

According to Hoyle...

<http://www.jonhoyle.com> Copyright ©2006 Jonathan Hoyle

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

by Jonathan Hoyle

jhoyle at macCompanion dot com
macCompanion
December 2006

We are in our second year of investigating cross-platform strategies from a Macintosh perspective. We have examined a various number of frameworks, tools, compilers over the course of these many months:

- **Intro:** <http://www.macompanion.com/archives/september2005/Columns/AccordingtoHoyle1.htm>
- **Qt:** <http://www.macompanion.com/archives/october2005/Columns/AccordingtoHoyle.htm>
- **wxWidgets:** <http://www.macompanion.com/archives/november2005/Columns/AccordingtoHoyle.htm>
- **CPLAT:** <http://www.macompanion.com/archives/december2005/Columns/AccordingtoHoyle.htm>
- **REALbasic:** <http://www.macompanion.com/archives/january2006/Columns/AccordingtoHoyle.htm>
- **Runtime Revolution:**
<http://www.macompanion.com/archives/february2006/Columns/AccordingtoHoyle.htm>
- **AMPC:** <http://www.macompanion.com/archives/march2006/Columns/AccordingtoHoyle.htm>
- **Java compilers:** <http://www.macompanion.com/archives/april2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part I):**
<http://www.macompanion.com/archives/may2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part II):**
<http://www.macompanion.com/archives/june2006/Columns/AccordingtoHoyle.htm>
- **Converting Legacy Frameworks:**
<http://www.macompanion.com/archives/july2006/Columns/AccordingtoHoyle.htm>
- **Basic compilers (Part III):**
<http://www.macompanion.com/archives/october2006/Columns/AccordingtoHoyle.htm>
- **C++ Application Programming with REALbasic:**
<http://www.macompanion.com/archives/november2006/Columns/AccordingtoHoyle.htm>

This month we will look at C/C++ programming from the perspective of using multiple compilers. Cross-platform developers will typically rely upon different compilers from different vendors, each with their own level of language support and bugs. Even within an operating system, there may be many compilers to choose from, and being able to run your code from each of these makes you more robust. This article is Part I on multi-compiler strategies.

Be ANSI Compliant Wherever and Whenever Possible

Unless a particular ANSI feature is simply not available in one of the compilers you need to support, always follow the ANSI standard. A number of developers shy away from supporting the latest standard features, fearing incompatibility down the road with future development environments. Although this is a small risk, it is far outweighed by the fact that C/C++ compilers are constantly being updated to be in closer conformance to the ANSI/ISO standard. Supporting a non-ANSI feature is a much bigger risk for a code base which you wish to keep supporting. The ANSI committee ratified the C++ standard in 1998, and updated the C standard in 1999, and it is these standards to which you should be adhering.

The flip side to following ANSI is to ignore features or extensions that are not ANSI standard. For example, *gcc* (the compiler used in the *Xcode* environment) has switches to support a number of language extensions. Although tempting, the use of these features will quickly become a difficulty once you try compiling with a different environment. A number of these extensions have already been rolled into the C99 specification, but those which have not should be avoided.

Also to be avoided are the use of compiler-defined or operating system defined types that have ANSI equivalents already in existence. The most obvious example of this are the multitude of boolean types that exist, which confusingly take on many more than the two values it should. For example, on the *Macintosh*, the **Boolean** type takes on 256 possible values with **True** and **False** defined for two of those values (although the other 254 possibilities remain). It's even worse on *Windows* with its **BOOL** type, as it takes on over 4 billion values more than the predefined **TRUE** and **FALSE**. It's frightening to see code reading

```
if (myBOOL == TRUE) { ... }
```

when you consider that **myBOOL** could easily be neither **TRUE** nor **FALSE**. The ANSI defined bool type, however, allows for only two choices: true and false, and you can safely write

```
if (mybool == true) { ... }
```

without hesitation.

Fortunately, the development environments themselves will help keep you in conformance. In *Xcode*, turn on C99 and turn off non-standard language extensions. In *Metrowerks CodeWarrior*, check the "ANSI Strict" checkbox in the project settings. For other compilers, read the manual for project settings.

Integral types

Although the ANSI C language defines the **short**, **int**, **long**, etc. types, it is implementation dependent as to what those actual sizes truly are. Until very recently, most compilers defined **short** as 16-bits wide, **long** as 32-bits wide and **long long** as 64-bits wide. In years past, **int** was often two bytes, but today it is more commonly four. Likewise, pointers are commonly 32-bits wide, although they had been shorter a decade and more ago.

With the coming of 64-bit compilation, many of these conventions are changing yet again. As you might expect, pointers will be widened from four bytes to eight to allow for 64-bit addressing. What you may not be aware of is the fact that some integral types are changing. In *Xcode* for example, not only is the pointer size increasing to eight bytes, but so is the definition of **long**. This is called the *LP64* standard (**LP** = **L**ong, **P**ointer). On many Unix/Linux compilers, the situation is worse, as they follow the *ILP64* standard (meaning that regular **int**'s themselves, as well as **long**'s and pointers), will be 64-bits wide. *Microsoft* is doing the sensible thing (surprisingly) for its *Visual Studio* and not changing any integral types for 64-bit compilation; after all, 16-, 32-, and 64-bit wide integers are already available in the 32-bit world via **short**, **long** and **long long** (respectively), so they felt no need to change anything. This is called the *P64* (alternatively, the *LLP64*) standard.

Given the anarchy of integral sizes, one wonders if you can rely on any of these sizes to remain stable. Well, simply put, you cannot. You can however, use the ANSI defined types **int16_t**, **int32_t**, **int64_t**, etc. as defined in the `<stdint.h>` header file. These types, available in both signed and unsigned flavors, are defined to always be the size desired. Most all modern compilers which support the C99 standard include this header file. In the event you must support an older compiler, such as the unfortunate *Visual C++ 6* environment, you can always create your own **stdint.h** file, **typedef**-ing these names yourself.

The problem continues beyond just 64-bit issues. For example, ANSI does not specify the size of **bool**. In some compilers, it is the same as whatever is set for **int**. In other compilers, it is only one byte wide. For this reason, it is not recommended to use **bool** within any structures that will be shared outside your program (such as placed in shared memory or written out to disk). Use one of the integer types to hold the value when writing out to a structure.

Floating point numbers and their format are not detailed by the ANSI or ISO committees, which led to a wide variance in floating point types in decades past. However, another standards committee, IEEE, did outline a particular format, and most compilers today follow the IEEE single precision (four byte) and double precision (eight byte) floating point format for their **float**'s and **double**'s respectively. Due to *Microsoft's* slow adoption of the **long double** floating point type (they essentially make it the same as a regular **double**), it is not recommended to use **long double**'s at this time.

Finally, one type which is easily overlooked is the wide character type **wchar_t**, used for Unicode strings. Operating system difference make this type particularly difficult to use, as a **wchar_t** is two bytes in *Windows*, but four bytes on *Unix* and *Mac OS X*. However, with the importance of Unicode support growing with each passing year, this is one type that really has no workaround.

Warning! Warning! Danger, Will Robinson!

One of the best ways to save hours of debugging is cut it off at the pass by turning up warnings as high as you can reasonably allow. This is one of the benefits of compiling your code across multiple development environments, as different compilers find different types of warnings. Furthermore, make it a policy to rewrite your code to prevent these warnings from occurring. Now granted, a warning is not an error, and often times what is being warning is exactly what you meant. *CodeWarrior* used to warn you when an embedded assignment is found within a while or if statement. Maybe it was what you intended, but there's no reason you can't be more explicit and separate them into different lines of code. *Visual Studio* likes to warn you about a possible loss of data when assigning a **long** to a **short**. Simply adding a cast removes that warning. The more safety nets you put in with warnings, the safer and more robust your code will be.

In addition, always refactor out deprecated function calls or behavior. Continuing to use deprecated code is a time bomb waiting to happen. If you don't fix it now when you have the chance, you'll be forced to do it later, and likely when you're in a hurry. Whether it's operating system level items like *QuickDraw* calls, or language-based behavior such as lacking function prototypes, you'll be happier once you've made the jump. Often times the replacement code is a simple structural rewrite, such as removing some nasty **goto**'s and replacing them with a clean exception handling mechanism.

Always keep your compilers up to date. When dealing with very old code, perform your upgrades one version at a time, as this will make life much easier. Recently, I was in discussion with someone who has a old *CodeWarrior Pro 5* project he wanted updating. Jumping from *CW Pro 5* to *Xcode 2.4* would probably end the project in utter frustration. However, upgrading *CodeWarrior* versions one at a time, from *Pro 5* to *Pro 9* allowed him to make changes more simply and to test these changes on an iterative basis. Once at *Pro 9*, he could then use *Xcode 2's* importer to convert the *CodeWarrior* project.

Coming Up: More multi-compiler tips and additional best practices for cross-platform code. See you in 30!