

According to Hoyle...

<http://www.jonhoyle.com> Copyright ©2007 Jonathan Hoyle

Cross-Platform Software Development from a Macintosh Perspective: Multi-Compiler Strategies with C/C++ (Part III) - The ANSI C/C++ Libraries

by Jonathan Hoyle

jhoyle at macCompanion dot com

macCompanion

February 2006

We continue into the new year with our investigation of cross-platform strategies from a Macintosh perspective. The following are the topics we have covered so far:

- **Intro:** http://www.maccompanion.com/archives/september2005/Columns/According_to_Hoyle_1.htm
- **Qt:** <http://www.maccompanion.com/archives/october2005/Columns/AccordingtoHoyle.htm>
- **wxWidgets:** <http://www.maccompanion.com/archives/november2005/Columns/AccordingtoHoyle.htm>
- **CPLAT:** <http://www.maccompanion.com/archives/december2005/Columns/AccordingtoHoyle.htm>
- **REALbasic:** <http://www.maccompanion.com/archives/january2006/Columns/AccordingtoHoyle.htm>
- **Runtime Revolution:** <http://www.maccompanion.com/archives/february2006/Columns/AccordingtoHoyle.htm>
- **AMPC:** <http://www.maccompanion.com/archives/march2006/Columns/AccordingtoHoyle.htm>
- **Java compilers:** <http://www.maccompanion.com/archives/april2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part I):** <http://www.maccompanion.com/archives/may2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part II):** <http://www.maccompanion.com/archives/june2006/Columns/AccordingtoHoyle.htm>
- **Converting Legacy Frameworks:** <http://www.maccompanion.com/archives/july2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part III):** <http://www.maccompanion.com/archives/october2006/Columns/AccordingtoHoyle.htm>
- **C++ Application Programming with REALbasic:** <http://www.maccompanion.com/archives/november2006/Columns/AccordingtoHoyle.htm>
- **Multi-Compiler strategies with C/C++ (Part I):**
<http://www.maccompanion.com/archives/december2006/Columns/AccordingtoHoyle.htm>
- **Multi-Compiler strategies with C/C++ (Part II):**
<http://www.maccompanion.com/archives/january2007/Columns/AccordingtoHoyle.htm>

This month we conclude our three part look into Multi-Compiler Strategies with C/C++ with an examination of the ANSI C and ANSI C++ Standard Libraries. These libraries provide the developer with a large number of cross-platform tools that are essential to the developer working with more than one operating system. When there exists a choice between cross-platform and Mac-only functions, it behooves the developer to understand the differences between them.

The ANSI C Library

The ANSI C Library is composed of collections of functions grouped together under different header files by category. Most C developers are already quite familiar with them, and these include:

```
#include <stddef.h>: size_t, wchar_t, NULL, etc. // standard types
#include <stdio.h>: printf(), scanf(), fopen(), etc. // I/O & file functions
#include <stdlib.h>: malloc(), free(), atoi(), etc. // allocation related functions
#include <string.h>: memcpy(), strcpy(), strcat(), etc. // string & memory functions
#include <math.h>: sin(), log(), floor(), ceil(), etc. // mathematical functions
#include <stdbool.h>: bool, true, false, etc. // boolean types
#include <stdint.h>: int16_t, int32_t, int64_t, etc. // integral types
// etc.
```

From a cross-platform perspective, it is usually better to use these Standard Library types and functions than operating system-specific equivalents when possible. Sometimes these OS equivalents are identical in nature and require merely a name change. An example of this would be Apple's definition of integral types found in `<OSTypes.h>`, which include `SInt16` and `UInt32` instead of `int16_t` and `uint32_t`, respectively. Others are completely different implementations, such as the ANSI Standard `bool` versions Apple's `boolean`. In this case, you must convert return values back and forth between Apple API calls and Standard C Library calls.

Standard C Library functions are usually preferred for the same reasons: `malloc()` and `free()` are preferred over `NewPtr()` and `DisposePtr()`. Likewise, `memcpy()` and `memmove()` are preferred over `BlockMove()` and `BlockMoveData()`. However, sometimes you may require the Apple equivalents for performance or functionality not available from the ANSI functions. For example, the Standard `fopen()` would be insufficient if you require your file to have a resource fork. Therefore, always be mindful of the requirements of your project prior to making these decisions.

The ANSI C Library within C++

The Standard C Library functions and types as defined by ANSI in 1990 are available as part of the Standard C++ Library for free. Although newer items added to the Standard in 1999 are not officially a part of the C++ Library, many C++ development environments have included them as a courtesy.

The associate C header files of the form `#include <xxxx.h>` are deprecated in C++ and replaced by those of the form `#include <cxxxx>`. Furthermore, C Library items in C++ are inside the `std` namespace and are not available globally. Thus, C Library items require either a `std::` prefix or a declaration of `using namespace std;` For example, the C snippet:

```
#include <string.h>

memcpy(destinationArray, sourceArray, numItems * sizeof(long));
```

...in canonical C++ would be written as:

```
#include <cstring>

std::memcpy(destinationArray, sourceArray, numItems * std::sizeof(long));
```

As most developers are fairly familiar with the Standard C Library, the remainder of this article will focus on Standard C++ Library.

cout & Other <iostream> Classes

One of the first objects one learns about in C++ is `cout`, which is an instantiation of the `ostream` object. The C++ `iostream` classes are functional replacements for the clunky `<stdio.h>` functions from C. Rather than the cumbersome C snippet:

```
float radius;
double pi = 3.14159;
printf("Enter the radius of a circle:\n");
scanf("%f", &radius);
if (radius <= 0.0)
    fprintf(stderr, "Improper value returned for radius: %2.3f", radius);
```

```

else
    printf("Circumference=%2.3f, Volume=%2.3f\n", (float) (2.0*pi*radius), (float)
(pi*radius*radius));

```

...the C++ equivalent code is more obvious:

```

float radius;
double pi = 3.14159;
cout << "Enter the radius of a circle:" << endl;
cin >> radius;
cout.precision(5);
if (radius <= 0.0)
    cerr << "Improper value returned for radius: " << radius << endl;
else
    cout << "Circumference=" << 2.0*pi*radius << ", Volume=" << pi*radius*radius << endl;

```

The C++ code requires no translation of the unusual embedded text tokens and reads more like English. Other formatting capabilities are also available, including setting output into various formats (octal, hexadecimal, scientific notation, etc.) and wide character usage.

ANSI `string` Class & Other Useful Objects

The Standard C++ Library comes with a number of very useful cross-platform objects. For the interests of space and time, I will give only a bird's eye view of them for now, and we will go into detail in future articles.

Virtually all frameworks offer a basic string class type. Such classes are certainly helpful to the user of the framework and will typically offer behavior which takes advantage of the given framework. Unfortunately, such classes are of very little use once you need them to be cross-platform. The one string class which is available to all C++ programmers is the ANSI Library class `string`. It has basic constructors accepting standard `char *`'s as well as other `strings`, and you may wish to create your own utility functions to convert to and from this type, particularly if you are dealing with other string types, such as MFC's `CString`, PowerPlant's `LString` or Mac OS X's `CFString`. There is also a wide character equivalent class `wstring` for those dealing with Unicode.

In addition to the `string` class, there are a number of other extremely useful cross-platform ANSI classes provided by C++ which you will wish to use when working between multiple frameworks. One is `auto_ptr<>`, which is a templated smart pointer class. To use it, one merely allocates an object with the traditional `new` keyword and assign it to an `auto_ptr<>`. The `auto_ptr<>` class' destructor will delete the memory associated with it, so that you need not be concerned with chasing pointers. `auto_ptr<>`'s can even pass ownership around when you wish them to.

`auto_ptr<>`'s point only to one object at a time. When handling arrays of data, the ANSI Standard Library has a number of objects to choose from. For numerical data arrays, the `valarray<>` templated type is perhaps the best, as it allows the array of data to be operated upon as if it were a single value. For example, if `val1` and `val2` are two `valarray`'s then `val3 = val1 * val2` would be an array of the products of the entries of `val1` and `val2` for each index. All manner of arithmetic can be performed on `valarray<>`, including standard math functions: `val2 = sin(val1)` creates a `valarray<>` which is a collection of the results of applying the `sin()` function upon each element of `val1`. "Slices" of `valarray<>`'s can also be extracted. `valarray<>`'s are highly optimized for mathematical calculations and are best suited for numerical types.

Another powerful mathematical part of the ANSI Standard is the `complex` data type. Unfortunately (as briefly mentioned in last month's column), the `complex` data type in the ANSI C++ Standard conflicts with the one

defined in the ANSI C99 Standard. In C, **complex** is a suffix modifier to floating point data types, declared as such: **double complex z**; In C++ on the other hand, **complex<>** is a templated type and is declared like so: **complex<double> z**; Either version is will likely fit your needs and it is merely a matter of preference which to use.

STL Containers

One of the most powerful features of C++ Library was the inclusion of the *Standard Template Library*, or *STL*. *STL* was designed by Alexander Stepanov in an attempt to create “*uncompromisingly generic algorithms*”. It was implemented in other languages, such as Ada, prior to being ported to C++, which happened in 1994. Initially, *STL* evolved independently from the C++ Standard Library, which is why it is very different from the rest of C++. For example, while most of the Library follows standard object oriented practices of inheritance to absorb common functionality, *STL* objects are very loosely connected and share very little in the way of an inheritance hierarchy. Despite this major difference in design philosophy, the ANSI committee voted to accept most of *STL* into the Standard Library.

STL consists of these components: Containers, Iterators, Algorithms & Functions, and Adapters. Containers are different data structures used to hold objects, such as **vector** (a one dimensional array), **list** (a doubly-linked list), **queue** (a FIFO model), **stack** (a LIFO model), **deque** (a double-ended queue), and others. **iterator**'s are generalizations of pointers, including input iterators, output iterators, forward iterators, bidirectional iterators, and more. (Not all iterators are available to all containers.) Algorithms involve optimized sorting, searching, swapping, etc. The important point here is that algorithms are container-independent.

STL is designed for performance, not for object design. This is why there is no common base class amongst the containers and why each container has its own iterator type. No special range checking is done, as target performance is the priority.

With few exceptions, *STL* has virtually all the container classes that one would need for an advanced project. The only notable piece missing is a sparse array class. In a cross-platform project, you would do very well in using *STL* objects as the universal underlying model, translating into framework specific containers only when necessary.

Coming Up: With this 16th installment, we will stop here in our cross-platform series and move onto more Macintosh-specific areas of interest. I hope you have enjoyed reading this series as much as I have in writing it. With Mac OS X 10.5 Leopard just around the corner, there are a number of exciting new topics that we will be delving into in the coming months. Please feel free to send your feedback and suggest and topics for the future. See you in 30!