

More STL

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.

The first thing I should say is that I hope that fact that this session has been falsely advertised hasn't inconvenienced anyone or pissed off the FTC.

In your conference program this session is called Advanced STL, but a better name is just "More STL." 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 intention is simple to build on what we discussed in my earlier session. I would like to see the hands of anyone that was in that earlier session...thanks.

I intend this session for three types of Macintosh programmers: those that make off-by-one errors and those that don't.

As you can see from the schedule, 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.

In my earlier session I gave an overview of what STL is trying to accomplish. I feel that this perspective is valuable, because I know how lost I was without it. I won't repeat that overview now, but the text of that session is on my web site, last year's MacHack CD, and will be on this year's CD. If you don't like to read, then feel free to find me after the session and talk to me about it. I think you will find that it is easier to get me to talk about it than it is to get me to shut-up about it.

The first topic I want to look at in this session is in what ways CodeWarrior in particular and compilers generally deviate from the standard.

Let me first clarify where the limitations lie. Although we tend to say something like "Metrowerks' library doesn't support this or that," the truth is that the library (which Metrowerks is licensing from Modina) is ahead of the compiler. If you look at Modina's library (since it is nothing but header files, you can see it all), you will see that it is strewn with `#ifdefs`. These `#ifdefs` provide work-arounds in places where the compiler doesn't yet have all of the features that are required to completely and correctly implement the library.

I will be talking about three areas: default template arguments, partial template specializations, and member templates.

<defargs.html>

Default Template Arguments

If your compiler doesn't (completely) support default template arguments, then instead of writing:

```
vector<char *> v;
```

you will have to write:

```
vector<char *, allocator<char *> > v;
```

CodeWarrior Pro 3 no longer requires this work-around.

One of the most troublesome limitations to deal with is lack of support for default template arguments, because anyone trying to declare any standard template-based container hits this problem immediately and has to use this ugly and wordy work-around. I mention this only for the benefit of those that are working with old code or an old version of the CodeWarrior compiler. As of CodeWarrior Pro 3, this notation is not required.

<remove slide>

Partial template specializations is more complicated. Let's start by explaining template specialization. Consider the standard vector class. It is written to be as efficient as possible without making any assumptions about what is contained. But the code could be better, in some cases, if we could make some assumptions. For example, the bool case can be considerably more space efficient by packing and unpacking bits. The case where we are storing pointers (not smart pointer objects, but really C++ pointers) we can be more efficient because we can safely move pointers as raw memory without concerns about calling constructors or destructors.

Let's look at how these specializations would be declared.

<parttemp.html>

Full Template Specialization

A template definition:

```
template <class T>
class vector
{
    ...
}
```

specialized:

```
template <>
class vector<bool>
{
    ...
}
```

The real template definition:

```
template <class T, class Allocator = allocator<T> >
class vector
{
    ...
}
```

specialized for T and the default allocator:

```
template <>
class vector<bool, allocator<bool> >    // note space, not ">>"
{
    ...
}
```

Partial Template Specialization

The real template definition:

```
template <class T, class Allocator = allocate<T> > // note space
class vector
{
    ...
}
```

specialized for T, but not for Allocator:

```
template <class Allocator>
class vector<bool, Allocator>
{
    ...
}

template <class T, class Allocator>
class vector<T*, Allocator>
{
    ...
}
```

The syntax for declaring a template specialization is straightforward. With full template specialization we specify every template parameter.

Unfortunately, in both of the examples that we just mentioned, we don't want to specify every template parameter. Neither of our specializations depends on the allocator, so we don't want to specify it. We want our specializations to be generic with respect to allocators. Declaring a specialization that specifies only some of the template parameters is called partial template specification and, it turns out, this is pretty advanced compiler technology.

Very few compilers can do this yet. According to the people that I have been talking to at Metrowerks they now have a compiler that does this and it is about to be beta tested. We might have this in CodeWarrior Pro 4.

So where do we stand until then? Well obviously we won't get some of the optimizations that we might otherwise have. (Note that Metrowerks/Modina does actually implement a bit vector specialization through some interesting template slight-of-hand.)

The most visible stumbling block is `__MSL_FIX_ITERATORS__()`. This macro is required for any user-defined object that will be used in a container. Here user-defined is meant to include pointers to predefined types.

<class.html>

This problem has a pretty lightweight work-around, but the problem of member templates is more troublesome.

Suppose a template class has a member function that should also be a template. Here is a real example from the standard.

<memtemp.html>

The standard declares that we can construct a container with a couple of iterators that define a valid range of the proper type of object. In this example we have a couple of int vector iterators being used to construct a deque.

Since anything that can produce an int when dereferenced is a valid iterator, it isn't possible to write this constructor except as a template, because we don't know in advance the type of the iterators. Unfortunately, CodeWarrior and most other commercially available compilers (on any platform) don't support member templates. The last word I heard from Metrowerks is that this is a possibility in CodeWarrior Pro 4.

Ironically enough, this usage actually worked in CodeWarrior Pro 1. The reason was that the compiler type checking was loose enough that it couldn't distinguish between vector and deque iterators if they pointed to the same type.

So what is the work-around? There is no simple use-this-macro-and-it-fixes-everything work-around. You just have to re-write your code. Here is one example of how to work around the constructor problem—and how not to work around it. The first example will compile, but will fail at run time. The copy algorithm doesn't increase the size of the target container. This can be done with the `insert_iterator` adapter as shown.

This adapter simply adapts an iterator's behavior to insert instead of overwrite.

This is a solution for one instance of this problem, but the scope of the problem is much larger than this one instance. This problem affects at least one form of all the following template class members: `assign()`, `append()`, `insert()`, `insert_equal()`, `merge()`, `remove_if()`, `replace()`, `sort()`, `unique()`, and constructors for all of the standard containers include `auto_ptr`, `string`, and `pair`. Good luck!

Actually it is not as hopeless as it sounds. Some of these member functions are pretty obscure, there are ways around all of these problems—you are highly paid professionals after all, right—and Metrowerks will probably support the entire standard library in the next release or two.

My next topic is the template object `auto_ptr<>` and the question that I want to ask is, "If you aren't using `auto_ptr<>` then how are you writing exception safe code?"

I can remember the first time I was introduced to stack based objects. This was before PowerPlant made them deservedly popular with their implementation of their "St" classes.

My initial reaction was “Wow!” This is very useful for a whole set of situations. We can restore state after locking handles, opening files, setting grafport attributes, and on and on.

My joy was just the convenience and safety of not having to remember to restore state before returning. At that time I wasn’t thinking about exceptions. Now that I am using exceptions in my code, my mantra for `auto_ptr<>` is, “Learn it; love it; live it.”

The `auto_ptr<>` template class is just a wrapper that calls `delete` on the pointer that it owns in its destructor. (It also has member operator overloads that allow it to transparently support pointer syntax. It **really** is just `set` and `forget`.) Let’s look at an example of why `auto_ptr` makes life good.

`<noautoptr.html>`

Here we have a class that contains pointers to objects that it owns. Imagine that our attempt to create the object pointed to by “b” throws an exception. What happens to “a?” It leaks! The destructor is not called for objects that throw while being constructed.

This type of situation comes up often. Not just in constructors, but any where that we are working with pointers that must be deleted and an exception might be thrown.

I have also provided an alternative implementation. This solves the problem, but look how much more work it is. Now I’m not opposed to work, but more work means more chances to forget something. I deliberately “forgot” something in this example. Have you spotted it?

That’s right. I need to re-throw the exception in the catch block.

Consider for a moment if “a” and “b” were `const` pointers. Not pointers to `const` objects, but `const` pointers. That would mean that they can only be initialized in a constructor’s initializers. Now our alternative fix isn’t just wordy and error prone; it’s also useless. This problem isn’t unsolvable without `auto_ptr<>`, of course, but the code will get messier. If we use `auto_ptr<>` the code gets cleaner.

`<autoptr.html>`

Here is our solution using `auto_ptr<>`. Hey that is exactly what we wrote the first time, except that we have declared “a” and “b” to be `const auto_ptr<>`s and our destructor got simpler.

The syntax for using `auto_ptr<>`s is designed to be transparent. Use them just as if they were pointers.

The code works just like we want it to. It’s exception safe because these pointers will always take care of themselves when they go out of scope. In the LFile example, `sourceFile` will not leak and the file will be closed whether we hit an exception or not.

With almost no overhead at all we converted potentially dangling pointers into objects that clean up after themselves. Learn it; love it; live it.

<remove slide>

Now we come to what is fast becoming my favorite container class. The standard string class. Every framework has a string class so why do we need yet another string class?

Perhaps we don’t, but this one is well designed, potentially performant, and, best of all for contractors like myself, it is, or soon will be universal.

I am not going to show you the entire set of public functions; I’m not even going to show you all of the different constructors. I just want to show you the high points.

<STL9.cp>

```
// STL9.cp
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string      a = "Standard strings are easy to initialize.";
    a += " And they can grow without end as needed.";
    string::iterator  iter = find(a.begin(), a.end(), 'A');

    if (iter != a.end())
    {
        size_t      offset = iter - a.begin();
        a[offset] = 'a'; // They use array notation like C
strings.
    }
    iter = find(a.begin(), a.end(), '.');
```

```
if (iter != a.end())
{
    a.erase(iter);    // They are STL containers.
}
cout << a << endl;
cout << "Size is ";
// cout << strlen(a);    // No coercion to char *.
cout << strlen(a.c_str());    // But it easy to get a char *.
cout << "." << endl;
// They automatically delete their allocated memory when they go
out of scope.
}
// Standard strings are easy to initialize and they can grow without
end as needed.
// Size is 80.
```

The string type's template nature is hidden by the "string" typedef. The result is that strings really come very close to looking and behaving as if they were fundamental language types. They support string/array syntax and STL container syntax. There is no char * operator, but the c_str() member function returns a const char * so we can continue to use all our old routines that depend on C strings. (In his session Darin will talk about how we can use get away from some of these old routines like sprintf().)

I should mention that the c_str() function only promises that the returned C string is valid until the next operation on the string container.

As I said, strings are becoming my favorite container and it's just because they are so darn convenient and easy to use. You don't have to worry about over running buffers or leaking memory.

Now I know what you are thinking. "There is no such thing as a free string. What is the catch?" You are right to be skeptical. Alex Stepanov, the creative force behind STL points out that library users really do need to know more than just a library's API. Without some kind of understanding about performance issues, a library isn't going to be used. Consider the MacOS Resource Manager. The API makes it look like a database. But it wasn't intended for use as a database and it won't perform well if that is how you use it. The API alone doesn't tell you all you need to know.

The standard, of course, doesn't tell us anything about the implementation of the standard strings or any of the standard classes because it doesn't want to mandate any one approach. The drawback for us is that this means that there is no documentation for the implementation. There are now quite a few books that deal with STL, but almost all of them avoid documenting the implementation for the same reason. There is no one implementation and anything that is written about implementation may be wrong with respect to different platforms, different compilers, and even different release of the same compiler.

With this as a caveat, I have done a little digging on the Metrowerks/Modina string class implementation.

The first important thing to notice is that the data is always contained in a single contiguous buffer that is null terminated. The implication of this is that the `c_str()` function is trivial; just return the address of this buffer. **Lesson one:** string conversion to C strings is cheap.

The next thing we find is that the buffer is grown in 32 byte increments. The total space overhead other than this buffer is twenty-four bytes so your space overhead per string is in the range of twenty-five to fifty-five bytes. For someone that is used to using `Str255`'s for strings, this is actually quite an improvement in space performance.

Since growing the string buffer involves a new allocation, a copy, and a delete of the old buffer, it is better to anticipate your string size requirement and call the class's `reserve()` member function if that is possible.

Of course, if you are really concerned about such matters, you can hack the library source and change the buffer increment.

Lesson two: strings are relatively space efficient and it is better to grow your strings in one fell swoop than incrementally.

The third thing we learn about the implementation is that the library only copies the string buffer if it has to. In other words, if you initialize a string and then create another string from it, either by copy construction or assignment, then both classes share the same buffer and the buffer is reference counted.

This can be both a space and time win if you are likely to have more than one string with the same value. Is that likely in the real world? Perhaps. Imagine that you are maintaining a list of files with strings containing the files' names for each of several different platforms. You might have many cases where the strings would be exactly the same. **Lesson three:** if you are working with strings that might be identical, create them from each other to maximize buffer sharing.

The last thing that we learn from looking at the source is that once you do anything that might allow you alter a string, the string immediately makes its own copy of the string buffer and puts itself into a state (I call it the don't-share state) that will never again allow its buffer to be shared. Why? Well if, for example, you call `begin()` on a string, you have an iterator to the first char of the string and you might change it (unless it is a const string, in which case this doesn't apply).

If you change the first char of this string then it wouldn't be equal to any other string that is sharing the buffer so it has to make its own copy. Also, once the new copy is made and you have an iterator to the first char, then you could change that first char at any time in the future, so it won't share its buffer with any other string to which it is assigned or from which it is constructed.

Lesson four: if you are working with strings whose values are acquired from other strings, don't do anything that might modify a string, unless that is your intention. This is much more subtle than it sounds.

<STL10.cp>

In this example strings "a" and "b" start out sharing a buffer and end up with each one having its own buffer. The question is: On which line does the buffer copy happen?

The answer is on line one. Why? The subscript operator to a non-const string returns a reference to a char in the string. Although this code treats this value as const, the string's subscript operator function doesn't have any way of knowing how you will use the reference, so it must assume that you will modify the char value and it makes its own copy of the buffer.

How can we avoid copying the buffer when all we want is read access? Well we could try casting the string to a const string, but that won't work. Casting the string to a const string actually constructs a temporary const string from from our string. This may not be as bad as it sounds, because the buffer itself will be reference counted and not copied (unless the original string is in the don't-share state), but it's not a very good solution, because we don't really have a good way of knowing if the original string is in the don't-share state. A better solution is found on line three. Since the `c_str()` member function returns a `const char *`, you can't modify the string, so the buffer is never copied. Remember `c_str()` is cheap.

If I were planning to implement an application that was a heavy string processor, I would want to look very hard at any candidate for a string base class, but for general purpose application string handling (whatever that is) the standard string class seems to have adequate space and time performance and great safety and portability potential. Learn it; love it; live it.

OK, you have one more objection, right. No string class is worth anything to a Macintosh programmer if it doesn't support conversion to and from Pascal strings.

Clearly, the standard string class doesn't support Pascal string neither is it likely to in the future. But adding such support ourselves is not a terrible burden.

<STL8.cp>

```
// STL8.cp
#include <iostream>
#include <memory>
#include <string>
#include <cstring>
#include <MacTypes.h>

template<class PascalString>
class PStrTemp
{
    PascalString s;
    void MakePString(const char *cString)
    {
        std::size_t length= std::strlen(cString);
        s[0] = length > (sizeof(s) - 1)? (sizeof(s) - 1):
length;
        std::strncpy((char *)&s[1], cString, (sizeof(s) -
1));
    };
public:
    PStrTemp(const char *cString)
```

```

        {
            MakePString(cString);
        };
PStrTemp(const std::string& stdString)
{
    MakePString(stdString.c_str());
};
operator const unsigned char *() const {return s;};
};

typedef PStrTemp<Str255> ConstStr255;
typedef PStrTemp<Str63> ConstStr63;
typedef PStrTemp<Str32> ConstStr32;
typedef PStrTemp<Str31> ConstStr31;
typedef PStrTemp<Str27> ConstStr27;
typedef PStrTemp<Str15> ConstStr15;

using namespace std;
void PStringFunction(ConstStr255Param stringValue);
void PStringFunction(ConstStr255Param stringValue)
{
    string      outString((const char *)&stringValue[1],
*stringValue);
    cout << outString << endl;
}

int main()
{
    char  *a = "C string";
    string      b = "std::string";
    PStringFunction(ConstStr255(a));
    PStringFunction(ConstStr15(b));
}
// C string
// std::string

```

In this example I have created a light-weight template class that can be created on the stack to convert from a Pascal string to a standard string. I used a template so that you can decide which size string you want to put on the stack.

This example also shows how easy it is to create standard strings from Pascal strings. I will concede that this is a bit wordy and could easily lead to a typo. I will leave a simple C style macro or a fancy template as an exercise you to enjoy.

If you have any question, now is the time.

Learn it; love it; live it.

