What does the /ALTERNATENAME linker switch do?

devblogs.microsoft.com/oldnewthing/20200731-00

July 31, 2020

Raymond Chen

There's an undocumented switch for the Microsoft Visual Studio linker known as `/ALTERNATENAME`. Despite being undocumented, people use it a lot. So what is it?

This is effectively a command line switch version of the `OLDNAMES.LIB` library. When you say `/ALTERNATENAME:X=Y`, then this tells the linker that if it is looking for a symbol named `X` and can't find it, then before giving up, it should redirect it to the symbol `Y` and try again.

The C runtime library uses this mechanism for various sneaky purposes. For example, there's a part that goes

```
BOOL (WINAPI * const _pDefaultRawDllMain)(HANDLE, DWORD, LPVOID) = NULL;
#if defined (_M_IX86)
#pragma comment(linker, "/alternatename:__pRawDllMain=__pDefaultRawDllMain")
#elif defined (_M_IA64) || defined (_M_AMD64)
#pragma comment(linker, "/alternatename:_pRawDllMain=_pDefaultRawDllMain")
#else  /* defined (_M_IA64) || defined (_M_AMD64) */
#error Unsupported platform
#endif  /* defined (_M_IA64) || defined (_M_AMD64) */
```

What this does is say, "If you need a symbol called `_pRawDllMain`, but you can't find it, then try again with `_pDefaultRawDllMain`." If an object file defines `_pRawDllMain`, then that definition will be used. Otherwise `_pDefaultRawDllMain` will be used.

Note that `/ALTERNATENAME` is a linker feature and consequently operates on decorated names, since the linker doesn't understand compiler-specific name-decoration algorithms. This means that you typically have to use different versions of the `/ALTERNATENAME` switch, depending on what architecture you are targeting. In the above example, the C runtime library knows that `__cdecl` decoration prepends an underscore on x86, but not on any other platform.

This use of `/ALTERNATENAME` here is a way for the compiler to generate hooks into the DLL startup process based on the code being compiled. If there is no `_pRawDllMain` defined by an object file, then `_pDefaultRawDllMain` will be used instead, and that version is just a null pointer, which means, "Don't do anything special."

This pattern of using the `/ALTERNATENAME` switch lets you provide a default value for a function or variable, which others can override if they choose. For example, you might do something like this:

```
void default_error_log() { /* do nothing */ }
// For expository simplification: assume x86 cdecl
#pragma comment(linker, "/alternatename:_error_log=_default_error_log")
```

If nobody defines a custom `error_log` function, then all references to `error_log` are redirected to `default_error_log`, and the default error log function does nothing.[1]

The C++/WinRT library uses `/ALTERNATENAME` for a different purpose. The C++/WinRT library wants to support being used both with and without `windows.h`, so it contains its own declarations for the Windows functions and structures that it needs.

But now there's a problem: If it is used *with* `windows.h`, then there are structure definition errors. Therefore, C++/WinRT needs to give its equivalent declarations of Windows structures some other name, to avoid redefinition errors.

But this in turn means that the function prototypes in the C++/WinRT library need to use the renamed structures, rather than the original Windows structures, in case the C++/WinRT library is used *without* `windows.h`. This declaration will in turn create a conflict if the C++/WinRT library is used *with* `windows.h` when the real declarations are encountered in `windows.h`.

The solution is to rename the C++/WinRT version of Windows functions, too. C++/WinRT gives them a `WINRT_IMPL_` prefix, so that there is no function declaration collision.

We now have two parallel universes. There's the `windows.h` universe, and the C++/WinRT universe, each with their own structures and functions. The two parallel universes are unified by the `/ALTERNATENAME` directive, which tells the linker, "If you find yourself looking for the function `WINRT_IMPL_GetLastError`, try again with `GetLastError`." Since nobody defines `WINRT_IMPL_GetLastError`, the "try again" kicks in, and all of the calls to `WINRT_GetLastError` end up redirected to the operating system `GetLastError` function, which is what we wanted in the first place.

[1] The more traditional way of doing this (that doesn't rely on undocumented vendor-specific linker features) is to take advantage of the classical model for linking, specifically the part where you can let an OBJ override a LIB: What you do is define `_pRawDllMain` in a separate OBJ file that defines nothing except that one variable, and put that OBJ in the C runtime LIB. If the module provides its own definition of `_pRawDllMain` in an OBJ file, then that definition is used. Otherwise, the linker will search through the LIBs, and eventually it will find the one in the C runtime LIB and use that one.

So why does `/ALTERNATENAME` exist if you could already get this effect via LIBs, and in way that all linkers support, not just the Microsoft C linker?

C++/WinRT is a header-only library. It has no LIB in which to put these default definitions. It therefore has to use the “command line switch version of a LIB”.

Raymond Chen

Follow