Programming with Haiku

Lesson 1

Written by DarkWyrm

For those who are reading this after finishing *Learning to Program with Haiku*, we will start with a few concepts to round out our basic knowledge of C++. Today's main topics will be templates and the Standard Template Library. There is a lot of information needed to understand both, so go slowly and carefully.

Templates

Templates are a way of generalizing a class or function to deal with any type of data. They are most often used in creating container classes such as stacks, lists, and others. Taking advantage of this flexibility requires a few small changes to the declaration of the class or function which uses them. A template-based array class could look like this:

```
template <class T>
class ObjectArray
{
public:
                        ObjectArray(const uint32 &size);
                        ObjectArray(const ObjectArray &from);
                        ~ObjectArray(void);

      ObjectArray &     operator=(const ObjectArray &from);
      T &               operator[](int index);

      uint32            GetSize(void);

private:
      T                 *fData;
      uint32            fSize;
};
```

In this instance, the main difference between this and a regular class definition is the `template <class T>` section before the definition. This tells the compiler that any time we see `T`, we are talking about the type used by that class. Note that we can use pretty much any name we like for the type name, such as T, Key, Type, or whatever, but we have to use the name we declare in the template section to refer to the type name. The ObjectArray class has two properties: the number of elements, kept in `fSize`, and a heap-allocated buffer used to hold the individual elements pointed to by `fData`. Without worrying about the implementation, an ObjectArray instance could be used like this:

```
#include <stdio.h>
#include "ObjectArray.h"

int
main(void)
{
      ObjectArray<int32> intArray(10);
      for (uint32 i = 0; i < intArray.GetSize(); i++)
            intArray[i] = intArray.GetSize() - i;

      for (uint32 i = 0; i < intArray.GetSize(); i++)
            printf("intArray[%ld] is %lu\n", i, intArray[i]);
}
```

The key to using class templates is to specify the type in angle brackets. In the above example, we have created an ObjectArray instance which uses 32-bit integers as its type. It does all of its work centered around `int32` objects and does not concern itself with any other type. Once created, it cannot change its type – an `ObjectArray<int32>` cannot be changed into an `ObjectArray<float>` or even an `ObjectArray<int16>`, for example. Also, two ObjectArray objects with different types are treated as completely different types by the compiler, i.e. `ObjectArray<float>` and `ObjectArray<int64>` are not the same type even though they both use the ObjectArray class as a template.

Defining functions in a template-based class isn't difficult. Many references to the class name need to include the type name. Here is the code used to define all of the functions in the above class definition for the ObjectArray class:

```
template <class T>
ObjectArray<T>::ObjectArray(const uint32 &size)
  :   fData(NULL),
      fSize(size)
{
      if (fSize > 0)
            fData = new T[fSize];
}


template <class T>
ObjectArray<T>::ObjectArray(const ObjectArray &from)
  :   fData(NULL),
      fSize(from.GetSize())
{
      if (fSize > 0)
      {
            fData = new T[fSize];
            for (uint32 i = 0; i < fSize; i++)
                  fData[i] = from[i];
      }
}


template <class T>
ObjectArray<T>::~ObjectArray(void)
{
      delete [] fData;
}


template <class T>
ObjectArray<T> &
ObjectArray<T>::operator=(const ObjectArray &from)
{
      if (this == &from)
            return *this;

      delete [] fData;

      fSize = from.GetSize();
      fData = new T[fSize];
      for (uint32 i = 0; i < fSize; i++)
            fData[i] = from[i];
}
```

```
template <class T>
T &
ObjectArray<T>::operator[](int index)
{
	return fData[index];
}


template <class T>
uint32
ObjectArray<T>::GetSize(void)
{
	return fSize;
}
```

Aside from being careful about adding <T> in a few places and the extra typing needed at the top of each function definition, there is one other caveat when using class templates: templated-based function definitions need to be placed in the header instead of the main source file. This is because these are templates for function definitions, not the definitions themselves. The actual definitions are generated at compile time. If these functions were placed in the main source file, the linker would spit back a host of `undefined reference` errors which, needless to say, is more than a little confusing.

Sometimes you don't need the entire class to use a template. Perhaps all that is needed is a function or two which make use of templates. In this instance, all that is needed is to place the template declaration section in front of the return type for the function declaration like this:

```
class MyClass
{
						MyClass(void);
						~MyClass(void);
				void	SomeRegularFunction(int value);

	template<class T> void	SomeTemplateFunction(T item);
};
```

In this case, `SomeTemplateFunction()` is the only part of MyClass which utilizes a template. Just like the functions for our ObjectArray class, it will need to be defined in the header and not in the main source file with the rest of the functions for MyClass. In these instances, only the function templates need to be in the header – the "regular" functions should go in a separate source file as they normally are.

Using Templates: The Standard Template Library

The best use of templates is for data containers. Luckily for us, a group of people took the time and effort to create an entire library of template-based containers called the Standard Template Library, or STL for short. Most of the Haiku API is so well-designed that we do not need to use the STL very often, but it does have a few containers that come in quite handy. Because the containers in the STL are so well-designed, most of the time our use of templates will be limited to using these containers instead of creating classes that use templates.

Any Haiku projects which use the STL need to add an extra library. GCC2 builds of Haiku use the library libstdc++.r4.so. Haiku GCC4 builds link to libstdc++.so for STL usage.

The containers provided by the STL fall into two categories: sequential and associative. The sequential containers are designed for working with items in a list which have a definitive order, such as a list. The BList class provided by the Haiku API is an example of a sequential container. Such containers are optimized for operating on the entire list one item at a time. Associative containers are intended for data which requires seemingly random lookup or by values that are not integers. Probably the most common case for needing an associative container is using a string to look up data. Choosing the specific container to use depends on the kind of work you intend to do with them.

Vector

Header: `<vector>`

The vector class is very similar to an array except that memory allocation is handled automatically. Vectors are good at quickly accessing elements by index, iterating over elements in any order, and adding and removing elements from their end. Adding elements can be slow, but only when it runs out of internal storage and has to allocate more.

Deque

Header: `<deque>`

A deque, usually pronounced "deck," is short for **d**ouble-**e**nded **que**ue. Deques are very similar to vectors except that they are efficient at adding elements at both the beginning and end and their elements are not guaranteed to occupy a contiguous chunk of memory. This has both benefits and drawbacks. Using pointers to access elements of a deque is not safe, unlike elements of a vector, but they are a much better choice for storing large numbers of elements.

List

Header: `<list>`

Lists are best described as a collection of dynamically-allocated items. They are best used for work which involves inserting, removing, and moving items around. One major drawback to using lists is that there is no way to quickly access an arbitrary element – accessing the tenth element in the list requires iterating from the beginning (or other reference point) to that element.

Namespaces

Using STL containers also requires some extra typing because all of them are encapsulated into their own namespace. Namespaces are a way of creating groups of classes, functions, and data types. Often they are used to prevent conflicts between classes in different libraries that have the same name. A namespace is declared like this:

```
namespace myNamespace
{
	int32 foo, bar;
}
```

Accessing elements inside a namespace requires specifying the namespace. For example, all STL containers are inside the `std` namespace. Declaring a vector of `int32` objects would look something like this:

```
// Note that there is no '.h' for this header and others in the STL. It's
// just <vector>.
#include <vector>

std::vector<int32> intVector;
```

The double colon is the scope operator. Specifying items within the same namespace do not require it. Likewise, specifying an item in the global namespace from within another requires a double colon.

```
#include <stdio.h>

bool gSomeFlag = true;

namespace myNamespace
{

int intValue = 1;

void
SomeFunction(void)
{
      // This specifies the gSomeFlag which is in the default (global)
      // namespace
      if (::gSomeFlag)
            printf("myValue is %d\n", intValue);
}

} // end myNamespace
```

Using a lot of items in another namespace can make for a lot of typing, so if you are using a particular namespace quite a lot, you can employ the `using` keyword to eliminate the extra typing.

```
#include <deque>
#include <stdio.h>

// This eliminates the need to add the std:: before each reference
// to deque containers.
using std::deque;

int
main(void)
{
      // Declare our deque to accept integers. Without the using statement
      // above, this would read std::deque<int> myDeque. Unless you're
      // using a lot of these, the using statement isn't really needed.
      deque<int> myDeque;

      // Add one element to the end of the list which has a value of 5
      myDeque.push_back(5);
```

```
	// Print the number of elements in the deque. In this case, we're
	// definitely not playing with a full deque.
	printf("This deque has %d elements\n", myDeque.size());

	return 0;
}
```

The using keyword offers fine-grained control over what requires us to type the namespace. In the above example, we removed the need to specify the namespace whenever we use the deque container. If we were also using the vector container, we would still have to type std::vector<myType> for vector type references. If there are no possibilities for name conflicts and we use many different containers from the STL, then we can make a blanket using declaration which covers everything in the std namespace:

```
using namespace std;
```

Be careful using the using keyword this way. If your code must deal with more than one namespace, it is highly advisable that you use it on a class-by-class basis or, better yet, not use it at all. However, if you are working with a Haiku program which just makes calls to the API and a few things from the STL, then using declarations which speed up your work a bit are just fine.

STL Iterators

The designers of the containers in the STL were wise to attempt to keep the API for all of the containers as similar as possible. One way that they did this was to create iterators. Because not all of the containers lend themselves to using integers to access each element like an array does, iterators are used instead.

Each iterator is more or less a pointer to the element type used by the container. The ++ and -- operators have been overloaded to go to the next or previous element in the container. Using a vector<int> in a for loop would look something like this:

```
#include <stdio.h>
#include <vector>

// This using statement only works for the vector class. It also makes
// the loop code a little more readable.
using std::vector;

int
main(void)
{
	vector<int> myVector;

	// Add a few values to our vector
	myVector.push_back(5);
	myVector.push_back(10);
	myVector.push_back(15);

	// The begin() method will always point to the first item in the
	// container. end() will always point to the last one. This format
	// works for *all* STL containers.
	for (vector<int>::iterator i = myVector.begin();
		i != myVector.end(); i++)
	{
		printf("myVector: %d\n", *i);
	}
}
```

We've seen a number of methods used without any explanation. Sadly, sometimes in the real world the only available documentation for code is the code itself, but it doesn't have to be that way here. These are the methods we've seen so far with vectors, deques, and lists and a few other useful ones. By no means is this an exhaustive list, however.

Method	**Description**
`iterator begin();`	Returns an iterator pointing to the first element in the container.
`iterator end();`	Returns an iterator pointing to the last element in the container.
`size_type size();`	Returns the number of elements in the container. `size_type` is an unsigned integer type.
`void push_back(const T &val);`	Adds an item to the end of the container's list which has the value `val`. Note that this effectively invalidates any existing iterators.
`void pop_back();`	Deletes the last item in the container.
`void push_front(const T &val);`	Adds an item to the beginning of the container's list which has the value `val`. Note that this effectively invalidates any existing iterators. This method is unavailable to the `vector` class.
`void pop_front();`	Deletes the first item in the container. This method is unavailable to the `vector` class.
`void clear();`	Deletes all items in the container.

Tying It All Together: An Example Usage

After seeing these containers, there aren't very many instances where using a BList seems all that advantageous. For example, one instance where using one of these containers would be more efficient than a standard issue BList is making a list of `entry_ref` objects from a directory. Below is an example of how all of the information that we've seen about STL containers can be put to use in a way that makes sense.

```
#include <deque>
#include <Directory.h>
#include <Entry.h>
#include <FindDirectory.h>
#include <Path.h>
#include <stdio.h>

using std::deque;

int
main(void)
{
      // Look up the home folder -- this is much preferable to a
      // hard-coded way of accessing a system path.
      BPath path;
      find_directory(B_USER_DIRECTORY, &path);
      BDirectory dir(path.Path());

      deque<entry_ref> refDeque;

      entry_ref ref;
      while (dir.GetNextRef(&ref) == B_OK)
            refDeque.push_back(ref);

      printf("Contents of the home folder: %s\n", path.Path());
      for (deque<entry_ref>::iterator i = refDeque.begin();
            i != refDeque.end(); i++)
      {
            // Note that because an iterator can be treated like
            // a pointer, we can access each entry_ref's name
            // using the iterator.
            printf("\t%s\n", i->name);
      }
}
```

Going Further

- How could this example be changed to use lists instead of deques?