The Standard Template Library and Macintosh Programming

George G. Geller, Ph.D.
Macintosh programmer and consultant.
email: 71321,2544@compuserve.com

The Standard Template Library (STL) was developed by Alexander Stepanov and Meng Lee. This large and innovative body of code has been adopted as an important part of the new ANSI draft standard C++ and promises to dramatically influence the way C++ programmers work.

STL is simultaneously efficient, general, and compact in representation. Its central paradigm is the decomposition of programming tasks into generalized algorithms that work on 'containers'. The containers hold native C++ data types or user-defined objects. This approach, known as generic programming, drastically reduces the code base necessary to handle many common programming tasks.

In this presentation, I will introduce STL and show examples of its application in Macintosh programming. The current state of STL support in Macintosh C++ compilers will also be reviewed.

In this paper I will demonstrate how to use the Standard Template Library (STL) to help you with your Macintosh C++ programming.

Although Alexander Stepanov, the principal author of the library, has a mathematical background which is readily apparent in the published specifications I won't spend a lot of time showing you a bunch of abstractions. I'll just go over the basics that you absolutely need to understand so that you can follow the concrete examples that I will present later.

In order to use STL you must have a good grasp of C++ templates. You have to understand some details about how your development environment implements templates. And, more likely than not, you have to get used to looking at template code in your debugger.

In exchange for this small investment, STL gives you an immense payback. To start with, you no longer have to implement most types of data structures; vectors (which are equivalent to dynamic arrays), linked lists, b-trees, and associative arrays are all provided. You no longer have to implement many common algorithms such as sorting and searching. Also, STL handles many issues of memory management automatically for you -- you almost never need to use the C++ new and delete.

STL will be part of the next ANSI standard. That means it will be included with any C++ compiler on any platform. Since it is a standard, you can be confident that code you write using STL will work in the future and with any hardware platform or compiler.

To get started with STL, you should probably read one of the introductory books I list in the bibliography. I recommend Mark Nelson's "C++ Programmer's Guide to the Standard Template

Library."

Comments about efficiency: In general the performance matches what you get for hand-coded C. The advanced container classes (set, multiset, map, and mulitmap) in STL use red-black trees. Alexander Stepanov, the principal author of STL, believes that this is the best structure for something that will reside in memory. Anyone is free to try their hand at substituting another structure such as a B*tree, skip lists, splay trees, or half-balanced trees. Comparing the performance of the various structures could become a major research project.

Much of my paid work involves porting applications from other platforms (whose names needn't be mentioned here, since we don't have barf bags handy) to the Macintosh. A really cute trick that the implementors of STL pulled off was to isolate the machine dependencies of the memory model to their allocator class. This leaves the code you actually work with looking very clean.

What is STL? It is fundamentally a collection of container templates, which are classes, and algorithm or function templates. Supporting these is the concept of an iterator. An iterator is a generalized pointer. Just as you de-reference a pointer in C to gain access to a member of an array, you use an iterator to gain access to an object stored in one of STL's container classes. There are several types of iterators. They all support the basic operations of incrementing and dereferencing. If you use the right kind of iterator object (an insertion iterator), STL's container classes automatically grow to accommodate however many elements you want to add, up to the limit of available RAM.

It is vital to understand that STL algorithms are not encapsulated in classes. They are function templates that operate on the container classes. This means that most algorithms work with most container classes. However, it does make "naked" STL harder to use than a typical class library. One solution to this is to write specialized wrapper classes that use STL. In the example program for this talk I used this approach for my macString class which transparently handles the chore of dealing with Pascal and C style strings in Macintosh programming. MacString objects uses the STL vector container class to store the string data. There are commercial libraries available, from RogueWave and probably others, that use the class wrapper approach to hide the complexity of naked STL.

Perhaps the strongest motivation for adopting STL and the generic programming approach is that it vastly reduces the amount of coding you have to do. You get this reduction because you (or a library author) write a container class only once, then use C++ templates and generic programming to apply various algorithms to your container class. The algorithm, say a sort, is written only once and is applicable to different types of containers such as arrays or queues. This is the central paradigm that I mentioned in the abstract. Consider this example: you have three different classes of objects, lets call them strings, window pointers, and sprites. And you have three different types of containers that you want to use to store these objects in, vectors, queues, and stacks. And you need insertion, extraction, and sorting algorithms for every combination of object and container. That leaves you with a total of 27 (3x3x3) algorithms to implement. If you use generic programming techniques you reduce the number of algorithms to just 3. This is a very simple example. In real world-applications, the benefits are potentially much greater.

STL is open. The source code is provided and is an a form that is easy to use and modify. You can write a new algorithm and apply it to any of STL's container classes. Conversely, you can write a new container class and use STL's library of algorithms on it.

What is missing from STL? Most notably, persistence. There is nothing built-in that lets you store and retrieve your objects and data structures to a file. There are commercial extensions to STL, from vendors such as RogueWave and ObjectSpace, that include persistent object databases. There are also hints (in the interview of Alexander Stepanov for the March 1995 Dr. Dobbs Journal) that future versions of STL will include persistent objects. Also there is nothing analogous to a class browser.

What will you find in STL? You'll find a great implementation of containers and algorithms. Alexander Stepanov has spent two decades working on generic programming and the ideas behind STL. The classes and functions are in there. There is a heck of a lot of functionality. It is up to you, the programmer, to know where it is and how to use it. You'll need to keep a reference book handy.

STL example program for the Macintosh

I wrote a program called Dupes to demonstrate using STL programming techniques on the Macintosh. Dupes catalogs your hard disk and creates a listing of files that have duplicate names.

From the pedagogic standpoint there are two primary points to be made about Dupes.

First, I created a macString class that uses the STL vector container to store string data in a form that is easy to access and manipulate. This class has member functions that allow transparent access to your string object as a C string or a Pascal string. The following code snippet illustrates:

```
  #include "macString.h"
  macString ms;
  strcpy(ms, "STL saves work."); /* copy a C string into ms */
  ParamText(ms, 0L, 0L, 0L);     /* access ms as a Pascal string */
  Alert(129, nil);
```

If you are so inclined, you can reuse the code for the macString class in your own application.

Second, Dupes uses the STL multimap container class to store and efficiently access each file name and the full path name in RAM. The file name (e.g. notes.txt) is used as the key, and the full path name (e.g.hd:MacHack:notes.txt) is stored as a value. You will see how little coding I had to do to accomplish this. STL does all the hard work.

Let's examine the code for macString.cp (listing 3, below). The first thing to notice is that we include stl.h. This, not an object library, is where the Standard Template Library is added to the project. Note again that all the code for the library is actually in stl.h and some associated text files. STL is completely open; nothing is held back or hidden.

The next interesting thing in macString.cp is a collection of no less than twelve template statements. These commands instantiate the templates in stl.h to create the type of objects and functions that you need for your application. Some of the template commands are remarkably complicated. Fortunately, you don't have to figure them out yourself. Some development environments implicitly generate the necessary template commands. In Symantec C++ you can figure out which ones you need based on the linker errors that are generated. You don't even need to type in the code, just copy it from the error output window.

Most of the magic of the macString class is encapsulated in the synchronize function. Synchronize uses two Boolean data members, bPStrOK and bCStrOK to ensure that the C string and Pascal string contain the same data. The other member function of macString are responsible for calling synchronize at the appropriate times.

Listing 1 contains the code for Dupes main itself. In terms of understanding STL, the primary thing to notice is that we use the multimap container class to store the full path name for each file on the disk indexed by the file name. This is done in the global simply defined by: mmap m;

In main we call CatalogADirectory to fill up the multimap, then we use a while loop to print out the files sorted by name. Note the conditional for the while loop:

```
    while (i != m.end())
```

Why can't we use:

```
    while (i < m.end()) /* !!wrong!! */
```

The answer is that as an iterator into a multimap, the acutal numerical value of i has no meaning. The valid operations for this type of iterator are dereferencing, copying, and incrimenting.

In main's next while loop, we erase all the entries in the map that correspond to files with unique names. The name is unique if the key is unique. The count function of STL multimap return the number of entries that have the supplied key. We want to erase those entries using the erase function (i.e. cout() == 1). But wait! After we erase an entry with m.erase(i), iterator i is no longer valid. So, we store the value of i in a tempory variable j that we can use the next time through the loop.

The last thing to do in main is to type out the list of duplicate file names we wanted in the first place. The final while loop does just this.

The other three functions in DupesMain.cp basically deal with the ugliness of the Macintosh file manager. This is the kind of code I like to write with one eye on THINK Reference and the other eye on my debugger. CatalogADirectory is a recursive function that starts at an indicated folder and traverses all the contained folders. In doing so it fills up the multimap global with the names of all the files it finds.

NextFileInDirectory is a tiny helper function for CatalogADirectory. It handles the logic of moving through a folder and makes the actual call to the PBGetCatInfo Macintosh toolbox function. For more information on PBGetCatInfo and how to use it consult Inside Macintosh or THINK Reference.

DirID2FullPath converts a Macintosh directory ID to the corresponding full path name. Note that it uses macString's += operator to conveniently concatenate the folder names to produce thefull path.

I hope you can see how much STL increase your productivity. Without STL's multimap class, writing Dupes would have been a daunting task. With STL, it took me about half a day to get the main functionality fo Dupes up and running. And, as a bonus we get the macString class which should be useful in a variety of contexts.

Appendix: Annotated STL Bibliography

Graham Glass and Brett Schuchert, "The STL <PRIMER>", Prentice Hall PTR, 1996. At 327 pages, this is the most concise of the STL books I've seen. In spite of the word primer in the title, it assumes more sophistication on the part of the reader than the other texts. It also pushes the ObjectSpace commercial library, an extension of the ANSI STL. The source for demo programs is on the Symantec Developer's Advantage CD-ROM and is available on the World Wide Web.

David R. Musser and Atul Saini, "STL Tutorial and Reference Guide", Addison Wesley, 1996. This book includes excellent example programs that use STL and generic programming techniques. The tutorial section is very good. The reference section is very formal.

Mark Nelson, "C++ Programmer's Guide to the Standard Template Library", IDG Books, 1995. For my taste this is the best single book on STL. The tutorial introduction starts slowly and covers everything thoroughly. Each entry in the reference section includes an illustrative source code sample. An MS-DOS formatted floppy with STL and the source for the example programs is bundled with the book.

Alexander Stepanov, "The Standard Template Library", BYTE Magazine, October 1995, Volume 20, Number 10, October 1995. Brief introduction and historical perspective straight from STL's primary author.

Dan Zigmond, "Generic Programming and the C++ STL", Dr. Dobbs's Journal, issue #233, August 1995. Includes good illustrative source code.

Al Stevens, "Alexander Stepanov and STL", Dr. Dobbs's Journal, March 1995. An interview with STL's primary author. Contains many historical insights and some speculation about possible future additions to the library.

Al Stevens, "The Standard Template Library", Dr. Dobbs's Journal, April 1995. A very brief introduction with useful source.

butler.hpl.hp.com -- The mother lode for the latest STL.

```
/*============================================================================*/
Building Dupes

Dupes is built from a Symantec C++, version 8 (from the SDA release 5
CD-ROM) project for PowerPC only.  The project has four source files:
DupesMain.cp, macString.cp and DemoSTR.r.  There are also seven runtime
libraries: InterfaceLib, MathLib, PPCANSI.o, PPCCPlusLib.o, PPCIOStreams.o,
PPCRuntime.o and StcCLib.o.  The output is to a console Window and is
echoed to a log file.
```

```
/*================================================================================*/

// Listing 1: DupesMain.cp

#define OLDROUTINELOCATIONS 0
#include <ConditionalMacros.h>
#include <Types.h> // For Str255
#include <Dialogs.h> // For Alert
#include <TextUtils.h> // For c2pstr
#include <PLStringFuncs.h> // For PLstrcpy
#include <ToolUtils.h> // For BitTst
#include <Files.h>

#include <console.h>

#include <assert.h>
#include <stdlib.h>
#include <iostream.h>

#include "macString.h"

#define rUserAlert 129

typedef multimap<macString,macString, less<macString> > mmap;
mmap m;

static void CatalogADirectory(long);
static void DirID2FullPath(long, macString&);
static Boolean NextFileInDirectory(long, HFileInfo&, macString&);

long   countFiles = 0;
long   countDirectories = 0;

int main()
{
      console_options.ncols = 135;  // Works for 17" monitor.
      console_options.nrows = 50;  // Works for 17" monitor.
      cecho2file("Dupes.log", 0, stdout);

      cout << "Dupes: Examining the default volume..." << endl;
      cout << endl;

      CatalogADirectory(fsRtDirID);     // start at the root
      cout << endl;

      cout << "Total Directories = " << countDirectories << endl;
      cout << "Total Files       = " << countFiles << endl;
      cout << endl;

      // At this point all the files are in mmap, indexed by the file name
      // Print out in sorted order
      mmap::iterator i;
      i = m.begin();
      while (i != m.end ())
      {
            cout << (*i).first << " -> " << (*i).second << endl;
            i++;
      }
      cout << endl;

      // Remove all the entries that have unique keys
      mmap::iterator j;
      i = m.begin();
```

```
        while (i != m.end ())
        {
                const macString& key = (*i).first;
                if (m.count(key) == 1)
                {
                        j = i;
                        j++;
                        m.erase(i);
                }
                else
                {
                        for (int ii = m.count(key); ii > 0; ii--)
                        {
                                i++; j++;
                        }
                }
                i = j;
        }

        // print out the list of files with duplicate names
        i = m.begin();
        while (i != m.end ())
        {
                cout << (*i).first << " -> " << (*i).second << endl;
                i++;
        }
        cout << endl;

        return EXIT_SUCCESS;
}

static void CatalogADirectory(
        long iDir)
{
        HFileInfo     fileInfo;
        macString    sName;
        macString     sDir;
        macString    sFull;

        DirID2FullPath(iDir, sDir);
        cout << sDir << endl;

        fileInfo.ioFDirIndex = 0;
        fileInfo.ioVRefNum = 0;
        while (NextFileInDirectory(iDir, fileInfo, sName)){
                if (fileInfo.ioFlAttrib & ioDirMask)
                {       // another directory, recurse
                        countDirectories++;
                        CatalogADirectory(fileInfo.ioDirID);
                }
                else
                {       // file, add to catalog
                        countFiles++;
                        sFull = sDir + sName;
                        cout << sFull << endl;
                        m.insert (pair<const macString, macString> (sName, sFull));
                }
        }
}
```

```
static void DirID2FullPath(
	long dirID,
	macString& sFullPath)
{
	DirInfo	dirInfo;
	Str255 str255Dir;
	OSErr err;
	macString sTemp;

	sFullPath = "";
	dirInfo.ioDrParID = dirID;
	dirInfo.ioNamePtr = str255Dir;
	do {
		dirInfo.ioVRefNum = 0;
		dirInfo.ioFDirIndex = -1;
		dirInfo.ioDrDirID = dirInfo.ioDrParID;
		err = PBGetCatInfo((CInfoPBRec *)&dirInfo, false);
		sTemp = sFullPath;
		sFullPath = str255Dir;
		sFullPath += ":";
		sFullPath += sTemp;
	} while (dirInfo.ioDrDirID != fsRtDirID);  // The root, e.g. "pb:"
}

static Boolean NextFileInDirectory(
	long iDir,
	HFileInfo& fileInfo,
	macString& s)
{
	fileInfo.ioFDirIndex++;
	fileInfo.ioDirID = iDir;
	fileInfo.ioNamePtr = (unsigned char *)s;
	return  PBGetCatInfo((CInfoPBRec *)&fileInfo, false) == noErr;
}

/*==========================================================================*/

// Listing 2: macString.h

#include <stl.h>

// The macString class stores a string and is accesible either as a Pascal
//     string or as a c-string. The data is stored in str, a vector<char> and also
//     in parallel in a Pascal string.
// Bugs:
//     Should issue warnings when a macString with over 255 characters is accessed
//    as a unsigned char * as this could result is truncation and data loss.
class macString{
public:
	macString() : bPStrOK(true), bCStrOK(true), s(256) {}
	macString(const macString&);
	macString(char c) : bPStrOK(false), bCStrOK(true), s(256) {s[0] = c;
		synchronize();}
	macString(const char *s);
	macString(const unsigned char *s);

	operator char *() {return str();}
	operator const char *() {return conststr();}
	operator unsigned char *() {return pstr();}
	operator const unsigned char *() {return constpstr();}
	friend ostream& operator<< (ostream& os, macString& ms)
		{os << ms.conststr(); return os;}
	friend ostream& operator<< (ostream& os, const macString& ms)
		{os << ms.conststr(); return os;}
```

```
        friend operator< (macString& ms1, macString& ms2)
                {return strcmp(ms1.conststr(), ms2.conststr()) < 0;}
        friend operator< (const macString& ms1, const macString& ms2)
                {return strcmp(ms1.conststr(), ms2.conststr()) < 0;}
        friend operator== (macString& ms1, macString& ms2)
                {return !strcmp(ms1.conststr(), ms2.conststr());}
        friend operator== (const macString& ms1, const macString& ms2)
                {return !strcmp(ms1.conststr(), ms2.conststr());}
        macString& operator+=(const macString &);
        friend macString operator+ (macString& ms0, macString& ms1);
        macString& operator= (const macString&);
        void dump(void);
private:
        bool bCStrOK;
        bool bPStrOK;
        vector<char> s;
        Str255 str255;
        const char* conststr() const;
        char* str();
        const unsigned char* constpstr();
        unsigned char* pstr() {synchronize(); bCStrOK = false; return str255;}
        void synchronize();
};

/*==============================================================================*/

// Listing 3: macString.cp

#define OLDROUTINELOCATIONS 0
#include <ConditionalMacros.h>
#include <TextUtils.h> // For c2pstr
#include <PLStringFuncs.h> // For PLstrcpy

#include <assert.h>
#include <iostream.h>
#include <stl.h>

// This is the Symantec C++ 8.1 way to instantiate templates.  Don't forget the
//      ';' on the end of the line
template class vector<char>;
template void uninitialized_fill_n(char *,unsigned int,const char&);
template char *uninitialized_copy(char *,char *,char *);
template char *copy_backward(char *,char *,char *);
template char *copy(const char *,const char *,char *);
template char *uninitialized_copy(const char *,const char *,char *);
template void fill_all(char *,char *,const char&);
template void __distance(rb_tree<macString,pair<const macString,macString>,
        select1st<pair<const macString,macString>,macString>,
        less<macString>>::const_iterator,rb_tree<macString,
        pair<const macString,macString>,
        select1st<pair<const macString,macString>,macString>,
        less<macString>>::const_iterator,unsigned int&,bidirectional_iterator_tag);
template void __distance(rb_tree<macString,pair<const macString,macString>,
        select1st<pair<const macString,macString>,macString>,
        less<macString>>::iterator,
        rb_tree<macString,pair<const macString,macString>,
        select1st<pair<const macString,macString>,macString>,
        less<macString>>::iterator,unsigned int&,bidirectional_iterator_tag);
template char *copy(char *,char *,char *);
template insert_iterator<vector<char>> copy(const char *,const char *,
        insert_iterator<vector<char>>);
template insert_iterator<vector<char>> copy(char *,char *,
        insert_iterator<vector<char>>);
```

```
#include "macString.h"

#define VERBOSE 0

macString::macString(
       const macString& ms)
       : bCStrOK(true), bPStrOK(false), s(256)
{
       macString &msIn = (macString)ms;
       msIn.synchronize();
       s = ms.s;
       synchronize();
}

macString::macString(
       const char *sIn)
       : bCStrOK(true), bPStrOK(false), s(256)
{
       s.reserve(strlen(sIn) + 1);
       strcpy(s.begin(), sIn);
       synchronize();
}

macString::macString(
       const unsigned char *sIn)
       : bCStrOK(false), bPStrOK(true), s(256)
{
       PLstrcpy(str255, sIn);
       synchronize();
}

const char* macString::conststr() const
{
       macString& ms = (macString)*this;
       ms.synchronize();
       return s.begin();
}

const unsigned char* macString::constpstr()
{
       macString& ms = (macString)*this;
       ms.synchronize();
       return str255;
}

void macString::synchronize()
{
       assert(bCStrOK || bPStrOK);
       assert(s.capacity() >= 256);
       if (!bCStrOK)
       {
              PLstrcpy((unsigned char *)s.begin(), str255);
              p2cstr((unsigned char *)s.begin());
              bCStrOK = true;
       }
       if (!bPStrOK)
       {
              strncpy((char *)str255, s.begin(), 255);
              c2pstr((char *)str255);
              bPStrOK = true;
       }
}
```

```
void macString::dump(void)
{
	assert(bCStrOK || bPStrOK);
	assert(s.capacity() >= 256);
	cout << "in macString::dump" << endl;
	cout << "                this = " << this << endl;
	cout << "        s.capacity() = " << s.capacity() << endl;
	cout << "   strlen(s.begin()) = " << strlen(s.begin()) << endl;
	cout << "-->" << *this << "<--" << endl;
}

macString& macString::operator+=(const macString &msc)
{
	size_t len;

	synchronize();
	macString& ms = (macString)msc;
	ms.synchronize();
	len = strlen(s.begin()) + strlen(ms.s.begin()) + 1;
	s.reserve(len);
	strcat(s.begin(), ms.s.begin());
	bPStrOK = false;
	synchronize();
	return *this;
}

macString operator+ (
	macString& ms0,
	macString& ms1)
{
	ms0.synchronize();
	ms1.synchronize();
	macString ms(ms0);
	ms += ms1;
	return ms;
}

char* macString::str()
{
	synchronize();
	bCStrOK = false;
	return s.begin();
}

macString& macString::operator=(const macString &mscIn)
{
	macString& msIn = (macString)mscIn;
	msIn.synchronize();
	s = msIn.s;
	bCStrOK = true;
	bPStrOK = false;
	synchronize();
	return *this;
}

/*==============================================================================*/
```

```
// Listing 4: DemoSTL.r

#define SystemSevenOrLater 1
#include "systypes.r"
#include "types.r"

#define rUserAlert 129

/* this ALRT and DITL are used as an error screen */
resource 'ALRT' (rUserAlert, purgeable) {
	{40, 20, 150, 260},
	rUserAlert,
	{
		OK, visible, silent,
		OK, visible, silent,
		OK, visible, silent,
		OK, visible, silent
	},
	alertPositionMainScreen
};


resource 'DITL' (rUserAlert, purgeable) {
	{
		{80, 150, 100, 230},
		Button {
			enabled,
			"OK"
		},
		{10, 60, 60, 230},
		StaticText {
			disabled,
			"^0"
		},
		{8, 8, 40, 40},
		Icon {
			disabled,
			2
		}
	}
};
```