Standard Template Library

INTRODUCTION TO STL PART 1 - GETTING STARTED

STL stands for Standard Template Library, and is a new feature of C++. We will be presenting some of the basic features of STL in this and subsequent issues. STL may not be available with your local C++ compiler as yet. The examples presented here were developed with Borland C++ 5.0. Third-party versions of STL are available from companies like Object Space and Rogue Wave, and HP's original implementation (which may be obsolete) is available free on the Internet.

To get an idea of the flavor of STL, let's consider a simple example, one where we wish to create a set of integers and then shuffle them into random order:

```
        #include <vector>
        #include <algorithm>
        #include <iostream>

        using namespace std;

        int main()
        {
                vector<int> v;

                for (int i = 0; i < 25; i++)
                        v.push_back(i);

                random_shuffle(v.begin(), v.end());

                for (int j = 0; j < 25; j++)
                        cout << v[j] << " ";
                cout << endl;

                return 0;
        }
```

When run, this program produces output like:

```
        6 11 9 23 18 12 17 24 20 15 4 22 10 5 1 19 13 3 14 16 0 8 21 2 7
```

There's quite a bit to say about this example. In the first place,
STL is divided into three logical parts:

```
        - containers

        - iterators

        - algorithms
```

Containers are data structures such as vectors. They are implemented as templates, meaning that a container can hold any type of data element. In the example above, we have "vector<int>", or a vector of integers.

Iterators can be viewed as pointers to elements within a container.

Algorithms are functions (function templates actually) that operate on data in containers. Algorithms have no special knowledge of the types of data on which they operate, meaning that an algorithm is generic in its application.

We include header files for the STL features that we want to use. Note that the headers have no ".h" on them. This is a new feature in which the .h for standard headers is dropped.

The next line of interest is:

```
        using namespace std;
```

We discussed namespaces in earlier newsletter issues. This statement means that the names in namespace "std" should be made available to the program. Standard libraries use std to avoid the problem mentioned earlier where library elements (like functions or class names) conflict with names found in other libraries.

The line:

```
        vector<int> v;
```

declares a vector of integers, and then:

```
        for (int i = 0; i < 25; i++)
                v.push_back(i);
```

adds the numbers 0-24 to the vector, using the push_back() member function.

Actual shuffling is done with the line:

```
        random_shuffle(v.begin(), v.end());
```

where v.begin() and v.end() are iterator arguments that delimit the extend of the list to be shuffled.

Finally, we display the shuffled list of integers, using an overloaded operator[] on the vector:

```
        for (int j = 0; j < 25; j++)
                cout << v[j] << " ";
        cout << endl;
```

This code is quite generic. For example, we could change:

```
        vector<int> v;
```

to:

```
        vector<float> v;
```

and fill the vector with floating-point numbers. The rest of the code that shuffles and displays the result would not change.

One point to note about STL performance. The library, at least the version used for these examples, is implemented as a set of header files and inline functions (templates). This structure is probably necessary for performance, due to the internal use of various helper functions (for example, begin() in the above example). Such an architecture is very fast but can cause code size blowups in some cases.

We will be saying more about STL in future issues. The library is not yet in widespread use, and it's too early to say how it will shake out.

INTRODUCTION TO STL PART 2 - VECTORS, LISTS, DEQUES

In the previous issue we introduced the C++ Standard Template Library. STL is a combination of containers used to store data, iterators on those containers, and algorithms to manipulate containers of data. STL uses templates and inline functions very heavily.

In this issue we'll talk about some of the types of containers that are available for holding data, namely vectors, lists, and deques.

A vector is like a smart array. You can use [] to efficiently access any element in the vector, and the vector grows on demand. But inserting into the middle of a vector is expensive, because elements must be moved down, and growing the vector is costly because it must be copied to another vector internally.

A list is like a doubly-linked list that you've used before. Insertion or splicing of subsequences is very efficient at any point in the list, and the list doesn't have to be copied out. But looking up an arbitrary element is slow.

A deque classically stands for "double-ended queue", but in STL means a combination of a vector and a list. Indexing of arbitrary elements is supported, as are list operations like efficiently popping the front item off a list.

To illustrate these notions, we will go through three examples. The first one is the same as given in the last newsletter issue, and shows how a vector might be used to store a list of 25 numbers and then shuffle them into random order:

```
        #include <vector>
        #include <algorithm>
        #include <iostream>

        using namespace std;

        int main()
        {
                vector<int> v;

                for (int i = 0; i < 25; i++)
                        v.push_back(i);

                random_shuffle(v.begin(), v.end());

                for (int j = 0; j < 25; j++)
                        cout << v[j] << " ";
                cout << endl;

                return 0;
        }
```

With lists, we can't use [] to index the list, nor is random_shuffle() supported for lists. So we make do with:

```
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
        list<int> v;

        for (int i = 0; i < 25; i++)
                v.push_back(i);

        //random_shuffle(v.begin(), v.end());

        for (int j = 0; j < 25; j++) {
                cout << v.front() << " ";
                v.pop_front();
        }

        cout << endl;

        return 0;
}
```

where we add elements to the list, and then simply retrieve the element at the front of the list, print it, and pop it off the list.

Finally, we present a hybrid using deques. random_shuffle() can be used with these, because they have properties of vectors. But we can also use list operations like front() and pop_front():

```
#include <algorithm>
#include <iostream>
#include <deque>

using namespace std;

int main()
{
        deque<int> v;

        for (int i = 0; i < 25; i++)
                v.push_back(i);

        random_shuffle(v.begin(), v.end());
```

```
                for (int j = 0; j < 25; j++) {
                        cout << v.front() << " ";
                        v.pop_front();
                }

                cout << endl;

                return 0;
        }
```

Which of vectors, lists, and deques you should use depend on the application, of course. There are several additional container types that we'll be looking at in future issues, including stacks and queues. It's also possible to define your own container types.

The performance of operations on these structures is defined in the standard, and can be relied upon when designing for portability.

INTRODUCTION TO STL PART 3 - SETS

In the last issue we talked about several STL container types, namely vectors, lists, and deques. STL also has set and multiset, where set is a collection of unique values, and multiset is a set with possible non-unique values, that is, keys (elements) of the set may appear more than one time. Sets are maintained in sorted order at all times.

To illustrate the use of sets, consider the following example:

```
        #include <iostream>
        #include <set>

        using namespace std;

        int main()
        {
                typedef set<int, less<int> > SetInt;
                //typedef multiset<int, less<int> > SetInt;

                SetInt s;

                for (int i = 0; i < 10; i++) {
                        s.insert(i);
                        s.insert(i * 2);
                }

                SetInt::iterator iter = s.begin();

                while (iter != s.end()) {
                        cout << *iter << " ";
                        iter++;
                }

                cout << endl;

                return 0;
        }
```

This example is for set, but the usage for multiset is almost identical. The first item to consider is the line:

```
        typedef set<int, less<int> > SetInt;
```

This establishes a type "SetInt", which is a set operating on ints, and which uses the template "less<int>" defined in <function> to order the keys of the set. In other words, set takes two type arguments, the first of which is the underlying type of the set, and the second a template class that defines how ordering is to be done in the set.

Next, we use insert() to insert keys in the set. Note that some duplicate keys will be inserted, for example "4".

Then we establish an iterator pointing at the beginning of the set, and iterate over the elements, outputting each in turn. The code for multiset is identical save for the typedef declaration.

The output for set is:

```
        0 1 2 3 4 5 6 7 8 9 10 12 14 16 18
```

and for multiset:

```
        0 0 1 2 2 3 4 4 5 6 6 7 8 8 9 10 12 14 16 18
```

STL also provides bitsets, which are packed arrays of binary values. These are not the same as "vector<bool>", which is a vector of Booleans.

INTRODUCTION TO STL PART 4 - MAPS

In the previous issue we talked a bit about STL sets. In this issue we'll discuss another data structure, maps. A map is something like an associative array or hash table, in that each element consists of a key and an associated value. A map must have unique keys, whereas with a multimap keys may be duplicated.

To see how maps work, let's look at a simple application that counts word frequency. Words are input one per line and the total count of each is output.

```
        #include <iostream>
        #include <string>
        #include <map>

        using namespace std;

        int main()
        {
                typedef map<string, long, less<string> > MAP;
                typedef MAP::value_type VAL;

                MAP counter;

                char buf[256];

                while (cin >> buf)
                        counter[buf]++;

                MAP::iterator it = counter.begin();

                while (it != counter.end()) {
                        cout << (*it).first << " " << (*it).second << endl;
                        it++;
                }

                return 0;
        }
```

This is a short but somewhat tricky example. We first set up a typedef for:

```
map<string, long, less<string> >
```

which is a map template with three template arguments. The first is the type of the key, in this example a string. The second is the value associated with the key, in this case a long integer used as a counter. Finally, because the keys of the map are maintained in sorted order, we provide a template comparison function (see issue #016 for another example of this).

Another typedef we establish but do not use in this simple example is the VAL type, which is a template of type "pair<string,long>". pair is used internally within STL, and in this case is used to represent a map element key/value pair. So VAL represents an element in the map.

We then read lines of input and insert each word into the map. The statement:

```
counter[buf]++;
```

does several things. First of all, buf is a char*, not a string, and must be converted via a constructor. What we've said is equivalent to:

```
counter[string(buf)]++;
```

operator[] is overloaded for maps, and in this case the key is used to look up the element, and return a long&, that is, a reference to the underlying value. This value is then incremented (it started at zero).

Finally, we iterate over the map entries, using an iterator. Note that:

```
(*it).first
```

cannot be replaced by:

```
it->first
```

because "*" is overloaded. When * is applied to "it", it returns a pair<string,key> object, that is, the underlying type of elements in the map. We then reference "first" and "second", fields in pair, to retrieve keys and values for output.

For input:

```
a
```

```
        b
        c
        a
        b
```

output is:

```
        a 2
        b 2
        c 1
```

There are some complex ideas here, but map is a very powerful feature worth mastering.

INTRODUCTION TO STL PART 5 - BIT SETS

We've been looking at various types of data structures found in the Standard Template Library. Another one of these is bit sets, offering space-efficient support for sets of bits. Let's look at an example:

```
        #include <iostream>
        #include <bitset>

        using namespace std;

        int main()
        {
                bitset<16> b1("1011011110001011");
                bitset<16> b2;

                b2 = ~b1;

                for (int i = b2.size() - 1; i >= 0; i--)
                        cout << b2[i];
                cout << endl;

                return 0;
        }
```

A declaration like:

```
        bitset<16> b1("1011011110001011");
```

declares a 16-long set of bits, and initializes the value of the set to the specified bits.

We then operate on the bit set, in this example performing a bitwise NOT operation, that is, toggling all the bits. The result of this operation is stored in b2.

Finally, we iterate over b2 and display all the bits. b2.size() returns the number of bits in the set, and the [] operator is overloaded to provide access to individual bits.

There are other operations possible on bit sets, for example the flip() function to toggle an individual bit.

INTRODUCTION TO STL PART 6 - STACKS

We're nearly done discussing the basic data structures underlying the Standard Template Library. One more worth mentioning is stacks. In STL a stack is based on a vector, deque, or list. An example of stack usage is:

```
#include <iostream>
#include <stack>
#include <list>

using namespace std;

int main()
{
        stack<int, list<int> > stk;

        for (int i = 1; i <= 10; i++)
                stk.push(i);

        while (!stk.empty()) {
                cout << stk.top() << endl;
                stk.pop();
        }

        return 0;
}
```

We declare the stack, specifying the underlying type (int), and the sort of list used to represent the stack (list<int>).

We then use push() to push items on the stack, top() to retrieve the value of the top item on the stack, and pop() to pop items off the stack. empty() is used to determine whether the stack is empty or not.

We will move on to other aspects of STL in future issues. One data structure not discussed is queues and priority_queues. A queue is something like a stack, except that it's first-in-first-out instead of last-in-first-out.

INTRODUCTION TO STL PART 7 - ITERATORS

In previous issues we've covered various STL container types such as lists and sets. With this issue we'll start discussing iterators. Iterators in STL are mechanisms for accessing data elements in containers and for cycling through lists of elements.

Let's start by looking at an example:

```
#include <algorithm>
#include <iostream>

using namespace std;

const int N = 100;

void main()
{
        int arr[N];
```

```
                arr[50] = 37;

                int* ip = find(arr, arr + N, 37);
                if (ip == arr + N)
                        cout << "item not found in array\n";
                else
                        cout << "found at position " << ip - arr << "\n";
        }
```

In this example, we have a 100-long array of ints, and we want to search for the location in the array where a particular value (37) is stored. To do this, we call find() and specify the starting point ("arr") and ending point ("arr + N") in the array, along with the value to search for (37).

An index is returned to the value in the array, or to one past the end of the array if the value is not found. In this example, "arr", "arr + N", and "ip" are iterators.

This approach works fine, but requires some knowledge of pointer arithmetic in C++. Another approach looks like this:

```
        #include <algorithm>
        #include <vector>
        #include <iostream>

        using namespace std;

        const int N = 100;

        void main()
        {
                vector<int> iv(N);

                iv[50] = 37;

                vector<int>::iterator iter = find(iv.begin(), iv.end(), 37);
                if (iter == iv.end())
                        cout << "not found\n";
                else
                        cout << "found at " << iter - iv.begin() << "\n";
        }
```

This code achieves the same end, but is at a higher level. Instead of an actual array of ints, we have a vector of ints, and vector is a higher-level construct than a primitive C/C++ array. For example, a vector has within in it knowledge of how long it is, so that we can say "iv.end()" to refer to the end of the array, without reference to N.

In future issues we will be looking at several additional examples of iterator usage.

INTRODUCTION TO STL PART 8 - ADVANCE() AND DISTANCE()

In the last issue we started discussing iterators. They are used in the Standard Template Library to provide access to the contents of data structures, and to cycle across multiple data elements.

We presented two examples of iterator usage, the first involving pointers, the second a higher-level construct. Both of these examples require some grasp of pointer arithmetic, a daunting subject. There's another way to write the example we presented before, using a couple of STL iterator functions:

```
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

using namespace std;

const int N = 100;

void main()
{
        vector<int> iv(N);

        iv[50] = 37;
        iv[52] = 47;

        vector<int>::iterator iter = find(iv.begin(), iv.end(), 37);
        if (iter == iv.end()) {
                cout << "not found\n";
        }
        else {
                int d = 0;
                distance(iv.begin(), iter, d);
                //cout << "found at " << iter - iv.begin() << "\n";
                cout << "found at " << d << "\n";
        }

        advance(iter, 2);
        cout << "value = " << *iter << "\n";
}
```

The function distance() computes the distance between two iterator values. In this example, we know that we're starting at "iv.begin()", the beginning of the integer vector. And we've found a match at "iter", and so we can use distance() to compute the distance between these, and display this result. Note that more recently distance() has been changed to work more like a regular function, with the beginning and ending arguments supplied and the difference returned as the result of the function:

```
d = distance(iv.begin(), iter);
```

A similar issue comes up with advancing an iterator. For example, it's possible to use "++" for this, but cumbersome when you wish to advance the iterator a large value. Instead of ++, advance() can be used to advance the iterator a specified number of positions. In the example above, we move the iterator forward 2 positions, and then display the value stored in the vector at that location.

These functions provide an alternative way of manipulating iterators, that does not depend so much on pointer arithmetic.

INTRODUCTION TO STL PART 9 - SORTING

We've spent the last couple of issues discussing STL iterators, which are used to access data structures. We will now start discussing some of the actual STL algorithms that can be applied to data structures. One of these is sorting.

Consider a simple example of a String class, and a vector of Strings:

```
        #include <vector>
        #include <algorithm>
        #include <iostream>
        #include <assert>
        #include <string>

        class String {
                char* str;
        public:
                String()
                {
                        str = 0;
                }
                String(char* s)
                {
                        str = strdup(s);
                        assert(str);
                }
                int operator<(const String& s) const
                {
                        return strcmp(str, s.str) < 0;
                }
                operator char*()
                {
                        return str;
                }
        };

        using namespace std;

        char* list[] = {"epsilon", "omega", "theta", "rho",
                "alpha", "beta", "phi", "gamma", "delta"};

        const int N = sizeof(list) / sizeof(char*);

        int main()
        {
                int i, j;

                vector<String> v;

                for (i = 0; i < N; i++)
                        v.push_back(String(list[i]));

                random_shuffle(v.begin(), v.end());
                for (j = 0; j < N; j++)
                        cout << v[j] << " ";
                cout << endl;

                sort(v.begin(), v.end());

                for (j = 0; j < N; j++)
                        cout << v[j] << " ";
                cout << endl;

                return 0;
        }
```

This String class provides a thin layer over char* pointers. It is provided for illustrative purposes rather than as a model of how to write a good String class.

We first build a vector of String objects by iterating over the char* list, calling a String constructor for each entry in turn. Then we shuffle the list, display it, and then sort it by calling sort() with a couple of iterator parameters v.begin() and v.end(). Output looks like:

```
phi delta beta theta omega alpha rho gamma epsilon
alpha beta delta epsilon gamma omega phi rho theta
```

There are a couple of things to note about this example. If we commented out the operator< function, the example would still compile, and the < comparison would be done by converting both Strings to char* using the conversion function we supplied. Comparing actual pointers, that is, comparing addresses, is probably not going to work, except by chance in a case where the list of char* is already in sorted order.

Also, sort() is not stable, which means that the order of duplicate items is not preserved. "stable_sort" can be used if this property is desired.

In the next few issues, we'll be looking at some of the other algorithms found in STL.

INTRODUCTION TO STL PART 10 - COPYING

In the last issue we started discussing the standard STL algorithms that are available, giving an example of sorting. Another of these is copying, which we can illustrate with a simple example:

```
#include <algorithm>
#include <iostream>

using namespace std;

int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0, 0};
int b[13];

int main()
{
        int i = 0;

        copy(a, a + 13, b);
        for (i = 0; i < 13; i++)
```

```
                cout << b[i] << " ";
        cout << endl;

        copy_backward(a, a + 10, a + 13);
        for (i = 0; i < 13; i++)
                cout << a[i] << " ";
        cout << endl;

        copy(b, b + 10, b + 3);
        for (i = 0; i < 13; i++)
                cout << b[i] << " ";
        cout << endl;

        return 0;
}
```

In the first case, we want to copy the contents of "a" to "b". We specify a couple of iterators "a" and "a + 10" to describe the region to be copied, and another iterator "b" that describes the beginning of the destination region. In the second example, we do a similar thing, except we copy backwards starting with the ending iterator. copy_backward() is important when source and destination overlap. In the third example, we copy a vector to itself, sort of a "rolling" copy. The results of running this program are:

```
1 2 3 4 5 6 7 8 9 10 0 0 0
1 2 3 1 2 3 4 5 6 7 8 9 10
1 2 3 1 2 3 1 2 3 1 2 3 1
```

As with previous examples, we could replace primitive arrays with vector<int> types, and use begin() and end() as higher-level iterator mechanisms.

Copying is a low-level, efficient operation. It does no checking while copying, so, for example, if the destination array is too small, then copying will run off the end of the array.

INTRODUCTION TO STL PART 11 - REPLACING

In the last issue we illustrated how one can do copying using STL algorithms. In this issue we'll talk about replacing a bit, that is, how to substitute elements in a data structure by use of iterators and algorithms that perform the actual replacement.

The first example uses replace() and replace_copy():

```
#include <algorithm>
#include <iostream>

using namespace std;

int vec1[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};
int vec2[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};
int vec3[10];

int main()
{
        int i = 0;
```

```
                replace(vec1, vec1 + 10, 10, 20);

                for (i = 0; i < 10; i++)
                        cout << vec1[i] << " ";
                cout << endl;

                replace_copy(vec2, vec2 + 10, vec3, 10, 20);

                for (i = 0; i < 10; i++)
                        cout << vec2[i] << " ";
                cout << endl;
                for (i = 0; i < 10; i++)
                        cout << vec3[i] << " ";
                cout << endl;

                return 0;
        }
```

In both cases we replace all values of "10" in the vectors with the value "20". replace_copy() is like replace(), except that the replacing is not done in place, but instead is sent to a specified location described by an iterator (in this case, "vec3").

The output of this program is:

```
        1 2 20 5 9 20 3 2 7 20
        1 2 10 5 9 10 3 2 7 10
        1 2 20 5 9 20 3 2 7 20
```

A more general form of replacement uses replace_if(), along with a specified predicate template instance:

```
        #include <algorithm>
        #include <iostream>

        using namespace std;

        int vec1[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};

        template <class T> class is_odd : public unary_function<T, bool>
        {
        public:
                bool operator()(const T& x)
                {
                        return (x % 2) != 0;
                }
        };

        int main()
        {
                int i = 0;

                replace_if(vec1, vec1 + 10, is_odd<int>(), 59);
                for (i = 0; i < 10; i++)
                        cout << vec1[i] << " ";
                cout << endl;

                return 0;
        }
```

In this example, is_odd<T> is a class template that is used to determine whether a value of type T is even or odd. The constructor call, is_odd<int>(), creates an object instance of the template where T is "int". replace_if() calls operator() of the template object to evaluate whether a given value should be replaced.

This program replaces odd values with the value 59. Output is:

```
59 2 10 59 59 10 59 2 59 10
```

There is also replace_copy_if(), which combines replace_copy() and replace_if() functions.

INTRODUCTION TO STL PART 12 - FILLING

In a similar vein to some of our previous STL examples, here is an illustration of how to fill a data structure with a specified value:

```
#include <algorithm>
#include <iostream>

using namespace std;

int vec1[10];
int vec2[10];

int main()
{
        fill(vec1, vec1 + 10, -1);
        for (int i = 0; i < 10; i++)
                cout << vec1[i] << " ";
        cout << endl;

        fill_n(vec2, 5, -1);
        for (int j = 0; j < 10; j++)
                cout << vec2[j] << " ";
        cout << endl;

        return 0;
}
```

fill() fills according to the specified iterator range, while fill_n() fills a specified number of locations based on a starting iterator and a count. The results of running this program are:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 0 0 0 0 0
```

INTRODUCTION TO STL PART 13 - ACCUMULATING

Another simple algorithm that STL makes available is accumulation, for example summing a set of numeric values. An example of this would be:

```
#include <iostream>
#include <numeric>

using namespace std;

int vec[] = {1, 2, 3, 4, 5};

int main()
{
        int sum = accumulate(vec, vec + 5, 0);

        cout << sum << endl;

        int prod = accumulate(vec, vec + 5, 1, times<int>());

        cout << prod << endl;

        return 0;
}
```

In this example, we specify iterators for a vector of integers, along with an initial value (0) for doing the summation.

By default, the "+" operator is applied to the values in turn. Other operators can be used, for example "*" in the second example. In this case the starting value is 1 rather than 0.

INTRODUCTION TO STL PART 14 - OPERATING ON SETS

We've been looking at various algorithms that can be used on STL containers. Another group of these are some algorithms for operating on ordered sets of data, illustrated by a simple example:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int set1[] = {1, 2, 3};
int set2[] = {2, 3, 4};
vector<int> set3(10);

int main()
{

        vector<int>::iterator first = set3.begin();
```

```
            vector<int>::iterator last =
                set_union(set1, set1 + 3, set2, set2 + 3, first);

            while (first != last) {
                    cout << *first << " ";
                    first++;
            }
            cout << endl;

            return 0;
    }
```

In the example we set up two ordered sets of numbers, and then take their union. We specify two pairs of iterators to delimit the input sets, along with an output iterator.

Algorithms are provided for taking union, intersection, difference, and for determining whether one set of elements is a subset of another set.