# Combining C++ and C - how does #ifdef __cplusplus work?



I'm working on a project that has a lot of legacy **C** code. We've started writing in C++, with the intent to eventually convert the legacy code, as well. I'm a little confused about how the **C** and C++ interact. I understand that by wrapping the **C** code with `extern "C"` the C++ compiler will not mangle the **C** code's names, but I'm not entirely sure how to implement this.

So, at the top of each **C** header file (after the include guards), we have

```
#ifdef __cplusplus
extern "C" {
#endif
```

and at the bottom, we write

```
#ifdef __cplusplus
}
#endif
```

In between the two, we have all of our includes, typedefs, and function prototypes. I have a few questions, to see if I'm understanding this correctly:

1. If I have a C++ file A.hh which includes a **C** header file B.h, includes another **C** header file C.h, how does this work? I think that when the compiler steps into B.h, `__cplusplus` will be defined, so it will wrap the code with `extern "C"` (and `__cplusplus` will not be defined inside this block). So, when it steps into C.h, `__cplusplus` will not be defined and the code will not be wrapped in `extern "C"` . Is this correct?
2. Is there anything wrong with wrapping a piece of code with `extern "C" { extern "C" { .. } }` ? What will the second `extern "C"` do?
3. We don't put this wrapper around the .c files, just the .h files. So, what happens if a function doesn't have a prototype? Does the compiler think that it's a C++ function?
4. We are also using some third-party code which is written in **C**, and does not have this sort of wrapper around it. Any time I include a header from that library, I've been putting an `extern "C"` around the #include. Is this the right way to deal with that?
5. Finally, is this set up a good idea? Is there anything else we should do? We're going to be mixing **C** and C++ for the foreseeable future, and I want to make sure we're covering all our bases.

c++ c c-preprocessor

edited Feb 29 at 19:34

Brian Tompsett - 汤莱恩
**2,774** 10 21 58

asked Sep 24 '10 at 17:03

dublev
**1,042** 2 9 12

4 Have a look at C++ FQA: mixing C and C++ and C++ FAQ: mixing C and C++. – Lazer Sep 24 '10 at 19:25

## 3 Answers

`extern "C"` doesn't really change the way that the compiler reads the code. If your code is in a .c file, it will be compiled as C, if it is in a .cpp file, it will be compiled as C++ (unless you do something strange to your configuration).

What `extern "C"` does is affect linkage. C++ functions, when compiled, have their names mangled -- this is what makes overloading possible. The function name gets modified based on the types and number of parameters, so that two functions with the same name will have different symbol names.

Code inside an `extern "C"` is still C++ code. There are limitations on what you can do in an extern "C" block, but they're all about linkage. You can't define any new symbols that can't be built with C linkage. That means no classes or templates, for example.

`extern "C"` blocks nest nicely. There's also `extern "C++"` if you find yourself hopelessly

trapped inside of `extern "C"` regions, but it isn't such a good idea from a cleanliness perspective.

Now, specifically regarding your numbered questions:

Regarding #1: __cplusplus should be defined inside of `extern "C"` blocks. This doesn't matter, though, since the blocks should nest neatly.

Regarding #2: __cplusplus will be defined for any compilation unit that is being run through the C++ compiler. Generally, that means .cpp files and any files being included by that .cpp file. The same .h (or .hh or .hpp or what-have-you) could be interpreted as C or C++ at different times, if different compilation units include them. If you want the prototypes in the .h file to refer to C symbol names, then they must have `extern "C"` when being interpreted as C++, and they should not have `extern "C"` when being interpreted as C -- hence the `#ifdef __cplusplus` checking.

To answer your question #3: functions without prototypes will have C++ linkage if they are in .cpp files and not inside of an `extern "C"` block. This is fine, though, because if it has no prototype, it can only be called by other functions in the same file, and then you don't generally care what the linkage looks like, because you aren't planning on having that function be called by anything outside the same compilation unit anyway.

For #4, you've got it exactly. If you are including a header for code that has C linkage (such as code that was compiled by a C compiler), then you must `extern "C"` the header -- that way you will be able to link with the library. (Otherwise, your linker would be looking for functions with names like `_Z1hic` when you were looking for `void h(int, char)`

5: This sort of mixing is a common reason to use `extern "C"` , and I don't see anything wrong with doing it this way -- just make sure you understand what you are doing.

answered Sep 24 '10 at 17:32

Andrew Shelansky
**2,049** 2 11 16



1. `extern "C"` doesn't change the presence or absence of the `__cplusplus` macro. It just changes the linkage and name-mangling of the wrapped declarations.
2. You can nest `extern "C"` blocks quite happily.
3. If you compile your `.c` files as C++ then anything not in an `extern "C"` block, and without an `extern "C"` prototype will be treated as a C++ function. If you compile them as C then of course everything will be a C function.
4. Yes
5. You can safely mix C and C++ in this way.

edited Nov 27 '15 at 11:02

Martin
**3,767** 5 21 43

answered Sep 24 '10 at 17:08

Anthony Williams
**38.3k** 5 83 124

A couple of gotchas that are colloraries to Andrew Shelansky's excellent answer and to disagree a little with *doesn't really change the way that the compiler reads the code*

Because your function prototypes are compiled as C, you can't have overloading of the same function names with different parameters - that's one of the key features of the name mangling of the compiler. It is described as a linkage issue but that is not quite true - you will get errors from both the compiler and the linker.

The compiler errors will be if you try to use C++ features of prototype declaration such as overloading.

The linker errors will occur later because your function will appear to not be found, if you do **not** have the *extern "C"* wrapper around declarations and the header is included in a mixture of C and C++ source.

One reason to discourage people from using the *compile C as C++* setting is because this means their source code is no longer portable. That setting is a project setting and so if a .c file is dropped into another project, it will not be compiled as c++. I would rather people take the time to rename file suffixes to .cpp.

answered Aug 26 '14 at 8:15

Andy Dent
**11.8k** 2 54 79

1 This was the cryptic cause, pulling my hair out. Really needs to be posted somewhere. – Mitchell Currie Jul

16 '15 at 23:13