

## Introduction to STL for Macintosh Programmers

Good afternoon, I would like to start by introducing myself. I am Jon Kalb and I work for Liberty Software a Macintosh-only Software Consulting firm in the bay area.

Although I have written shipping code that makes heavy use of the Standard Template Library, I don't hold myself out as an expert. My mantra for STL is "learn it; love it; live it." Now I'm here to share it.

My goal for this session is not to explore STL in detail or to get into advanced or even intermediate topics. My goal is to provide an introduction to experienced C++ programmers that have little or no experience with STL.

If you **are** an experienced STL user, that's great because, later in this talk I will give you a chance to share some of your hard-won knowledge.

Although there is some minor updating, this is the essentially the same session that I gave last year. I would like to see hands of anyone that was in last year's session. As you can see from the schedule, there will be follow-on STL session later this afternoon and Darin Adler will be covering some of the standard library in his session called "PowerPlant and the Std. C++ Library." I am looking forward to that session.

STL, or the Standard Template Library, is now a part of the draft ANSI C++ Standard. Users of STL can begin to expect it to be universally available. For those of us that are not yet familiar with STL, the future is clear—professional C++ programmers will be expected to understand and maintain objects that depend on this new library.

STL code will probably look a little strange at first and learning how to use the library will require some effort. Fortunately, STL's power, ease of use, and consistency make it worthwhile.

My focus here is going to be on giving you the information you need to begin to use STL in your code. This discussion will be about practice, not theory, which is, in a way, a shame, because the thinking behind how the library is structured and used is very interesting and fun to explore. You will see that very little knowledge of templates is needed for getting started. Only when you are ready to extend the library is **writing** templates required.

Before we plunge into the code-level nitty-gritty, I do want to give us an overview of what it is that STL is trying to accomplish. I feel that this perspective is valuable, because I know how lost I was without it.

I was first drawn to STL as a standard container library. But as I read about STL, I was impatient that STL writers tend to start off by talking about generic algorithms, function objects, and/or iterators. If STL is about building a better list class, than why start off with these side topics?

It turns out that STL is not about containers. STL is really about generic algorithms. Containers are just a necessary step to reaching generic algorithm nirvana.

As an example of a generic algorithm, consider `qsort()`. `qsort()` is an ANSI C implementation of a generic algorithm. Regardless of the contents of your array, your data is sorted.

<qsort.html>

---

`qsort()` from <stdlib.h> An ANSI C Generic Algorithm

```
void qsort( void *base,
            size_t nmemb,
            size_t size,
            int(*compare)(const void *, const void *));
```

where:      `base`    is a pointer to an array  
         `nmemb` is the number of elements in the array  
         `size`    is the size of each array element  
         `compare`    is a pointer to a function defined as:

```
int compare(const void *e1, const void *e2);
```

where:      `e1` and `e2` are elements pointers  
returns      if  
         < 0      `(*e1) < (*e2)`  
         0      `(*e1) == (*e2)`  
         > 0      `(*e1) > (*e2)`

When subtraction is a valid operations for elements of type `T`, the compare funtion can be implemented as:

```
{
    return (int)((*(T *)e1) - (*(T *)e2));
}
```

---

But consider the limitation of this kind of generic algorithm.

First, every use of `qsort()` requires that we code a "compare" function, even if this function is trivial. Of course, if the compare is trivial, then dereferencing a function pointer is pretty inefficient. Whether or not we care depends on the relative time spent in compares as opposed to swaps. For most uses of `qsort()` this may not be significant, but it points up a weakness of this approach to generic algorithms.

Next, we have no control over construction, destruction, or assignment of the array items being sorted. All swapping is done by raw memory moves.

Also, by using void pointers, we have completely sacrificed type safety.

Finally, we are restricted to working with arrays and these must exist in the standard memory model. God, and the ANSI Standard, forbid that you use an unconventionally memory model or that your data be in a list, a tree, a file, or any container other than a simple array.

These limitations can all be overcome, and have been in STL, by the use of templates.

SLIDE OFF

Before looking at the STL equivalent of `qsort()` I want to present a quick and dirty definition.

One of the basic building blocks of STL is the iterator. For now, I want to define an iterator as anything that can take the deference operator or, to put it another way, if it can take pointer syntax, it's an iterator.

<iterator.html>

---

quick and dirty iterator definition

```
If this makes sense:
```

```
    object == *thing;
```

```
Then "thing" is an iterator.
```

```
    char *a = "";  
    '\0' == *a;
```

```
All pointers are iterators.
```

---

For now, when you hear iterator, think pointer.

Now let's look at STL's version of the quick sort algorithm:

<sort.html>

---

STL sort from <algorithm> An STL Generic Algorithm

```
template <class RandomAccessIterator>
    void sort( RandomAccessIterator first,
               RandomAccessIterator last)    {...}

template <class RandomAccessIterator, class Compare>
    void sort( RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp) {...}
```

where our quick and dirty translation is:

```
void sort(pointer first, pointer last);
    (Uses "<" to compare elements.)

void sort(pointer first, pointer last, Compare funct
*comp);
```

---

Instead of passing an array's base address and the number and size of elements in the array we pass a pair of iterators (think pointers). Due to template magic, these iterators are type safe. We need not pass in the element size because, like type safe pointers, we can safely increment and decrement as needed.

Passing two iterators is the standard way of passing a collection of items in STL. The first iterator refers to the first item in the collection, but the second iterator, despite its name, does **not** refer to the last. The convention is that the second iterator points one **past** the last item in the collection.

Using this convention, the two iterators exactly enclose all and only the elements to be considered.

<collection.html>

---

iterating over an STL collection

A set of data is represented in STL by a pair of iterators. The first iterator references the first element and the last iterator references one past the last element. This is represented in half-open interval notation as:

`[first, last)`

Iterating:

Instead of the typical C iteration:

```
for (i = 0; i < count; ++i)
{
    arrayBase[i]; /* element */
}
```

We use:

```
for (i = first; i != last; ++i)
{
    (*i); // dereferenced object
}
```

Never dereference the "last" iterator. It can only be used for comparing with the "first" iterator.

`(*last)` is undefined.

---

This may take some getting used to at first, but, on reflection, it makes a great deal of sense. Because the iterators may be pointers into a list, we can't rely on the standard "less than" comparison that we are used to. As we iterate over our collection, we just need to increment our "first" iterator and compare it to our "last" iterator. When the iterators are equal (which, by definition, means that they refer to the same item), then we are done.

Note that we **never** dereference the "last" iterator. It may be nil or any other value. It can only be used to compare with the "first" iterator. You should always assume that dereferencing it will bus error.

<sort.html>

```
template <class RandomAccessIterator>
    void sort( RandomAccessIterator first,
               RandomAccessIterator last)    {...}

template <class RandomAccessIterator, class Compare>
    void sort( RandomAccessIterator first,
               RandomAccessIterator last,
               Compare comp) {...}
```

where our quick and dirty translation is:

```
void sort(pointer first, pointer last);
    (Uses "<" to compare elements.)

void sort(pointer first, pointer last, Compare funct
*comp);
```

---

Returning to the STL sort declarations, we see that the first declaration does not take a comparison function. If no comparison function is passed, the STL sort implementation will simply use the "less than" operator for comparison (of the items, not the iterators). At first glance, this may seem to be of little use, but since template magic gives us type safety, the objects to be sorted can overload the "less than" operator. Without our having to code a comparison function, the objects are sorted into their "natural" order.

SLIDE OFF

In our first sample program I demonstrate the efficiency that we can achieve with templates. Often new coding technologies offer us a trade-off with power of expression or ease of use on one side and performance on the other side. This program compares quick sorting an array with qsort() and the STL sort() generic algorithm.

<STL1.cp>

---

```

// STL1.cp
#include <iostream>
#include <cstdint>

using namespace std;

int  ANSICqsortCompare(const void *el1, const void *el2);
int  ANSICqsortCompare(const void *el1, const void *el2)
{
    return (int)(*(long *)el1 - *(long *)el2);
}

int main()
{
    cout << "STL sort sample!\n\n";           // setting up arrays
    const size_t  aSize = 10240;
    long  *qArray = new long[aSize];
    long  *stlArray = new long[aSize];
    assert(qArray && stlArray);

    srand(time(NULL));                        // randomize
    for (int i = 0; i < aSize; ++i)
    {
        qArray[i] = stlArray[i] = rand();
    }

    long  qsortTickStart = clock();           // ANSI sort
    qsort(qArray, aSize, sizeof(long), ANSICqsortCompare);
    long  qTix = clock() - qsortTickStart;

    long  stlTickStart = clock();             // STL sort
    sort(stlArray, &stlArray[aSize]);
    long  stlTix = clock() - stlTickStart;

    for (int i = 0; i < aSize; ++i)          // verify
    {
        assert(qArray[i] == stlArray[i]);
    }
    cout << "qsort() " << qTix << "\nSTL sort " << stlTix << "\n";
    delete [] qArray; delete [] stlArray;
}

```

---

I tested this on both 68K and PowerPC machines with both the HP STL and the Modina STL implementations. The STL sort was from three to twelve times faster. Learn it; love it; live it.

Of course, I stacked the deck with this sample program. Because the objects being sorted were of a built-in type (they are longs) and because we used the form of `sort()` that does not use a "compare" function object we don't have any function call overhead when doing compares. But that is my point. Template magic allows us to implement a generic algorithm that, in some cases, doesn't require functional call overhead, when the traditional approach would always require it.

My point is not that generic algorithms are more efficient than specifically coded algorithms my point is that, using templates, we **can** be more efficient than using the traditional approaches to generic algorithms.

SLIDE OFF

Now that we have tasted generic algorithms, we have a better idea of why implementing them leads us to iterators and containers. The whole point of generic algorithms is that the work that needs to be done does not depend on the kind of data involved.

If a class of objects supports addition, then we can accumulate. If it supports addition and division by a scalar then we can average. If it supports the "less than" operator, then we can sort. It doesn't matter if the class represents ints, floats, strings, test scores, portfolios, medical records, or box scores. It also doesn't matter if the items are stored in an array, a list, a file, or on a web server.

If our generic algorithms only worked on items that are passed to them, like `min` and `max`, then we would have no need of iterators or containers.

Most generic algorithms are designed for an arbitrary number of items. This is why we have iterators. With this single piece of information we can get the value of an item (by dereferencing) or move to the next item in the collection (by incrementing or decrementing).

If the only containers that we ever used were arrays, then we could just use pointers and forget about this iterator nonsense. The one drawback to pointers is that they don't hide the container from the generic algorithm. Because `qsort()` uses pointers, it can only operate on arrays.

Imagine a linked list. A pointer to an element in the list has part of what we need in an iterator--if we dereference it we have our element. But if we increment this pointer we don't get the next item in the list, we are pointing at some arbitrary memory location.



Now imagine that we have defined a class of objects that point at items in lists. We can override the increment operator to have the object pointer walk the list. *Voila*,! We have a list iterator.

This also explains why iterators and containers are so closely linked. The implementation of the iterator is dependent on the nature of the container class.

We are almost ready to get down and dirty with some code, but before we start looking at containers we need to be aware of some restrictions placed on the objects that will be contained.

<class.html>

---

#### Theoretical class requirements

Classes whose instances will be stored in STL containers must have:

- a copy constructor
- operator=

If the instances will be sorted or compared then the class must have:

- operator==
- operator<

#### Practical class requirements

In practice, it may be necessary to please the compiler gods by defining some functions that are never called. The practical list of requirements is:

- a default constructor
- a copy constructor
- operator=
- operator==
- operator<

With the current release of CodeWarrior, every user defined type (include pointers to predefined types) requires a special macro to be used inside an STL container:

```
struct ContainedObject
{
    int a;
}
__MSL_FIX_ITERATORS__(ContainedObject);
```

---

OK, let's look at containers and their iterators. We already know that we can use arrays and pointers with generic algorithms, now let's look at the other containers that STL defines.

The first containers are called vectors and deques. Vector is not a terrible intuitive name (at least it wasn't for me). The idea is that it grows in only one direction. A vector is like an array except that it can grow, but only at the end. A deque is like an array except that it can grow at both the beginning and the end.

<STL2.cp>

---

```

// STL2.cp
#include <iostream>
#include <vector>
#include <deque>
using namespace std;
__MSL_FIX_ITERATORS__(char *);      // Not req. w/built ins in CWPro1
int main(void)
{
    vector<char *>    v;              // OK in CW Pro3.
//    vector<char *, allocator<char *> > v;    // Required in CWPro1

    assert(v.empty());
    v.push_back("sleep");
    v.insert(v.end(), "was");
    assert(v.size() == 2);
    v.push_back("for");
    v.push_back("the");
    v.push_back("weak");
    v.push_back("or");
    v.pop_back();
    v.push_back("and");
    v.push_back("sickly");
    v[1] = "is";                      // Used to replace existing: "is" to
"was"
//    v[7] = "DOS users";            // Cannot use [] notation to add
elements.

    vector<char *>::iterator vi;      // Not OK in CWPro1
//    vector<char *, allocator<char *> >::iterator vi;
    for (vi = v.begin(); vi != v.end(); ++vi)
    {
        cout << (*vi) << " ";
    }
    cout << endl;

    typedef deque<char *, allocator<char *> > MyDeque;
//    MyDeque    d(v.begin(), v.end()); // Was OK in CWPro1
    MyDeque    d(v.size(), "");       // This ugly work around
    copy(v.begin(), v.end(), d.begin()); // is required in CW Pro 3.

    d.erase(d.end() - 3, d.end() - 1);
    d.push_front("MacHack:");
    ostream_iterator<char *>out(cout, " "); // Not OK in CWPro1.
//    ostream_iterator<char *, char, char_traits<char> > out(cout, "
");
    copy(d.begin(), d.end(), out); cout << endl;
}
// sleep is for the weak and sickly
// MacHack: sleep is for the sickly

```

---

In this sample program we play with a vector and a deque.

As the saying goes, “In theory—theory is as good as practice... In practice, it isn’t.” The STL spec says that we should be able to declare our vector, `v` as “vector left angle char splat right angle.”

Metrowerks is not yet quite up to the STL spec, so as of CW12 or CWPro 1, we needed to add an allocator declaration to each STL container declaration. The second template parameter is supposed to be a default parameter. Since this wasn’t yet supported as a default parameter, we had to specify it explicitly. A similar problems was encountered when instantiating the `ostream_iterator<>` template.

Allocators do not fall into scope of this introduction so we won’t be saying anymore about them

As of CW Pro 3, Metrowerks still does not completely implement the entire spec, but the default template parameter problem has gone away. Unfortunately, some new problems have crept in. The first is the use of the `__MSL_FIX_ITERATORS__()` macro. As you can see from this example, it isn’t a terrible burden, but it is worse than it was last year. In CW Pro 1, this macro was only required for user defined classes and structs. Now it appears that it is also required for pointers to predefined types like `char *`s.

A much worse problem, in my opinion is that certain container member functions do not accept iterators from other containers. I’ll deal more with the extent of this problem in my next session. As you can see, we cannot call the deque constructor with vector iterators (although we could in CW Pro 1) and we cannot use the `deque::insert()` member as a work around.

This code demonstrates several functions of vectors and deques:

- |                          |  |
|--------------------------|--|
| <code>empty()</code>     | which works the way you would expect for all containers.   |
| <code>push_back()</code> | which is the workhorse for adding to a vector  |
| <code>insert()</code>    | which, although I am using it here to insert at the end of the vector, can be used to insert anywhere. Inserting in the middle of a vector or deque can be done, but it is not efficient. If you need to do this often, consider using a list. |
| <code>pop_back()</code>  | erases the last item in the vector or deque. <code>pop_back</code> does <b>not</b> return the last item. There is a very good  |

reason for this, which becomes apparent if you try to design an exception safe `pop()` member.

- array notation      both vectors and deques use array notation. This notation returns a reference so it can be used for both reading and writing, but it cannot be used to add elements to the container.
- `erase()`            which can be called to erase a collection by passing two iterators. I want to erase the elements that are the third from the end to the second to the end, but since the iterator should point passed the last item to be erased, I passed `d.end() - 3` and `d.end() - 1` for the range.

Note how we declare an iterator for our container. We simply restate the container declaration followed by “scope resolution operator iterator.” As container declarations tend to get a bit wordy, we often use a typedef to make life easier. I have done that for the deque declaration.

Iterating over the container is just as we discussed earlier and it is identical for both the vector and the deque.

The only other things to notice are the deque constructor, we populate by iterating across the vector, and the `push_front()` call. Deques can grow at both ends and the efficient way to insert at the beginning of the deque is `push_front()`. There is also a `pop_front`. These call is not supported for vectors.

SLIDE OFF

Vectors, deques, and arrays are called sequence containers because they store items in the sequence in which they are added. With one exception, other STL containers store items in sorted order and are called associative containers. The last sequence container is the list.

<STL3.cp>

---

```

// STL3.cp
#include <iostream>
#include <list>
using namespace std;
int main()
{
    typedef list<char> MyList;
    char start[] = "machack";
    MyList l(start, start + strlen(start));

    ostream_iterator<char> out(cout);
    cout << " start: ";
    copy(l.begin(), l.end(), out);

    cout << "\nremoved: ";
    l.remove('c');
    copy(l.begin(), l.end(), out);

    cout << "\n sorted: ";
    l.sort();
    copy(l.begin(), l.end(), out);

    cout << "\n unique: ";
    l.unique();
    copy(l.begin(), l.end(), out);

    cout << "\nreverse: ";
    l.reverse();
    copy(l.begin(), l.end(), out);
    cout << "\n";
}
// start: machack
// removed: mahak
// sorted: aahkm
// unique: ahkm
// reverse: mkha

```

---

In this example we see that using a list is similar to the other containers. The biggest differences are in the things that you already know about the differences between array-like classes and lists. Arrays have less overhead and lists are more flexible.

Here we are showing off `remove()`, `sort()`, `unique()`, and `reverse()`, which are member functions of the list class. This class also has functions to splice() lists together or merge() sorted lists.

But in going from array-like containers to lists we did have to give up some things. We can no longer use array notation or the `at()` function to address items by their position in the list.

The rest of the STL containers are associative. These containers keep the items they contain in sorted order. As you would expect, at construction, we either pass a function to use as a compare function or we use the default “less than” comparison.

Because these containers are always sorted, we no longer use `push_back()` or `push_front()`. We can call `insert()` without passing a positions iterator.

<STL4.cp>

---

```
// STL4.cp
#include <iostream>
#include <set>
using namespace std;
int main()
{
    typedef      set<char, less<char> MySet;
    typedef      multiset<char> MyMultiSet;
    ostream_iterator<char>  out(cout);
    char  start[] = "sleep is a poor substitute for caffeine";
    MySet s(start, start + strlen(start));
    MyMultiSet ms;
    ms.insert(start, start + strlen(start));

    cout << "      start: ";
    copy(s.begin(), s.end(), out);
    cout << "\n      start: ";
    copy(ms.begin(), ms.end(), out);

    MySet::iterator  si1, si2;
    MyMultiSet::iterator  msil, msi2;
    si1 = s.lower_bound('b');
    si2 = s.upper_bound('s');
    s.erase(' ');
    s.erase(si1, si2);
    msil = ms.lower_bound('b');
    msi2 = ms.upper_bound('s');
    ms.erase(' ');
    ms.erase(msil, msi2);

    cout << "\nremoved b-s: ";
    copy(s.begin(), s.end(), out);
    cout << "\nremoved b-s: ";
    copy(ms.begin(), ms.end(), out);
    cout << "\n";
}
//      start:  abcefilnoprstu
//      start:  aabceeeeefffiilnoooprrrrssstttuu
// removed b-s: atu
// removed b-s: aatttuu
```

---

In this example we use two of the associative containers, the set and the multiset. The set is a simple ordered collection that differs from the multiset only in that it does not support duplicate items.

Both set and multiset have `lower_bound()` and `upper_bound()` members. These members return an iterator that refers to the lowest or highest location that an item of the passed value could be placed without violating the ordering rules.

The last two containers that we are going to examine are called the map and the multimap. As you have already guessed, the difference between them is that the multimap can contain duplicate entries.

All of the containers that we have looked at so far have been designed to contain items, but the map is designed to contain pairs of times. Each pair consists of a key and a value. When a pair is added to a map the key is used to find where, in the sort order, this pair belongs.

A map is like a set in which each item in the set has a link to another item that may be of a different type.

<sets.html>



---

A Visualization for sets and maps:

set == ordered items w/o dups

1  
4  
7  
9

multiset == ordered items w/dups

1  
4  
4  
7  
9

map == order keys with values w/o dups

key	value
1	-- c
4	-- x
7	-- j
9	-- w

multimap == order keys with values w/dups

key	value
1	-- c
4	-- x
4	-- d
7	-- j
9	-- w

---

<STL5.cp>

---

```
// STL5.cp
#include <iostream>
#include <map>
using namespace std;
int main()
{
    int integers[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    char *intNames[] = {"zero", "one", "two", "three",
                        "four", "five", "six", "seven",
                        "eight", "nine", "ten"};

    typedef map<int, char *, less<int>, allocator<char *> >
MyMap;
    typedef vector<int, allocator<int> > MyVector;
    MyMap m;
    for (int i = 0; i < 11; ++i)
    {
        m.insert(MyMap::value_type(integers[i], intNames[i]));
    }
    MyVector lhs(integers, integers + 11), rhs(integers, integers +
11);
    random_shuffle(lhs.begin(), lhs.end());
    random_shuffle(rhs.begin(), rhs.end());
    for (int j = 0; j < 11; ++j)
    {
        int sum = lhs[j] + rhs[j];
        int diff = lhs[j] - rhs[j];
        if (sum < 11)
        {
            cout << m[lhs[j]] << " + " << m[rhs[j]] <<
                " = " << m[sum] << endl;
        }
        if (diff >= 0)
        {
            cout << m[lhs[j]] << " - " << m[rhs[j]] <<
                " = " << m[diff] << endl;
        }
    }
}

// two + zero = two
// two - zero = two
// eight - five = three
// nine + one = ten
// nine - one = eight
// three + two = five
// three - two = one
// one + four = five
// ten - ten = zero
// zero + seven = seven
// seven + three = ten
// seven - three = four
```

---

In this example, we create a map that maps integers to their names. We create two vectors of integers and use the `random_shuffle` generic algorithm to shuffle their values. Next we walk the vectors and add and subtract their corresponding values. What makes it interesting is that when we print our results we use the map to replace all the integers with their names.

We can use array notation so that the expression “`map sub key`” returns the map value. Learn it; love it; live it.

My final code example deals with exceptions. My best words to you are proceed with caution.

<STL6.cp>

---

```
// STL6.cp
#include <iostream>
#include <deque>
using namespace std;
/*
```

The exception hierarchy:

```

exception          root of all exceptions
|
|      bad_alloc   memory allocation exception
|
logic_error        precondition violation exception root
|      |      |
|      |      out_of_range    attempt to reference a container
|      |                        using an illegal index
|      |
|      length_error    attempt to create a string with an
|                        illegal length
|
invalid_argument   attempt to create a container with an
                    illegal size such as -1

```

```
*/

int main()
{
    const int    limit = 10;
    vector<int> v;
    for (int j = 0; j < limit; ++j)
    {
        v.push_back(j);
    }
    try
    {
        int i = v[limit];
        cout << i << " no exception for i = v[limit];" << endl;
        i = v.at(limit);
        cout << i << " no exception i = v.at(limit);" << endl;
        i = v.at(limit + 1);
        cout << i << " no exception i = v.at(limit + 1);" << endl;
    }
    catch (out_of_range& error)
    {
        cout << "An out of range execption was caught" << endl;
    }
}

// CW Pro 1 output
// 1868783980 no exception for i = v[limit];
// 1868783980 no exception i = v.at(limit);
// An out of range execption was caught

// CW Pro 3 output
// -8 no exception for i = v[limit];
// An out of range execption was caught

```

---

As this example shows not every call will throw exceptions when you might expect. Here the vector's `at()` call will throw, but the array notation fails silently. You have been warned!

Note that a CW Pro 1 bug that we saw last year has been fixed. The `at()` call used to not throw until it was off by two and now it throws correctly.

I want to leave you with a few tips:

`<hints.html>`

---

Hints for STL coders.

Don't forget that the "last" iterator points "one past".

For your custom objects, always define:

- a default constructor
- a copy constructor
- operator=
- operator<
- operator==

With the current release of CodeWarrior, every user-defined struct or class and some predefined types (specifically pointers) requires a special macro to be used inside an STL container:

```
struct ContainedObject
{
    int  a;
};
__MSL_FIX_ITERATORS__ (ContainedObject);
__MSL_FIX_ITERATORS__ (char *);
```

Objects are copied into contains. The original object still exist and may need to be deleted.

If it ain't broke break it. Better to learn how the compiler complains when you do know what to fix.

Don't do relational compares on iterators. Only use == or !=.

Don't forget to match the const/non-constness or iterators and containers.

Watch remove(), it doesn't erase().

Watch nested templates--don't let them look like ">>".

```
vector<int, allocator<int>>> v.
```

You don't even what to know what your compiler thinks of this.

If you use the HP STL implementation beware of init\_page\_size in defalloc.h. It allocates 4K for every STL container that is created. This makes sense on UNIX, but it is probably not what you want.

---

We have looked at all of the STL container classes except the string class and briefly discussed generic algorithms. My hope is that you have enough information to begin to use STL in your code, but the list of items that we have not discussed is longer than the items that we have.

STL is very complex and I have deliberately tried to steer us away from complicating issue. About the issues that we have discussed, I am now ready to take questions.

