects that the container can hold based on limitations imposed by the system.

```
bool empty() const;
```

Returns true if the container has zero elements.

```
void resize(size_type newSize, T from = T());
```

The container is resized to hold `newSize` elements. If this is fewer than the current number, then the excess elements are dropped. If the new size is greater, then new elements are created from the object passed as the parameter `from`. If none is specified, the default constructor for the container object's type is used. This method is available only to sequential containers like `vector`.

```
reference front();
const reference front() const;
reference back();
const reference back() const;
```

These methods return the first or last element of the container, respectively. This is not the same as the iterator returned by `begin()` or `rbegin()`. These methods are not available to associative containers like `map`.

vector, deque

```
reference operator[size_type index];
const_reference operator[size_type index] const;
```

map

```
T & operator[const key_type &key];
```

Using the array operator on a deque, vector, or map returns the item with the specified index. In the case of map, this is the value at the specified key. If a map does not have a value at the specified key, it is created with an empty value. This operator is available for only these three containers.

```
reference at(size_type index);
const_reference at(size_type index) const;
```

`at()` works just like the array operator with two key differences: it's only available to the `deque` and `vector` containers and if an index is used that is out of the container's bounds, `at()` throws an `out_of_bounds` exception.

```
template class<InputIterator>
void assign(InputIterator first, InputIterator last);
void assign(size_type newSize, const T &from);
```

`assign` is a wonderful way to set items in a sequential container to a value all at once or to copy one container into another. The first version takes iterators from

another container and copies those elements into the owning container starting with `first` through, but not including, `last`. The second version sets all elements to the value specified by `from`. In both cases, the container is resized to `newSize` or the number of elements specified by the range in the iterator version.

```
iterator insert(iterator pos, const T &item);
template <class InputIterator>
void insert(iterator pos, InputIterator first, InputIterator last);
```

vector, deque, and list only

```
void insert(iterator pos, size_type count, const T &item);
```

map and set only

```
pair<iterator, bool> insert(const value_type &item);
```

multimap and multiset only

```
iterator insert (const value_type &item);
```

`insert()` adds another item to a container. This method is common to all STL containers, although each container has an additional form of the call unique to itself. The speed of an item insertion depends on the implementation of the container. For example, adding an item in the middle of a vector is potentially slow, but adding an item to its end is quite fast.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

set, multiset, map, and multimap only

```
size_type erase(const key_type &lookupValue);
```

`erase()` deletes an item from a container. Like `insert()`, performance of this call depends on the implementation of the container.

```
swap(<container to swap with>);
```

This call takes another container of the same type and swaps the items between the two containers. All pointers, references, and other outside data related to the items kept in the two containers remain valid.

```
void clear();
```

In one fell swoop, this call deletes all items in the container, making it completely empty.

```
void push_front(const T &item);
void pop_front();
```

These two calls enable you to add or remove an item from the front of a `deque` or `list`. `pop_front()` not only removes the item from the container, it also deletes the item, too.

```
void push_back(const T &item);
void pop_back();
```

These two calls perform the same operations as the `front` versions, but they operate on the items at the back. However, these calls are available for the `vector` container, as well as `deque` and `list`.

```
key_compare key_comp() const;
```

This returns the object used to compare items in the container. It can either be a pointer to a function or an instance of a class which implements the function call operator. The comparison function tests two objects of the container's type and returns true if the first item is to be less than or to be placed before the second item in the container or false in any other case. This function is only available to associative containers.

```
value_compare val_comp() const;
```

This works just like `key_comp()` except that it returns the function used to compare two values. For the `set` container, this is the same as `key_comp()`. This function is only available to associative containers.

```
iterator find(const key_type &lookupValue) const;
```

Searches the container for the element which matches `lookupValue` and returns an iterator which points to it or `end()` if not found. This function is only available to associative containers.

```
size_type count(const key_type &lookupValue) const;
```

Returns the number of elements in the container which match the lookup value. Although this method is available to all associative containers, it only makes sense to use it on `multiset` and `multimap`, because `map` and `set` require their lookup values to be unique.

```
iterator lower_bound(const  key_type &lookupValue);
const iterator lower_bound(const key_type &lookupValue) const;
iterator upper_bound(const key_type &lookupValue) const;
pair<iterator, iterator> equal_range(const key_type &lookupValue) const;
```

`lower_bound()` returns an iterator which points to the first item which is greater than or equal to the lookup value. `upper_bound()` returns an iterator to the first item which is greater than the lookup value. `equal_range()` returns two iterators, the first of which is the same as `lower_bound(lookupValue)` and the second is the same as `upper_bound(lookupValue)`. Like `count()`, these methods are available to all associative containers, but they only make sense to use with `multiset` and `multimap`.

STL and the Standard Library: So What?

Having taken a whirlwind tour through a lot of different template classes after only a short introduction to namespaces and templates, this might all be a little overwhelming. Not to worry. These don't have to be used as often as you might think. Haiku has a class internal to Tracker called BObjectList which provides all of the ease-of-use of the BList class with the optional ability to have it handle memory management. This covers the need for indexed storage. The `map` and `multimap` containers do really well for random access containers. The remaining containers are more for reference in specialized instances, cross-platform programming, and recognizing them in others' code. They also are necessary when nesting a container within another, such as a `map` of `vectors`.

The same can be said for the Standard C++ library. Some of it will come in handy for Haiku development, and some of it won't be immediately useful. Their usefulness partly depends on how much development you do on other platforms, such as Linux or Windows. If you plan on programming just for Haiku, you probably won't use much of the standard library or the STL, but if you also develop for other platforms, using these will ease the transitions made between platforms.