How to mix C and C++

FEATURES

Current ISO C++ status

Upcoming ISO C++ meetings

Upcoming C++ conferences

Compiler conformance status

NAVIGATION

FAQ Home

FAQ RSS Feed

FAQ Help

SEARCH THIS WIKI

Go

GO TO PAGE

Go

Contents of this section:

FAQ What do I need to know when mixing C and C++ code?

Here are some high points (though some compiler-vendors might not require all these; check with your compiler-vendor's documentation):

- You must use your C++ compiler when compiling `main()` (e.g., for static initialization)
- Your C++ compiler should direct the linking process (e.g., so it can get its special libraries)
- Your C and C++ compilers probably need to come from the same vendor and have compatible versions (e.g., so they have the same calling conventions)

In addition, you'll need to read the rest of this section to find out how to make your C functions callable by C++ and/or your C++ functions callable by C.

BTW there is another way to handle this whole thing: compile all your code (even your C-style code) using a C++ compiler. That pretty much eliminates the need to mix C and C++, plus it will cause you to be more careful (and possibly —hopefully!— discover some bugs) in your C-style code. The down-side is that you'll need to update your C-style code in certain ways, basically because the C++ compiler is more careful/picky than your C compiler. The point is that the effort required to clean up your C-style code may be less than the effort required to mix C and C++, and as a bonus you get cleaned up C-style code. Obviously you don't have much of a choice if you're not able to alter your C-style code (e.g., if it's from a third-party).

FAQ How do I call a C function from C++?

Just declare the C function `extern "C"` (in your C++ code) and call it (from your C or C++ code). For example:

```
// C++ code

extern "C" void f(int); // one way

extern "C" {    // another way
    int g(double);
    double h();
};

void code(int i, double d)
{
    f(i);
    int ii = g(d);
    double dd = h();
    // ...
}
```

The definitions of the functions may look like this:

```
/* C code: */

void f(int i)
{
    /* ... */
}

int g(double d)
{
    /* ... */
}

double h()
{
    /* ... */
}
```

Note that C++ type rules, not C rules, are used. So you can't call function declared `extern "C"` with the wrong number of arguments. For example:

```
// C++ code

void more_code(int i, double d)
{
```

```
    double dd = h(i,d); // error: unexpected arguments
    // ...
}
```

FAQ How do I call a C++ function from C?

Just declare the C++ function `extern "C"` (in your C++ code) and call it (from your C or C++ code). For example:

```
// C++ code:

extern "C" void f(int);

void f(int i)
{
    // ...
}
```

Now `f()` can be used like this:

```
/* C code: */

void f(int);

void cc(int i)
{
    f(i);
    /* ... */
}
```

Naturally, this works only for non-member functions. If you want to call member functions (incl. virtual functions) from C, you need to provide a simple wrapper. For example:

```
// C++ code:

class C {
    // ...
    virtual double f(int);
};

extern "C" double call_C_f(C* p, int i) // wrapper function
{
    return p->f(i);
}
```

Now `C::f()` can be used like this:

```
/* C code: */

double call_C_f(struct C* p, int i);

void ccc(struct C* p, int i)
{
    double d = call_C_f(p,i);
    /* ... */
}
```

If you want to call overloaded functions from C, you must provide wrappers with distinct names for the C code to use. For example:

```
// C++ code:

void f(int);
void f(double);
```

```
extern "C" void f_i(int i) { f(i); }
extern "C" void f_d(double d) { f(d); }
```

Now the `f()` functions can be used like this:

```
/* C code: */

void f_i(int);
void f_d(double);

void cccc(int i,double d)
{
    f_i(i);
    f_d(d);
    /* ... */
}
```

Note that these techniques can be used to call a C++ library from C code even if you cannot (or do not want to) modify the C++ headers.

FAQ How can I include a standard C header file in my C++ code?

To `#include` a standard header file (such as `<cstdio>`), you don't have to do anything unusual. E.g.,

```
// This is C++ code

#include <cstdio>                // Nothing unusual in #include line

int main()
{
  std::printf("Hello world\n");  // Nothing unusual in the call either
  // ...
}
```

The `std::` part of the `std::printf()` call may look unusual if you're coming from C, but this is the correct way to write it in C++.

If you are compiling C code using your C++ compiler, you don't want to have to tweak all these calls from `printf()` to `std::printf()`. Fortunately in this case the C code will use the old-style header `<stdio.h>` rather than the new-style header `<cstdio>`, and the magic of namespaces will take care of everything else:

```
/* This is C code that I'm compiling using a C++ compiler */

#include <stdio.h>          /* Nothing unusual in #include line */

int main()
{
  printf("Hello world\n");  /* Nothing unusual in the call either */
  // ...
}
```

Final comment: if you have C headers that are *not* part of the standard library, we have somewhat different guidelines for you. There are two cases: either you *can't* change the header, or you *can* change the header.

FAQ How can I include a non-system C header file in my C++ code?

If you are including a C header file that isn't provided by the system, you may need to wrap the `#include` line in an `extern "C" { /*...*/ }` construct. This tells the C++ compiler that the functions declared in the header file are C functions.

```
// This is C++ code

extern "C" {
  // Get declaration for f(int i, char c, float x)
  #include "my-C-code.h"
}

int main()
{
  f(7, 'x', 3.14);   // Note: nothing unusual in the call
  // ...
}
```

Note: Somewhat different guidelines apply for C headers provided by the system (such as `<cstdio>`) and for C headers that you can change.

FAQ How can I modify my own C header files so it's easier to `#include` them in C++ code?

If you are including a C header file that isn't provided by the system, and if you are able to change the C header, you should strongly consider adding the `extern "C" {...}` logic inside the header to make it easier for C++ users to `#include` it into their C++ code. Since a C compiler won't understand the `extern "C"` construct, you must wrap the `extern "C" {` and `}` lines in an `#ifdef` so they won't be seen by normal C compilers.

Step #1: Put the following lines at the very top of your C header file (note: the symbol `__cplusplus` is `#define`d if/only-if the compiler is a C++ compiler):

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

Step #2: Put the following lines at the very bottom of your C header file:

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

Now you can `#include` your C header without any `extern "C"` nonsense in your C++ code:

```
// This is C++ code

// Get declaration for f(int i, char c, float x)
#include "my-C-code.h"   // Note: nothing unusual in #include line

int main()
{
```

```
  f(7, 'x', 3.14);       // Note: nothing unusual in the call
  // ...
}
```

Note: Somewhat different guidelines apply for C headers provided by the system (such as `<cstdio>`) and for C headers that you can't change.

Note: `#define` macros are evil in 4 different ways: evil#1, evil#2, evil#3, and evil#4. But they're still useful sometimes. Just wash your hands after using them.

FAQ How can I call a non-system C function `f(int,char,float)` from my C++ code?

If you have an individual C function that you want to call, and for some reason you don't have or don't want to `#include` a C header file in which that function is declared, you can declare the individual C function in your C++ code using the `extern "C"` syntax. Naturally you need to use the full function prototype:

```
extern "C" void f(int i, char c, float x);
```

A block of several C functions can be grouped via braces:

```
extern "C" {
  void   f(int i, char c, float x);
  int    g(char* s, const char* s2);
  double sqrtOfSumOfSquares(double a, double b);
}
```

After this you simply call the function just as if it were a C++ function:

```
int main()
{
  f(7, 'x', 3.14);   // Note: nothing unusual in the call
  // ...
}
```

FAQ How can I create a C++ function `f(int,char,float)` that is callable by my C code?

The C++ compiler must know that `f(int,char,float)` is to be called by a C compiler using the `extern "C"` construct:

```
// This is C++ code

// Declare f(int,char,float) using extern "C":
extern "C" void f(int i, char c, float x);

// ...

// Define f(int,char,float) in some C++ module:
void f(int i, char c, float x)
{
  // ...
}
```

The `extern "C"` line tells the compiler that the external information sent to the linker should use C calling conventions and name mangling (e.g., preceded by a single underscore). Since name overloading isn't supported by C, you can't make several overloaded functions simultaneously callable by a C program.

FAQ Why is the linker giving errors for C/C++ functions being called from C++/C functions?

If you didn't get your `extern "C"` right, you'll sometimes get linker errors rather than compiler errors. This is due to the fact that C++ compilers usually "mangle" function names (e.g., to support function overloading) differently than C compilers.

See the previous two FAQs on how to use `extern "C"`.

FAQ How can I pass an object of a C++ `class` to/from a C function?

Here's an example (for info on `extern "C"`, see the previous two FAQs).

`Fred.h`:

```
/* This header can be read by both C and C++ compilers */
#ifndef FRED_H
#define FRED_H

#ifdef __cplusplus
  class Fred {
  public:
    Fred();
    void wilma(int);
  private:
    int a_;
  };
#else
  typedef
    struct Fred
      Fred;
#endif

#ifdef __cplusplus
extern "C" {
#endif

#if defined(__STDC__) || defined(__cplusplus)
  extern void c_function(Fred*);   /* ANSI C prototypes */
  extern Fred* cplusplus_callback_function(Fred*);
#else
  extern void c_function();        /* K&R style */
  extern Fred* cplusplus_callback_function();
#endif

#ifdef __cplusplus
}
#endif

#endif /*FRED_H*/
```

`Fred.cpp`:

```
// This is C++ code

#include "Fred.h"
```

```
Fred::Fred() : a_(0) { }

void Fred::wilma(int a) { }

Fred* cplusplus_callback_function(Fred* fred)
{
  fred->wilma(123);
  return fred;
}
```

`main.cpp`:

```
// This is C++ code

#include "Fred.h"

int main()
{
  Fred fred;
  c_function(&fred);
  // ...
}
```

`c-function.c`:

```
/* This is C code */

#include "Fred.h"

void c_function(Fred* fred)
{
  cplusplus_callback_function(fred);
}
```

Unlike your C++ code, your C code will not be able to tell that two pointers point at the same object unless the pointers are *exactly* the same type. For example, in C++ it is easy to check if a `Derived*` called `dp` points to the same object as is pointed to by a `Base*` called `bp`: just say `if (dp == bp) ...`. The C++ compiler automatically converts both pointers to the same type, in this case to `Base*`, then compares them. Depending on the C++ compiler's implementation details, this conversion sometimes changes the bits of a pointer's value.

(Technical aside: Most C++ compilers use a binary object layout that causes this conversion to happen with multiple inheritance and/or virtual inheritance. However the C++ language does not impose that object layout so in principle a conversion could also happen even with non-virtual single inheritance.)

The point is simple: your C compiler will not know how to do that pointer conversion, so the conversion from `Derived*` to `Base*`, for example, must take place in code compiled with a C++ compiler, not in code compiled with a C compiler.

NOTE: you must be especially careful when converting both to `void*` since that conversion will not allow either the C or C++ compiler to do the proper pointer adjustments! The comparison `(x == y)` might be `false` even if `(b == d)` is `true`:

```
void f(Base* b, Derived* d)
```

```
{
  if (b == d) {   ☺ Validly compares a Base* to a Derived*
    // ...
  }

  void* x = b;
  void* y = d;
  if (x == y) {   ☹ BAD FORM! DO NOT DO THIS!
    // ...
  }
}
```

As stated above, the above pointer conversions will typically happen with multiple and/or virtual inheritance, but *please* do *not* look at that as an exhaustive list of the *only* times when the pointer conversions will happen.

You have been warned.

If you really want to use `void*` pointers, here is the safe way to do it:

```
void f(Base* b, Derived* d)
{
  void* x = b;
  void* y = static_cast<Base*>(d);  // If conversion is needed, it will happen i
  if (x == y) {   // ☺ Validly compares a Base* to a Derived*
    // ...
  }
}
```

FAQ Can my C function directly access data in an object of a C++ `class`?

Sometimes.

(For basic info on passing C++ objects to/from C functions, read the previous FAQ).

You can safely access a C++ object's data from a C function if the C++ class:

- Has no `virtual` functions (including inherited `virtual` functions)
- Has all its data in the same access-level section (private/protected/public)
- Has no fully-contained subobjects with `virtual` functions

If the C++ class has any base classes at all (or if any fully contained subobjects have base classes), accessing the data will *technically* be non-portable, since `class` layout under inheritance isn't imposed by the language. However in practice, all C++ compilers do it the same way: the base class object appears first (in left-to-right order in the event of multiple inheritance), and member objects follow.

Furthermore, if the class (or any base class) contains any `virtual` functions, almost all C++ compilers put a `void*` into the object either at the location of the first `virtual` function or at the very beginning of the object. Again, this is not required by the language, but it is the way "everyone" does it.

If the class has any `virtual` base classes, it is even more complicated and less portable. One common implementation technique is for objects to contain an object of the `virtual` base class (`V`) last (regardless of where `V` shows up as a `virtual` base class in the inheritance hierarchy). The rest of the object's parts appear in the normal order. Every derived class that has `V` as a `virtual` base class actually has a *pointer* to the `V` part of the final object.

FAQ Why do I feel like I'm "further from the machine" in C++ as opposed to C?

Because you are.

As an OO programming language, C++ allows you to model the problem domain itself, which allows you to program in the language of the problem domain rather than in the language of the solution domain.

One of C's great strengths is the fact that it has "no hidden mechanism": what you see is what you get. You can read a C program and "see" every clock cycle. This is not the case in C++; old line C programmers (such as many of us once were) are often ambivalent (can you say, "hostile"?) about this feature. However after they've made the transition to OO thinking, they often realize that although C++ hides some mechanism from the programmer, it also provides a level of abstraction and economy of expression which lowers maintenance costs without destroying run-time performance.

Naturally you can write bad code in any language; C++ doesn't guarantee any particular level of quality, reusability, abstraction, or any other measure of "goodness."

C++ doesn't try to make it impossible for bad programmers to write bad programs; it enables reasonable developers to create superior software.