How does the linker decide whether to call WinMain or wWinMain?

devblogs.microsoft.com/oldnewthing/20241004-00

October 4, 2024

If you don't specify the `/ENTRY` option to the Visual C++ linker, it tries to guess. The details are spelled out in the documentation, but I'll recapture it here since it will lead to a troubleshooting session.

First, if you specified the `/DLL` flag, then the default entry point is `_DllMainCRTStartup`. That's the easy case.

If you did not specify the `/DLL` flag, then the linker looks at your `/SUBSYSTEM` flag. If you asked for `/SUBSYSTEM:CONSOLE`, then it looks for `wmain` and `main`. If you asked for `/SUBSYSTEM:WINDOWS`, then it looks for `wWinMain` and `WinMain`. If you didn't specify a subsystem, then it looks for all four symbols, and whichever symbol it finds first determines what the implied `/ENTRY` is:

If linker finds	Then entry point is
`wmain`	`wmainCRTStartup`
`main`	`mainCRTStartup`
`wWinMain`	`wWinMainCRTStartup`
`WinMain`	`WinMainCRTStartup`

The search for these magic symbols is made over the symbols present in the module after external references have been resolved (according to the classical model for linking).[1] If there is no match for anything, then it uses the ANSI entry point and hopes for the best.

Okay, now you can try to solve this customer's problem:

> We have several programs that follow the same overall structure. The idea is that we will put the `wWinMain` function in a common static library, and the `wWinMain` function will use global variables (defined differently in each program) to manage the program-specific business logic. But when we do that, we get a linker error saying "unresolved external symbol WinMain referenced by invoke_main." It works if we use `WinMain`, but we want to support Unicode.

Moving the `wWinMain` function into a library means that the linker doesn't see it when looking through your module's symbols trying to find an entry point. It therefore assumes ANSI and then gets stuck when it can't find `WinMain`.

One way to solve this is to require clients of the library to define their own `wWinMain`, but have it just forward the call to your library.

```
int WINAPI wWinMain(HINSTANCE hinst,
    HINSTANCE hinstPrev,
    PWSTR pszCmdLine,
    int nCmdShow)
{
    return Contoso::wWinMain(hinst, hinstPrev,
            pszcmdLine, nCmdShow);
}
```

Now that the `wWinMain` function is defined in the project object files, the linker will see it and use it as the entry point.

Having a separate function into which clients forward their `wWinMain` also means that it's possible for a program to use multiple libraries.

```
int WINAPI wWinMain(HINSTANCE hinst,
    HINSTANCE hinstPrev,
    PWSTR pszCmdLine,
    int nCmdShow)
{
    if (GetConfiguration("useContoso")) {
        return Contoso::wWinMain(hinst, hinstPrev,
                pszcmdLine, nCmdShow);
    } else {
        return Fabrikam::wWinMain(hinst, hinstPrev,
                pszcmdLine, nCmdShow);
    }
}
```

If you don't really care about peaceful coexistence with other libraries, another solution would be to have a function that everybody must call, but which doesn't do anything,[2] and put that function in the same .obj file as your `wWinMain`. The classical model of linking will try to resolve that function, and it will pull in the .obj file from the library, and that .obj file carries `wWinMain` along for the ride.

Another possibility is to just put `WinMain` in your library, since that's the name that the linker will use for a `/SUBSYSTEM:WINDOWS` module if it can't find `wWinMain`. Your `WinMain` can ignore its ANSI command line and call `GetCommandLineW()` to get the Unicode command line.

But probably the easiest solution if you don't care about peaceful coexistence with other libraries is to add a

```
#pragma comment(linker, "/include:wWinMain")
```

This forces the linker to resolve `wWinMain`, and it will find it in your library and add its object file to your project, at which point the "Gosh, what entry point should I use?" will see that `wWinMain` is present and use it.

[1] Naturally, this is done prior to dead code removal: Without an entry point, all you have is dead code!

[2] Basically, you're using the InitCommonControls trick.