Porting From Microsoft®
Visual C++® to CodeWarrior

A Metrowerks White Paper

By Rick Grehan

Porting From Microsoft® Visual C++® to CodeWarrior

This is a gathering of porting problems and solutions compiled by CodeWarrior users and engineers.

By Rick Grehan

AS GROWING POPULATIONS of different CPUs and different operating systems spread across the computing landscape, developers are increasingly presented with the chore of "porting" an application. And it's no small chore. Moving an application from one platform to another means dealing with differences in CPUs, operating systems, and -- the focus of this paper -- development tools.

Many developers use Microsoft's well-known Visual C++ compiler -- part of Microsoft's Visual Studio. Though its popularity is widespread and well-deserved, VC++ is nonetheless limited to use on Windows operating systems (Windows 95, 98, and NT). There is, for example, no version of VC++ for Macintosh systems, nor for Unix systems. A programmer who creates an application on Windows using VC++ and endeavors to move that application to, say, Unix is moving code not only to a different operating system, but to a different development toolset as well. Problems often arise that have nothing to do with the change in CPU or the change in OS.

Metrowerks CodeWarrior supports development for all the Windows OSes, as well as the Mac OS, Solaris, Linux, NetWare, the Palm OS; game development platforms such as PlayStation, Nintendo, and Sega; and realtime operating systems such as QNX, Nucleus Plus, and VxWorks. In addition, CodeWarrior compilers exist for x86 processors, Motorola 68K, MIPS, NEC V800, PowerPC, M-Core, Hitachi SH, Philips Trimedia, and Motorola DSP568xx. More processors and operating systems are on the way. Because Metrowerks engineers regularly cope with moving the CodeWarrior development environment to new CPUs and OSes, they are keenly aware of porting issues.

A Porting Toolchest

This white paper is a compendium of the more significant snares that CodeWarrior users, developers, and field-applications engineers have encountered as they have moved code from the VC++ environment to CodeWarrior.

Every application is unique. Porting problems that you tackle in one application may never surface in the next. It is therefore difficult to attach a "likelihood of occurrence" to any of the problems listed below. Nor can we ascribe to them a "severity level" to somehow gauge how far-reaching a problem might be. Something as (apparently) innocuous as the way the runtime system handles allocating zero-length memory can cause as much trouble as the runtime's handling of floating- point comparisons.

Consequently, the topics below are arranged in no particular order. This is a gathering of porting problems and solutions compiled by CodeWarrior users and engineers. We have, whenever possible, included the why's and the how-to's, so that if your next porting problem is one that someone else has already dealt with -- you won't have to.

Functions Unique to Microsoft

The Microsoft runtime library supports a number of functions that -- though useful -- are not part of the ANSI standard. Here are a couple of examples:

- `_strnicmp`. This function has the same prototype as the standard `strncmp()` function. It compares two strings --given pointers as well as a length count. The difference is that `_strnicmp()` is case-insensitive; `strncmp()` is case-sensitive.
- `_wtoi`. This function is similar to the ANSI-compatible `atoi()`, which converts an ASCII string to an integer (given a pointer to the string). The `_wtoi()` function, on the other hand, accepts as an input argument a pointer to a "wide" character string. (A "wide" character string is Microsoft's terminology for Unicode characters.)

A list of the non-ANSI functions supplied by Microsoft that CodeWarrior supports is shown in Table 1.

If your code uses any of the functions shown in Table 1, you can still use that code without having to write your own versions of the functions. For CodeWarrior compilers that 8target Windows systems, the functions are part of the MSL (Metrowerks Standard Library); you need only provide the header, `extras.h`. For CodeWarrior compilers that target non-Windows systems, you'll need to include the `extras.c` file in your project.

_access	_getdrive	_strdup
_alloca	_getdrives	_stricmp
_beginthreadex	_getHandle	_strlwr
_chdir	_get_osfhandle	_strnicmp
_chdrive	_heapmin	_strrev
_close	_isatty	_strupr
_creat	_itoa	_tell
_endthreadex	_itow	_ultoa
_fcntl	_lseek	_unlink
_fdopen	_makepath	_wcsdup
_fileno	_mkdir	_wcsicmp
_findclose	_msize	_wcslwr
_findfirst	_open	_wsnicmp
_findnext	_open_osfhandle	_wcsnset
_fstat	_read	_wcsrev
_fullpath	_rmdir	_wcsset
_gcvt	_searchenv	_wcsupr
_getcwd	_splitpath	_write
_getdcwd	_stat	_wstrrev
_getdiskfree	_strdate	_wtoi

TABLE 1. These functions are unique to VC++ (they are not part of the ANSI standard C runtime library). However, CodeWarrior does support these functions.

(Note: The MSL C reference manual, provided as part of the CodeWarrior development tools, lists all functions supported by MSL. In addition, associated with each function is a compatibility table that shows which operating systems and/or chips that function is compatible with.)

Microsoft's Inline Assembler

CodeWarrior supports inline assembly (the ability to include assembly language in C source code) that is nearly compatible with the assembly allowed in VC++. There are minor differences. For example, VC++ ignores letter case when resolving labels. In the following code, both `jmp` statements will branch to label `goThere`:

```
    ...
    jmp goThere
    ...
    jmp gothere
    ...
goThere:
    ...
```

In contrast, CodeWarrior sees labels through case-sensitive eyes. The above snippet of inline assembly code would cause the compiler to emit an error on the second `jmp` statement. The compiler would complain that no "gothere" label was defined. The fix is simply to change "gothere" to "goThere".

In addition, CodeWarrior does not currently support the following assembler directives recognized by VC++:

- EVEN - This directive pads the output into the object file so that the next variable defined is aligned to the next even address (word boundary). Though CodeWarrior does not support this directive, it does support ALIGN, which pads the output to a specified alignment. Consequently, you can use the directive ALIGN 2 in place of EVEN.
- LENGTH - This operator returns the number of elements in an array. LENGTH will return a 1 for non-array elements.
- TYPE - This operator returns the number of bytes in each data object of a particular variable. If the variable is an array, TYPE returns the size of one element of the array.

LENGTH and TYPE are often used to determine the number of bytes in arrayed data items. In fact, a data array's TYPE times its LENGTH is the total number of bytes consumed by the data array. The array's TYPE times its LENGTH will return the same value as will the operator SIZE on the array. CodeWarrior does support the SIZE operator.

(Note: Support for LENGTH and TYPE will be available in the Professional 5 release of CodeWarrior.)

Pointer Conversion

Many data objects in the Windows API are manipulated via handles, and the Windows header files define a variety of handle types. Some programmers of older Window programs were lax in their use of the handle datatypes. Such programmers would use one handle data type where a different one was called for. (Windows include files define a generic datatype called HANDLE. Many programmers would simply use the generic

HANDLE when a specific handle datatype should have been used.) Since the handle size is the same regardless of handle type, this caused little problem if the compiler simply ignored the misuse -- which VC++ did.

To manage this problem, a selection in one of the CodeWarrior settings panels allows you to relax the rules the compiler uses when converting from one pointer type to another. (This selection can be found in the "C/C++ Language" settings panel for the project. The checkbox is labeled "Relaxed Pointer Type Rules".) So, for example, if a programmer had created a handle to a window and defined that item as being of data type HWND, then later passed that handle as a data type HANDLE, without the relaxed rules selection, the compiler would emit an error. With the selection active, the compiler would allow a more lax use of handles.

This feature is ignored in C++ programs. C++ requires rigorous pointer use.

(Note: Turning off the relaxed rules is similar to enabling the STRICT symbol in VC++.)

RTTI

RTTI (runtime type identification) is a powerful feature in C++ that allows an application to determine an object's type at runtime. RTTI is available to the programmer through two operators: `typeid()` and `dynamic_cast<>`.

The `typeid()` operator takes an object or expression and returns a reference to an object of class `type_info`. The `type_info` class provides several methods that allow the program to deduce the argument object's type. For example, the `type_info` .name method will return a pointer to a string that contains the human-readable name of the object's type.

The `dynamic_cast<>` operator performs casting at runtime, rather than at compile time -- hence, its "dynamic" nature. Typically, a program will use this operator to cast a base-class pointer to a pointer of one of the base class's derived classes.

Now, here's the catch. Though both VC++ and CodeWarrior's support of RTTI is ANSI/ISO-compatible, the ANSI/ISO specification of RTTI is not particular about how the runtime object's type information is to be internally maintained by the runtime. Hence, the internal representation of an object's type is different in VC++ than in CodeWarrior. The result is that, if you have linked a library built by VC++ into your CodeWarrior application, and that library has created an object, attempting to use the `typeid()` or `dynamic_cast<>` operators on that object will fail. The CodeWarrior runtime will simply be confused by the internal structure of that object.

The solution to this is to obtain the source code for any object files or libraries that include objects on which your application calls RTTI functions. Either include the source code in your project, or build a library using the CodeWarrior compiler and add that library to your project.

Name-Mangling

Name-mangling is a technique by which a compiler encodes function names whereby the full prototype of the function is retained. This allows the compiler to distinguish between:

```
int getmin(int, int)
```

and

```
float getmin(float, float)
```

Both functions are called "`getmin()`", but one processes integers while the other processes floats.

The VC++ compiler has difficulty distinguishing between non-template functions and template functions that do not include the template type (the formal parameter) in the function's parameter list. The root of this problem has to do with the name-mangling techniques used. For example, suppose you have defined the following template and non-template functions in the same application:

```
template <> int foo<class T>(void)
{ ... }

int foo(void);
int foo()
{ ... }
```

If, later in your application you call the functions:

```
int i,j,k;
...
i=foo();
j=foo<int>();
k=foo<long>();
...
```

the compiler will be unable to distinguish among the various calls to `foo()`.

The CodeWarrior compiler will have a similar name-mangling problem with the above code, but only with ARM conformance on (one of the selections in the "C/C++ Language" settings panel).

(Note: ARM conformance, when active, assumes that the code will adhere to specifications spelled out in the Annotated C/C++ Reference Manual [Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, ISBN: 0201414591]. Having ARM conformance active enables the handling of scopes for variables declared in `for()` loops -- the variable's scope extends beyond the loop if declared in the `for()` expression itself. It also disables protected access base classes, disables variable declarations in `switch()` expression lists, and other syntactic nuances.)

There are two solutions. One is to turn off ARM conformance. Your source code may not require ARM specifics. The other solution is to rewrite any template function so that they include the formal parameter as one of the function arguments. (This latter solution is recommended by Microsoft.)

(Note: Turning off ARM conformance solves the preceding problem for the Professional 5 version of CodeWarrior. With previous versions, you'll have to rewrite the template function. A similar problem occurs if you define a template function as follows:

```
template <class T> int foo(void) { }
```

and a non-template function:

```
int foo(void) { ... }
```

Compiling the above under Professional 5 will produce an error message; the compiler will complain that the foo() function has been redefined. Earlier versions of CodeWarrior will not complain, but are unable to distinguish between these two functions.

In a similar vein, the CodeWarrior compiler can sometimes fail to properly mangle the name of a function that returns an enum. If this happens, the linker may fail to locate the function. The problem can be corrected by selecting the "Enums Always Int" checkbox from the project's C/C++ Language settings panel.

Floating-Point Differences

The IEEE floating-point specifications (IEEE 754) define a number of special values, one of which is referred to as NaN (not a number). A NaN is represented by a specific bitpattern designed so that floating-point runtime routines (or a FPU, if one is available) will not mistake the bitpattern for a valid floating-point number.

Certain floating-point functions will return a NaN to indicate that the program attempted an invalid operation. Examples include attempting to divide zero by itself and attempting to take the square root of a negative number. (You can refer to a number of sources that catalog the specific operations that yield an invalid result. One is Intel's 486 Microprocessor Family Programmer's Reference Manual. Another is Gary Kacmarcik's "Optimizing PowerPC Code" from Addison- Wesley. Another good source is Apple's PowerPC Numerics manual, available for download from:

http://developer.apple.com/techpubs/macos8/Utilities/PowerPCNumerics/

powerpcnumerics.html)

The problem in porting from VC++ to CodeWarrior arises from two characteristics of NaNs, and how they are treated.

First, NaNs have the property of "percolating" through floating-point operations. That is, if you have a complex floating-point expression, and a NaN is generated somewhere "inside" the expression, the NaN will pass through the entire expression so that the result is a NaN.

Second, NaNs are unordered. They are considered by the floating-point library to be neither bigger, smaller, nor equal to ANY other floating- point values.

The IEEE floating-point specification dictates that the result of floating-point comparisons should be false whenever a NaN is involved, with the exception of the != (not equals) comparison, which always returns true. Microsoft's C++ compilers do not adhere to this, with the result being that the results of comparisons involving NaNs can vary depending on the structure of the expression involved. CodeWarrior does adhere to the IEEE specification.

It is possible, then, that a comparison involving a NaN as one of the operands will behave differently when compiled under VC++ than when compiled with CodeWarrior. If you suspect that your program might have this problem, the routine in Listing 1 will enable the floating-point exception for invalid operations. Include this routine in your program, and call it before executing any floating-point operations.

If a NaN *is* at the heart of the problem, it will throw an exception when the program attempts to use it as part of a comparison.

```
#include <fenv.h>
void enable_nan_exception(void)
{
    fenv_t fe;
    // This should turn on exceptions
    //  on NaN generation
    fgetenv(&fe);
    fe=fe & ~FE_INVALID;
    fsetenv(&fe);
}
```

Listing 1. This routine unmasks the "invalid floating-point exception" bit in the floating-point control word. If a NaN is used as a factor in a subsequent floating-point expression, an exception is thrown. What catches that exception varies from system to system. Usually, a handler in the OS catches it. You can, however, write your own exception handler.

Thread Functions

Many books describing Win32's thread support -- and many Microsoft documents -- warn the programmer that the preferred function for creating and starting a thread is the _beginthread() function. (Consequently, terminating the thread should be done using the _endthread() function.) A similar function is _beginthreadex() -- the "ex" indicates that this is an "extended" call. The _beginthreadex() function provides all the capabilities of _beginthread() plus three additional arguments. (The details can be found in any Win32 documentation.)

The _beginthread()/_beginthreadex() calls were preferred over the more spartan CreateThread() function because the latter performed no initialization needed for the runtime library. For example, the random number generation routines in the standard runtime library require a "seed" value to be stored somewhere. This seed value is used to generate the next random number in the sequence. Such storage must be made available on a per-thread basis. CreateThread() doesn't allocate that storage -- _beginthread() and _beginthreadex() do.

However, beginthread() is a deprecated call. The reason lies not within beginthread(), but within the required, accompanying endthread(). Because endthread() closes the file handle, it's possible that a thread could be created, execute, and terminated, all before the actual call to beginthread() returned from the *creating* thread. The result is that the

creating thread would be left with a handle that referenced a non-existend thread, with no indication that the thread had run and terminated.

Though VC++ continues to support `_beginthread()/_endthread()`, even Microsoft suggests programmers move to `_beginthreadex()/endthreadex()`. Thankfully, since the two functions are so similar to one another, this is not an overwhelming issue. In most cases, the arguments in `_beginthreadex()` that have no counterpart in `_beginthread()` can simply be set to an appropriately-typed NULL value. And, of course, you'll need to replace `_endthread()` with `_endthreadex()`.

Finally, both `_beginthread()` and `_beginthreadex()` create a handle to the created thread. The `_endthread()` call automatically closes the handle, so an explicit call to `CloseHandle()` was unnecessary. However, `_endthreadex()` does not automatically close the thread handle. You'll need to add `CloseHandle()` call to your source code in the appropriate place.

Versions of CodeWarrior before Professional 5 support only `_beginthreadex/_endthreadex`. The Professional 5 release of CodeWarrior will contain an implementation of `beginthread()/endthread()` -- with all the accompanying warts. The implementation is provided for portability's sake. Nevertheless, use with caution.

C9X support

C9X is a proposed update to the ANSI standard for the C programming language. The standard is still in deliberation, though the original C9X charter had set a goal of being completed by the year 2000. Time will tell.

However, portions of the standard have been accepted by many compiler companies. Developers are even now using elements of the C9X standard.

The functions and macros already accepted fall primarily in the area of floating-point routines. CodeWarrior already supports many of these C9X-proposed functions and macros. Microsoft has an equivalent subset, though their names are slightly different than the names suggested in C9X (and their interfaces may differ). Table 2 shows a list of functions and macros already supported by Metrowerks and (where they are available) their Microsoft equivalents.

C9X FUNCTION SUPPORTED BY CODEWARRIOR	*MICROSOFT EQUIVALENT*
acosh	---
asinh	---
atanh	---
copysign()	_copysign()
exp2()	---
expm1()	---
fdim()	---
fmax()	__max()
fmin()	__min()
fpclassify()	_fpclass()
hypot()	_hypot()
isfinite()	_finite()
isnan()	_isnan()
log1p()	---
log2()	---
logb()	_logb()
nan()	---
nearbyint()	---
nextafter()	_nextafter()
remainder()	---
remquo()	---
rint()	---
rinttol()	---
round()	---
roundtol()	---
scalb()	_scalb
trunc()	---

Table 2. C9X Math macros and functions supported by CodeWarrior, and their Microsoft equivalents.

Min and max

The Microsoft `windef.h` header files defines a pair of macros, `min()` and `max()`. They are simple macros that resolve to the conditional ("? :") operator. However, the standard template library (STL), now part of the C++ standard, defines a pair of

algorithms, also called `min()` and `max()`. As you might guess, there's a collision waiting to happen.

Fortunately, Microsoft has provided a means of disabling the `min()` and `max()` macros in the windef.h header file. Those macros are bracketed in an `#ifdef` statement. So, if your application uses the `min()` and `max()` template algorithms, you can disable Microsoft's `min()` and `max()` by adding the line

```
#define NOMINMAX
```

just prior to the `#include <windows.h>` line in your source code. (The `windows.h` file is a standard include in all Windows applications. The `#include` to `windef.h` is nested within `windows.h`.)

Multiple Definitions in MFC and MSL

There are multiple definitions between code in the Metrowerks runtime (MSL) and the MFC libraries. The MFC libraries override the MSL. (Most often, developers are "bitten" by multiple definitions for the "new" and "delete" operators.) This can cause multiple definition errors when an application is linking.

The solution is to change the link order in the CodeWarrior project so that the Metrowerks library files appear after the MFC library file. You can do this using the "link order" view within a CodeWarrior project. The link order view shows a list of a project's files, arranged in the order they will be passed to the linker. You can simply drag and drop files to different locations in the list to modify the order.

(Note: The CodeWarrior files are MWCRTL.DLL or MWCRTL.Lib for the release version of a target, and MWCRTLD.DLL or MWCRTLD.Lib for the debug version.)

Allocating Nothing

The ANSI specification for the behavior of `malloc(0)` says that the call can either return a pointer to a very small chunk of memory, or it can return a NULL. VC++ returns the former, CodeWarrior returns the latter. Both results are valid, and both work with any subsequent calls to `free()` or `realloc()`.

Note that calls to `malloc()` (or `calloc()` or `realloc()`) with data items of zero length are discouraged by ANSI specifications; precisely because the result is implementation-dependent.

However, the discrepancy between VC++ and CodeWarrior may cause problems in some programs, if the programs examine the returned value. For example, a program might misinterpret the NULL result of the call to `malloc()` as an indication of insufficient memory (which the NULL result otherwise indicates).

These differences in behavior between VC++ and CodeWarrior will be corrected in the Professional 5 release of CodeWarrior with the addition of a define statement. Add the following to the start of a source file:

```
#define _MSL_MALLOC_0_RETURNS_NON_NULL
```

and any calls to malloc() in that file will behave as does VC++. Of course, the real solution to this problem is not to make any calls to allocate (or reallocate) memory of zero size (as suggested by the ANSI standard).

Moving A Project to CodeWarrior

VC++ project files are not compatible with CodeWarrior files. The Professional 5 version of CodeWarrior, however, will introduce a new Makefile Converter Wizard. This wizard will significantly reduce the time it takes to get a VC++ project into CodeWarrior. As the name implies, the wizard "imports" a makefile; creating a corresponding CodeWarrior project.

A developer can move a project from VC++ to CodeWarrior by:

1) Exporting the makefile from VC++. (This creates an nmake-compatible makefile from the VC++ project.)
2) Making a backup of the makefile and source files. (If some problem occurs during conversion, the developer can always restore the original project from the backup.)
3) Running CodeWarrior's Makefile Importer Wizard.

Conclusion

Software is an organic, growing thing. While we sleep, engineers at Microsoft and Metrowerks are at work modifying their respective development tools; improving the compilers, adding new features, making them faster and (we hope) smarter. This nonstop growth is a two-edged sword. It means better tools for us all, but it also means more opportunities for those sorts of discrepancies that we've described above.

If we've left something out in the preceding discussion, we encourage you to contact Metrowerks so we can share your problem -- and its solution -- with others. One popular Metrowerks motto is: "We listen, we act." Please visit the support section of the Metrowerks website at www.metrowerks.com/support.

The author would like to thank the many Metrowerks engineers who contributed to this white paper. It would not have been possible without their experiences and insights.