Why do I get errors or warnings about some weird symbol called ?main@@YAHP$01E$AAV?$Array@PE$AAVString@Platform…, part 3

devblogs.microsoft.com/oldnewthing/20250627-00/?p=111316

June 27, 2025

We have been investigating why a project is getting an error about a weird C++/CX symbol, and we tracked it down to three things:

- If you compile with C++/CX, the compiler injects `vccorlib.lib` as a default library.
- The `vccorlib.lib` library provides a definition of `main`.
- The linker special rule for resolving references introduced by a library causes the search for `main` to look in `vccorlib.lib` ahead of the fuzzer library that contains the `main` we want.

To get the linker to find the intended `main`, we need to take away one of the conditions.

For the first item, we could take away all the components that use C++/CX. But presumably they are there because we need to test them, so that's not an option.

Another possibility is to remove `vccorlib.lib` from the default library list. The library is still needed, but we can add it back as an explicit library.

```
link /out:fuzzer.exe /subsystem:console fuzzer.obj cx.obj lib.lib vccorlib.lib
/NODEFAULTLIB:vccorlib.lib
```

The avoids the problem with the special rule: The reference to `main` came from `libcmt.lib`, so the search proceeds through the rest of the default libs, and then wraps around back to the explicit libraries. In the list of explicit libraries, we have been careful to put `lib.lib` ahead of `vccorlib.lib`, so that the `main` in `lib.lib` gets found first.

For the second item, there's not much we can do because the `vccorlib.lib` is provided as part of the toolchain, so we are not at liberty to modify it.

For the third item, we can try to avoid the linker special rule by making sure that the reference to `main` does not come from a library in the first place. That ensures that the search starts with the first explicitly library rather than doing the weird wraparound thing.

One way to force it is to have another object file that contains an explicit reference to `main`

```
rem new! An object file that requests main.
>forcemain.cpp echo int __cdecl main(int, char**); auto forcemain = main;
cl /c forcemain.cpp

rem Add it as the first object file.
link /out:fuzzer.exe /subsystem:console forcemain.obj fuzzer.obj cx.obj lib.lib

rem success!
```

The first reference to `main` comes from `forcemain`, which is not a library, so the special library search rule does not come into play.

I put `forcemain.obj` first to increase the likelihood that it will provide the first reference to `main`. If it came second, then maybe resolving a symbol from the first object file leads to a reference that is resolved by a library, and that in turn requests a reference to `main`, and now the special library search rule kicks in.

It may be difficult to ensure that `forcemain.obj` comes first. For example, some tooling might sort the object files alphabetically, or somebody might just decide to sort them alphabetically as part of just making things more tidy,[1] causing `forcemain.obj` to lose its special place at the front of the object list.

Therefore, I like to use the `/INCLUDE` trick.

```
link /out:fuzzer.exe /subsystem:console fuzzer.obj cx.obj lib.lib /INCLUDE:main

rem success!
```

The compiler team tells me that references injected via `/INCLUDE` get ushered to the front of the line, so they get resolved before any references that come from the object files themselves. In this case, it means that `/INCLUDE:main` ensures that `main` is resolved before any symbols from object files, thereby removing the dependency on the order of object files.

My colleague Martyn Lovell noted that even though you can cobble together something that works, he considers it generally a mistake to put your entry point in a library. Libraries generally carry the meaning of “Use this only if necessary,” but in the case of the fuzzing library, their specific `main` function is mandatory, not a fallback. This is a problem I discussed earlier in the context of choosing between `WinMain` and `wWinMain`.

The entry point should be in an explicit object file that is added to the project, or (my preferred option) the library should provide its main function under a name like `fuzzer_main` which programs are expected to forward to.

```
// fuzzer.cpp
#include <fuzzerlibrary.h>
int __cdecl main(int argc, char** argv)
{
    return fuzzer_main(argc, argv);
}

bool fuzzer_callback(void* data, int length)
{
    ⟦ ... ⟧
}
```

This also allows you to do things like choose between two fuzzers at runtime, or run multiple fuzzers from a single binary or run the same fuzzer twice.

```
// fuzzer.cpp
#include <fuzzerlibrary1.h>
#include <fuzzerlibrary2.h>
int __cdecl main(int argc, char** argv)
{
    // If run with no arguments, then provide
    // some defaults.
    if (argc == 1) {
        static char arg1[] = "default-argument1";
        static char arg2[] = "default-argument2";
        static char* args[] = { argv[0], arg1, arg2 };
        argc = 3;
        argv = args;
    }

    // Run it through both fuzzers
    int result = fuzzer1_main(argc, argv);
    if (result == 0) {
        result = fuzzer2_main(argc, argv);
    }
}
    return result;
}

bool fuzzer_callback(void* data, int length)
{
    ⟦ ... ⟧
}
```

Now, for convenience, the fuzzer library could also provide the `main` function that we put into `fuzzer.cpp`. But even so, there should be a separate name (like `fuzzer_main`) that can be used to invoke it explicitly.

[1] For example, keeping lists in alphabetical or numeric order reduces the likelihood of bad merges.