

According to Hoyle...

Moving from CodeWarrior to Xcode (Part I)

by Jonathan Hoyle

jhoyle@maccompanion.com

April 2007

One year ago this month, Metrowerks ceased all development and support of its CodeWarrior product line for the Macintosh.

The Rise and Fall of Metrowerks CodeWarrior

Back in the days of 68K Macs, Metrowerks was a small education market compiler developer, known principally for its Pascal and Modula-2 products on the Macintosh and MIPs. With the advent of the Power Macintosh, all of that changed. Metrowerks introduced CodeWarrior by 1993's year's end, the only user-accessible development environment that could create native PowerPC applications. CodeWarrior offered three front end languages, Pascal, C and C++, and had two back-ends that could either 68K or PowerPC. Later versions of CodeWarrior would offer additional language and processor support, such as Java and Win32 development. Metrowerks also introduced the PowerPlant C++ class framework, In less than two years, Metrowerks went from niche to market dominance, completely changing the face on Macintosh software development. For the next decade, no other tools developer (including Apple) could touch Metrowerks.

Metrowerks grew, adding support for a number of new environments, went public and was eventually purchased by Motorola for its innovative compiler technologies. Throughout this time, Metrowerks remained a faithful Apple partner, even during the fiascos of Copland, YellowBox and Rhapsody. With the evolution of Mac OS X, CodeWarrior evolved with Carbonized versions of its compiler and PowerPlant. However, in 2003, the face of Metrowerks began to change. Motorola was in its initial stages of spinning off portions of the company to Freescale, a company which had no respect or understanding of Metrowerks.

For the first time in 10 years, Metrowerks failed to even make an appearance at Apple's Worldwide Developer's Conference in 2003. Things went from bad to worse in 2004 when Metrowerks was absent again from WWDC, and for the first time in CodeWarrior history, failed to deliver its annual Macintosh release. In early 2005, Metrowerks sold off its Intel compiler technology such weeks prior to Steve Jobs' announcing Apple's move away from PowerPC and to Intel. The trickle became a flood as Macintosh developers abandoned CodeWarrior to switch to Xcode. At MacHack 2005, Metrowerks announced that they were cancelling their Macintosh product line, effective March 31, 2006.

Migration to Xcode

Today, all new Macintosh computers are built with the Intel processor. To take advantage of this processor's speed, applications must be built as Universal Binaries, that is, containing both PowerPC and Intel-based compiled code. With Metrowerks' departure from the Macintosh (and Windows) development market, only Xcode can be used to create these Universal binaries. Most Macintosh developers have already begun the process of migrating their projects from CodeWarrior to Xcode. Apple has provided a document free to download describing this process:

<http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/MovingProjectsToXcode.pdf>

The purpose of this article is not to duplicate this documentation, but rather to go into greater detail from a CodeWarrior user's perspective. Although some information in here may be repeated from that document, I will also offer some of my personal recommendations, as one who has had to port a few projects of various sizes and vintages.

I will be making the following assumptions about your project for this article:

1. You are currently using CodeWarrior Pro 8.3, 9.x, or 10.
2. You are building your application on Mac OS X 10.3 or higher.
3. Your project is already Carbonized and is generating a Mac OS X native executable.
4. Your project is written in C/C++.
5. You are interested in porting your project to Xcode 2.4 or higher
6. You wish to build your application as a Universal Binary to be run on Mac OS X 10.3.9 or higher

Pre-Flighting Your CodeWarrior Project

Many of the compatibility problems that exist between CodeWarrior and Xcode can be avoided by making changes in your CodeWarrior project prior to importing it into Xcode. Simply importing your CodeWarrior project as is without modification will increase the debugging burden in Xcode. This is an acceptable solution if your knowledge and experience with Xcode is greater than that of CodeWarrior. However, most developers have had many more years of experience with CodeWarrior as compared with Xcode. It is easier to leverage your own experience and knowledge in the platform you know for those things you can, so as to reduce your difficulty on the new platform.

Don't Mach Me

If your project is anything like the typical CodeWarrior project, yours is probably a CFM-based Carbon application. This is likely because your application began as a traditional Classic app and evolved with time. Its most recent overhaul would be the Carbonization effort making it native to OS X. Mac OS X on PowerPC supports two native executable types: PEF (for CFM applications with compatibility to Mac OS 9) and Mach-O (a Mac OS X-only file type). Mac OS X on Intel, however, natively supports only Mach-O. As you are probably already aware, PEF applications run fine on Intel-based Macs, but only under Rosetta emulation (with performance being about 1/3 of what it could be).

Since an Xcode 2.4 version of your project will be built as Mach-O only, you can begin making preparations for this migration with CodeWarrior now. The first step is to create a Mach-O target in your CodeWarrior project (if you haven't already done so). The benefit to doing this is to identify any compatibility problems you are likely to find when interfacing with other libraries, plug-ins and other executables. If yours is a simple application, this will be a straightforward process. However, more complicated projects often have tanglings of inter-dependencies with other compiled code. For these projects, this can be a very involved step on the way to a universal binary. By creating this separate Mach-O target, you can still deploy the PEF target as you always have, until the transition is complete.

Even if you already had the foresight to create a Mach-O target for your project, you were likely to have created the wrong kind, as you will soon see. In your project's *Target Settings* panel, you will find among in the Linker popup menu these linkers options (among others):

Macintosh PowerPC
Apple Mach-O PowerPC
Mac OS X PowerPC Mach-O

Although all three are for “Macintosh PowerPC”, you probably have guessed that only the first one is a CFM/PEF linker, due to its lack of *Mach-O* reference. Of the remaining two, which linker should you use? Aside from the arbitrary reversal of *Mach-O* and *PowerPC* in the name, what is the difference between them? With only this text before us, we are left with the bewildering conclusion that one is for *Apple* and the other is for *Mac OS X*.

As it turns out, these are two separate Mach-O linkers created independently of each other. The *Apple Mach-O PowerPC* linker is the one Apple uses in Xcode, while *Mac OS X PowerPC Mach-O* is Metrowerks’ own designed linker. This confusion in naming is one of the few areas where Metrowerks really dropped the ball with regard to ease of use. It would have been far clearer to everyone had Metrowerks simply populated this menu with more informative text, such as:

Macintosh PEF Linker
Apple Mach-O Linker
Metrowerks Mach-O Linker

Both Metrowerks linkers are far superior to Apple’s in features, most notably offering dead code stripping capabilities. Apple’s version, in addition to contributing to code bloat, is only a single pass linker, making it a bit more fragile. If you already have a Mach-O target in your CodeWarrior project, you are likely to be using the Metrowerks linker. Sadly, you will need to switch this to the *Apple Mach-O PowerPC* linker to better prepare for potential link errors which Xcode may generate. This linker will also give you an early preview of the overweight file sizes Apple’s Mach-O will be creating for you.

For more information on Mach-O target creation, visit: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/mig_bef_converting/chapter_3_section_4.html

wchar_t and wstring’s

The move to Mach-O may generate errors for projects using the **wchar_t** or **wstring** data types. There can be a couple of reasons for this. Firstly, the **wchar_t** data type is defined to be two bytes wide in CFM/PEF and Windows projects but four bytes wide in Mach-O. This discrepancy is due to changes in Unicode. The original Unicode specification was able to fit all of its symbols within 65,536 characters, which is why the original **wchar_t** data type was only two bytes in size. As later versions of Unicode outgrew this threshold, many Unix-based compilers, including gcc, changed **wchar_t** to be four bytes wide. This is a particularly ugly incompatibility for cross-platform projects, since Microsoft has refused to update their **wchar_t** size.

Another possible area of conflict is due to the support of **wchar_t** and **wstring** on Mach-O is available only with the Mac OS X 10.3 Panther SDK and higher. This is the default for CodeWarrior 9 and higher, but CodeWarrior 8 used the 10.2 Jaguar SDK which had no wide character support. Those requiring wide character support in their projects ought to upgrade to CodeWarrior 9 prior to porting to Mach-O.

CodeWarrior Settings

The CodeWarrior C/C++ Language Settings preference pane contains a number of settings which can significantly aid you in your code cleanup work. GCC 4, the internal compiler used by the Xcode 2 development environment, is more ANSI-strict than previous versions of GCC, as well as more strict than earlier versions of CodeWarrior. Thus, making your code as ANSI-compliant as possible from the beginning is extremely beneficial. You can help enforce this by making use of two of the C/C++ Language Settings: *ANSI Strict* and *ANSI Keywords Only*. By turning these checkboxes on, non-ANSI compliant code within your project will be immediately flagged at compile-time.

Another important selection in this settings panel is the *Require Function Prototypes* checkbox. Errors generated by this setting indicate that you failed to declare the function's prototype prior to its definition. For functions which exist within a single file only, you can avoid this error by defining the function as **static** and position it within the file prior to its first use. To remain ANSI compliant, you will also want to turn off *Legacy for-scoping*, as this setting allows for non-standard for-loop behavior called out by the *Annotated C++ Reference Manual*. For full C++ conformance, be sure to check ON the *ISO C++ Template Parser*, *Enable C++ Exceptions*, *Enable RTTI* and *Enable bool Support* settings, while checking OFF the *EC++ Compatibility Mode* and *Use Unsigned Chars* settings.

Two other compile-time selections are worth noting here: *Enable C99 Extensions* and *Enable GCC Extensions*. My recommendation is the reverse of what some writers have suggested: that is, I say to turn ON the former and to turn OFF the latter. C99 is an ANSI/ISO standard of the C programming language, and any of its extensions are now legitimately available to you the programmer. By turning it on, the Xcode importer will see this flag associated with your CodeWarrior project, and it shall turn this flag on for your converted Xcode project as well. GCC extensions, on the other hand, are *not* standards-compliant. Although it may appear to be a *freebie* from the perspective of a language feature, it sets a bad precedent. Just as you are now having to remove Metrowerks extensions from ancient code, you do not want to have to do the same with GCC extended code many years from now.

The C/C++ Warnings Settings pane is also a treasure chest of preventative medicine. Make it a point to turn on as many warnings as possible to identify all potential problems. I recommend turning on one at a time, and then modify your code to remove the warnings. In particular, casting warnings and undefined macros are likely to come back as errors in Xcode.

Bool Me Once, Shame On You...

The C++ standard does not specify how many bytes **bool** and **enum** types should occupy, but compiler writers have to select some size during memory allocation. CodeWarrior, presumably with a mind to memory optimization, chooses the smallest number of bytes required for storage for an **enum**, and 1 byte for a **bool**; Xcode, on the other hand, knee-jerked the size to be the same as an **int** (4 bytes). For properly written code, this difference should never matter. However, poor code can be found in the most innocuous of places. Examples of bad programming practice that would be affected by this include the imbedding of an **enum** or **bool** type within a structure, or the exporting of functions using one as a parameter or return type. Its use in normal coding, such as a local variable, or non-exported method or function, would have no impact on your application.

For **bool**, the correct solution would be to replace the improper instances with a compatible size-specified type, such as **uint8_t**, and perform the usual conversion to **true** and **false**. Alternatively, these instances can be replaced with the archaic Macintosh type **Boolean**, which is coincidentally one byte long and has the added benefit of communicating its intentions of being a boolean type. Either of these solutions, however, requires the programmer to examine all instances of his use of **bool**, which may be impractical for a large codebase. If debugging is expected to yield faster results than searching, try temporarily turning off the *Enable bool Support* language setting, and add the following macros to your headers:

```
#define bool int
#define true 1
#define false 0
```

If no problems arise with these macros while running your unit tests, then you are probably safe in your **bool** usage. You can then remove this temporary code and remember to recheck the *Enable bool Support* language setting checkbox.

Fortunately, the solution is a bit easier for **enum** types: turn on the *Enums Always Int* checkbox. This will instantly make all the enumerated types in your project 4 bytes long in CodeWarrior. You will still, of course, need to test this to ensure that you are not making any improper uses of them, but the risk is likely to be much smaller than will **bool**.

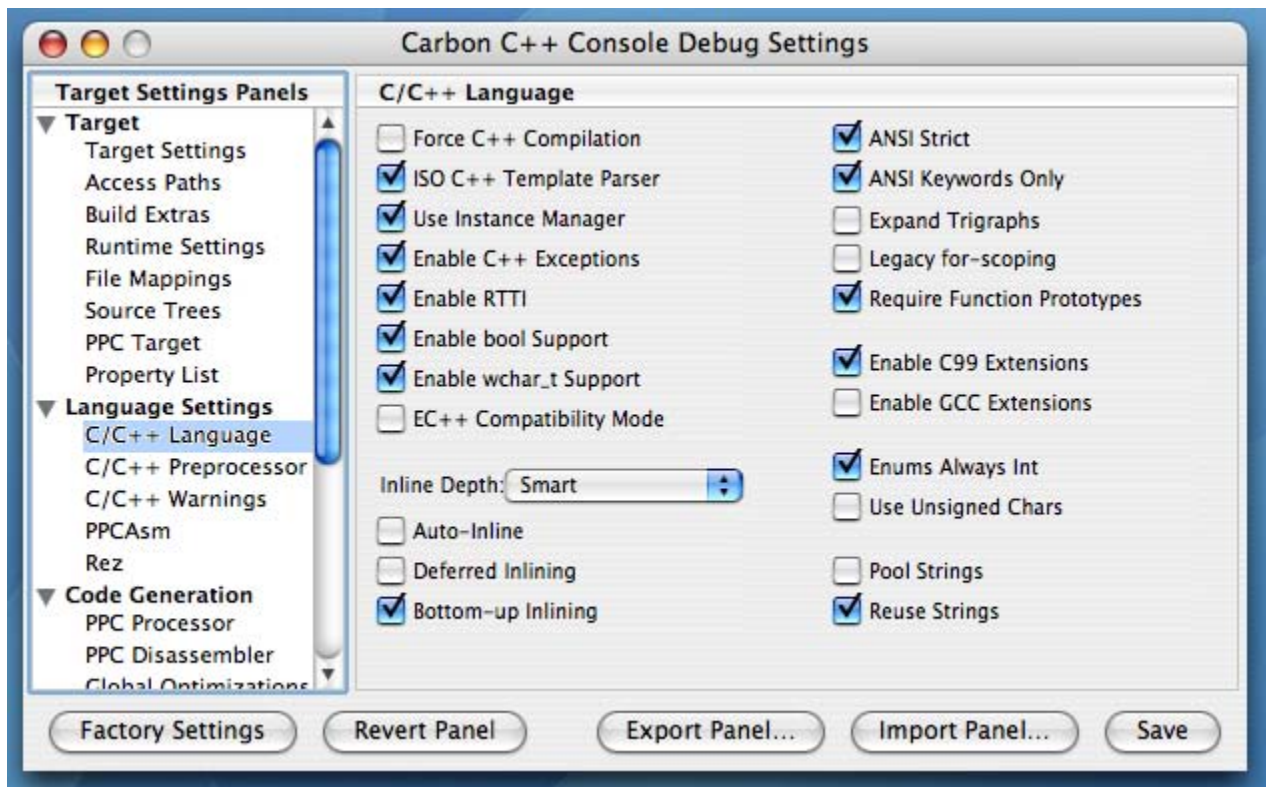
Source File Changes

Your C and C++ files will typically have **.c** and **.cpp** (or alternatively, **.cp**) extensions, respectively. Although CodeWarrior does not enforce this convention, Xcode honors the file extension more closely, so you should ensure that each of your source files has the appropriate ending. For completeness in testing, turn off the checkbox *Force C++ Compilation* from the C/C++ Language Settings pane.

For Carbon development, CodeWarrior allowed the continued use of old style Macintosh includes, such as **#include <MacHeaders.h>**. These should be replaced by the frameworks style includes, such as **#include <Carbon/Carbon.h>**.

Standard C++ header files deprecate the C-style **.h** file extension. CodeWarrior provided duplicate versions of these header files which contained the **.h** extension. Furthermore, it added the convenience of including the directive **using std;** inside these deprecated headers, so as to allow the global use of the included functions. Since these deprecated headers do not exist in Xcode, you will need to replace any that exist with the standardized equivalents.

Below is an example of what your CodeWarrior C/C++ Language Settings should appear like:



The Pragmatic Programmer

A very useful feature of the CodeWarrior development environment was its offerings of **#pragma** options. A **#pragma** is a capability provided to you by the specific compiler vendor. Metrowerks supplied them for

everything from warning suppression to function exporting. Unfortunately, gcc is rather anemic in this regard. If you use **#pragma** for anything other than informational convenience, you will need to search and find each one and replace it with appropriate code.

For specific details on **#pragma**, visit: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/migration_differences/chapter_2_section_9.html

Align in the Sand

One of the most insidious errors awaiting you has to do with struct and memory alignment differences between Xcode and CodeWarrior. If your code writes structured data out to disk or to a separate application, you are in severe danger of crashing your application. The devil is certainly in the details here, and the details are specific edge cases that appear as innocent code.

For specific details on memory alignment issues, visit: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/mig_after_importing/chapter_5_section_5.html

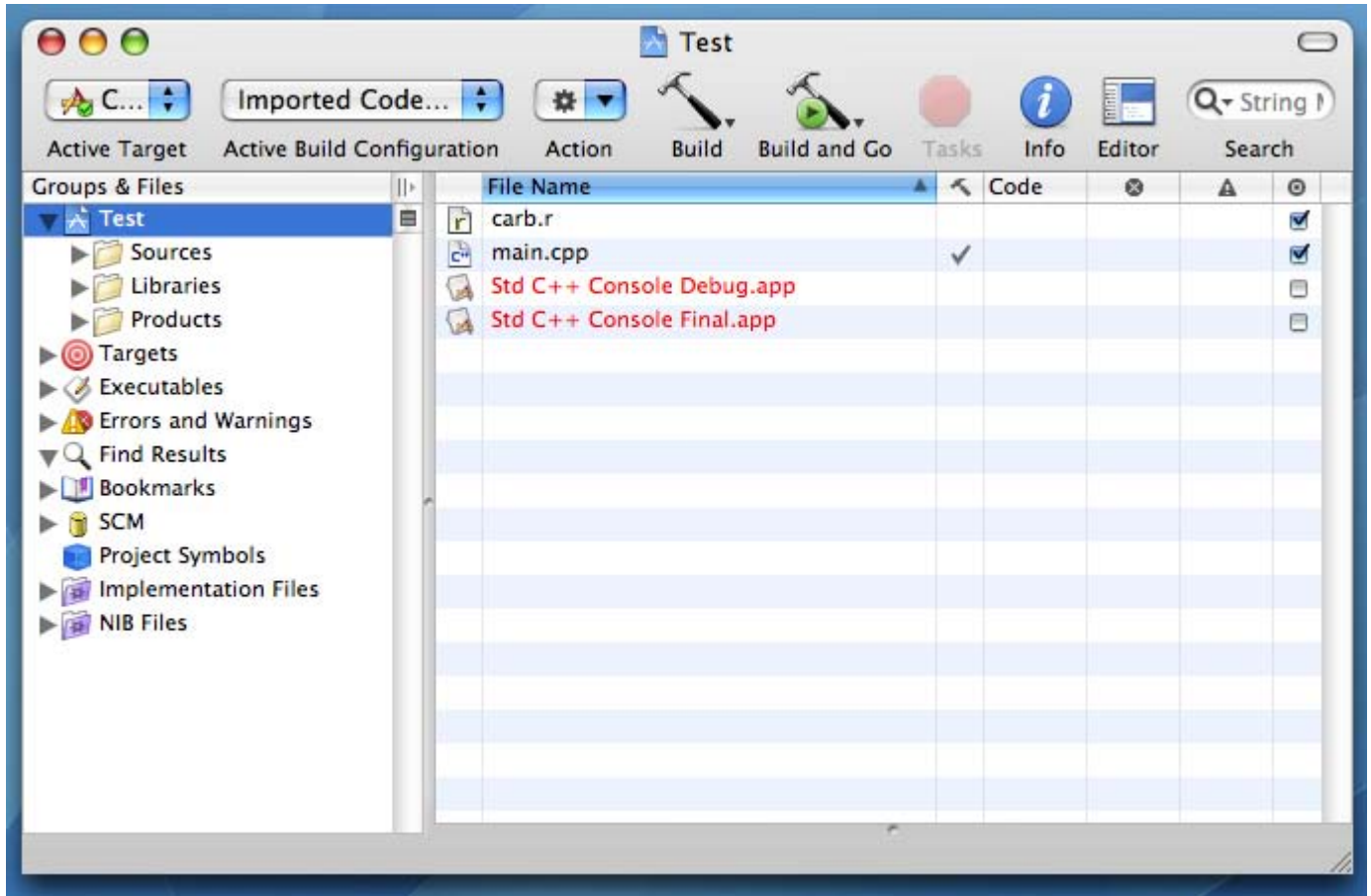
Flight Attendants, Prepare for Take-Off

You are nearly ready to import your project. There is one final caveat that Apple's documentation doesn't make very clear pertaining to CodeWarrior 10 users. CodeWarrior 10 projects can be optionally saved in a newer text-based XML format in addition to the standard format compatible with CodeWarrior 8 and 9. The Xcode Project Importer does not support new CodeWarrior 10 project file format, so you may need to re-save your project using the older format before proceeding.

At this point you are ready to throw caution into the wind and import your CodeWarrior project. Launch the latest version of Xcode available to you, preferably 2.4, and under the **File** menu, select **Import Project...** This will bring up the *Import Project Assistant*. Click on *Import CodeWarrior Project*. The next pane will allow you to select the CodeWarrior project file you wish to import. This file is typically one with a **.mcp** extension. Click on the **Choose...** button to make your selection. The Assistant should look something like this:



Click on the **Finish** button and Xcode will do its thing. Upon completion, a new project file with a **.xcodproj** extension will be generated, and a somewhat alien window will be presented to you, looking something like this:



For more information on importing a CodeWarrior project into Xcode, visit: http://developer.apple.com/documentation/DeveloperTools/Conceptual/MovingProjectsToXcode/mig_importing/chapter_4_section_3.html

Next Month: Our work is only half over. We will continue our discussion of porting your project by examining the Xcode side of the equation, as well as how to set Xcode preferences to make your development environment more civilized.