

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 II)

By Jonathan Hoyle

jhoyle at macCompanion dot com
macCompanion
January 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>

This month we continue with C/C++ coding techniques for multiple compilers. In this installment, we will be looking primarily at C code and how to make it run properly with a C++ compiler.

C, once the *lingua franca* of programming languages, is now overshadowed by its more powerful successor, C++. Development environments which compile C only (without C++) are becoming rarer and rarer with time. Authors of C code need be cognizant of this fact, particularly if they wish to keep their code maintainable moving into the future. At some point, it is likely that your C code will need to be able to run on C++ compilers, even if you have no intention of porting to C++. Fortunately, there are some basic guidelines to follow that will keep your code running happily under the watchful eye of the C++ parser. These guidelines not only bulletproof your C code for C++, they also make it better C code:

Always Use ANSI Function Prototypes and Declarations

This is the 21st century for Pete's sake. It is inexcusable and downright negligent to use these deprecated pre-ANSI declaration in this day and age. Sadly, the ANSI C99 committee chose to keep these old K&R-style function declarations legal. In case you haven't seen what they look like before, below is an example of a function declaration, followed by a function definition (something that might have been written by your grandfather 35 years ago):

```
/* Declaration of foo() - requires no parameters, implicit in return */  
foo();  
  
/* Definition of foo() - requires no parameter types */  
foo(x, y)  
x;  
float y;  
{  
    ...  
}
```

For this shameful code above, a C++ compiler will report an error, and a C compiler with reasonable integrity will at least generate a warning. First you'll notice a lack of return type; this means (bizarrely) that an **int** (not **void**, as you might expect) is being returned. Although C90 still allows the implicit **int**, C99 does not. Secondly, you'll notice that no parameters are displayed in the declaration. Using this format will prevent your C compiler from being able to performing type safety checks.

The modern version of this function declaration and definition looks like this:

```
// Declaration of foo() - prototype identical to the definition  
int foo(int x, float y);  
  
// Definition of foo() - prototype identical to the declaration  
int foo(int x, float y)  
{  
    ...  
}
```

Use `extern "C"` for C Function Declarations

Since name mangling rules differ between the languages, it is useful to wrap your function prototypes with the `extern "C"` declaration, so that the C++ compiler knows to use C naming conventions, as so:

```
#ifdef __cplusplus
extern "C"
{
#endif // __cplusplus
    void foo(int x);
    double bar(const char *p);
#ifdef __cplusplus
}
#endif // __cplusplus
```

Note the use of `#ifdef __cplusplus` wrappers, since a C compiler won't understand the invocation of `extern "C"`.

Take Advantage of C++ Features Now Available in C

ANSI C99 introduced to the C programming language a number of very nice features that was formerly available only to C++ compilers. No longer does your C code have to look cumbersome from a C++ perspective. Usability and readability are greatly improved with these new features. Here is a small list of recommendations to keep your C code in the 21st century:

1. Use the C++ style `// comments`, reserving `/* comments */` for multi-line documentation only.
2. Move your variable declarations down closer to the code it belongs to.
3. Use conditional expression declarations, such as `for (int i = 0; i < 10; i++)`, rather than declare `int i`; in the body of the code.
4. For performance, `inline` your small functions (rather than using awkward macros).
5. Use the standard `bool` type, rather than an `int`, for boolean expressions (you may need to `#include <stdbool.h>`).

Avoid C Behaviors which are Incompatible with C++

Although C++ was designed with C in mind, there were some necessary modifications made to certain parts of C++ which behaved differently than it did from its C beginnings. Some of these can be subtle, and newer programmers (who are likely to be more familiar with C++) may be confused by your code if you rely on special C-only behavior, it is simply best to avoid them.

Here are a few of these subtle gotcha's;

1. Using `sizeof()` on character literals: in C: `sizeof('a') == sizeof(int)`; however, in C++: `sizeof('a') == sizeof(char)`.
2. Do not rely on `enum` constants always being `signed int`'s.
3. Remember that a global variable declared as `const` in C++ is implicitly `static`, so you will not be able to link to it from a separate file. For this reason, it's best to declare all your `const`'s as `static const`'s and place in the headers. C++ won't care about the redundant `static`, and now your behavior will remain the same despite which language you are compiling under.

Be Wary of C99 Features which Conflict with C++

In a move that can only be considered reckless, the ANSI C99 committee introduced a couple of new features into the language which were completely incompatible with C++. Why these committee members chose to make your lives more difficult, I cannot say. However, it does mean that there are some unfortunate workarounds you may have to make, lest your C++ compiler come to a screeching halt:

1. The C99 `complex` keyword is incompatible with its C++ usage. In C99, `complex` is a suffix modifier for the `float` and `double` data types (analogous to the modifier prefixes `long` and `short` used for the `int` data type). To declare a double precision complex variable `z` in C, the language syntax looks like this:

```
double complex      z;
```

In C++, `complex` is a templated type, and thus the same declaration would appear as:

```
complex<double>     z;
```

Sadly, neither one of these declarations can compile in the other language. With no safe way to arbitrate, it is often best to avoid this keyword altogether when you can. However, since complex types can be extremely useful in mathematical operations, this simple workaround can be used:

```
#ifdef __cplusplus
#include <complex>
typedef complex<float>      single_complex;
typedef complex<double>    double_complex;
typedef complex<long double> quad_complex;
#else
#include <complex.h>
typedef float complex      single_complex;
typedef double complex     double_complex;
typedef long double complex quad_complex;
#endif // __cplusplus
```

At this point, you can then safely declare the variable this way:

```
double_complex      z;
```

2. Do not `#include <complex.h>` wherever `#include <iostream>` may be found. In addition to C99's new complex types are the complex versions of standard mathematical functions, each with a added `c-` prefix, such as `cpow()`, `cexp()`, `csin()`, etc. All of these functions are compatible with C++ with the exception of one: the complex logarithm function `clog()`. Its name conflicts with the C++ `iostream` object `clog`, used for logging data to the error stream. If this very common C++ header file `iostream` gets compiled into the same module containing `complex.h`, a compiler error will occur:

```
#include <complex.h>           // clog is a function  
#include <iostream>          // clog is an iostream object
```

Therefore, you will need to avoid the collision of these two header files.

C++: You will be Assimilated. Resistance is Futile.

The power of languages like C++, make continuance with C a less attractive proposition with each passing day. Many C programmers may even think they are still writing in C and not even realize that they have the C++ language interpreter turned on. Some of these programmers would be surprised to find that their code wouldn't compile if they even bothered to turn it off. And why should they? C++ gives them far more flexibility and power than C does. In the end, C++ is still a better C than C, so it's best to turn on the C++ flag as soon as you can.

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