Article

Mixing C and C++ Code in the Same Program

Print-friendly Version

By Stephen Clamage, Sun Microsystems, Sun ONE Studio Solaris Tools Development Engineering

The C++ language provides mechanisms for mixing code that is compiled by compatible C and C++ compilers in the same program. You can experience varying degrees of success as you port such code to different platforms and compilers. This article shows how to solve common problems that arise when you mix C and C++ code, and highlights the areas where you might run into portability issues. In all cases we show what is needed when using Sun C and C++ compilers.

Contents

Using Compatible Compilers

The first requirement for mixing code is that the C and C++ compilers you are using must be compatible. They must, for example, define basic types such as int, float or pointer in the same way. The Solaris Operating System (Solaris OS) specifies the Application Binary Interface (ABI) of C programs, which includes information about basic types and how functions are called. Any useful compiler for the Solaris OS must follow this ABI.

Sun C and C++ compilers follow the Solaris OS ABI and are compatible. Third-party C compilers for the Solaris OS usually also follow the ABI. Any C compiler that is compatible with the Sun C compiler is also compatible with the Sun C++ compiler.

The C runtime library used by your C compiler must also be compatible with the C++ compiler. C++ includes the standard C runtime library as a subset, with a few differences. If the C++ compiler provides its own versions of of the C headers, the versions of those headers used by the C compiler must be compatible.

Sun C and C++ compilers use compatible headers, and use the same C runtime library. They are fully compatible.

Accessing C Code From Within C++ Source

The C++ language provides a "linkage specification" with which you declare that a function or object follows the program linkage conventions for a supported language. The default linkage for objects and functions is C++. All C++ compilers also support C linkage, for some compatible C compiler.

When you need to access a function compiled with C linkage (for example, a function compiled by the C compiler), declare the function to have C linkage. Even though most C++ compilers do not have different linkage for C and C++ data objects, you should declare C data objects to have C linkage in C++ code. With the exception of the pointer-to-function type, types do not have C or C++ linkage.

Declaring Linkage Specifications
Use one of the following notations to declare that an object or function has the linkage of language *language_name*:

```
extern "language_name" declaration ;
extern "language_name" { declaration ; declaration ; ... }
```

The first notation indicates that the declaration (or definition) that immediately follows has the linkage of *language_name*. The second notation indicates that everything between the curly braces has the linkage of *language_name*, unless declared otherwise. Notice that you do not use a semicolon after the closing curly brace in the second notation.

You can nest linkage specifications, but they do not create a scope. Consider the following example:

```
extern "C" {
    void f();             // C linkage
    extern "C++" {
        void g();         // C++ linkage
        extern "C" void h(); // C linkage
        void g2();        // C++ linkage
    }
    extern "C++" void k();// C++ linkage
    void m();             // C linkage
}
```

All the functions above are in the same global scope, despite the nested linkage specifiers.

Including C Headers in C++ Code
If you want to use a C library with its own defining header that was intended for C compilers, you can include the header in `extern "C"` brackets:

```
extern "C" {
    #include "header.h"
}
```

Warning-

Do not use this technique for system headers on the Solaris OS. The Solaris headers, and all the headers that come with Sun C and C++ compilers, are already configured for use with C and C++ compilers. You can invalidate declarations in the Solaris headers if you specify a linkage.

Creating Mixed-Language Headers
If you want to make a header suitable for both C and C++ compilers, you could put all the declarations inside `extern "C"` brackets, but the C compiler does not recognize the syntax. Every C++ compiler predefines the macro `__cplusplus`, so you can use that macro to guard the C++ syntax extensions:

```
#ifdef __cplusplus
```

```
extern "C" {
#endif
... /* body of header */
#ifdef __cplusplus
} /* closing brace for extern "C" */
#endif
```

Adding C++ features to C structs
Suppose you want to make it easier to use a C library in your C++ code. And suppose that instead of using C-style access you might want to add member functions, maybe virtual functions, possibly derive from the class, and so on. How can you accomplish this transformation and ensure the C library functions can still recognize the struct? Consider the uses of the C `struct buf` in the following example:

```
struct buf {
    char* data;
    unsigned count;
};
void buf_clear(struct buf*);
int  buf_print(struct buf*); /* return status, 0 means fail */
int  buf_append(struct buf*, const char*, unsigned count); /* same return */
```

You want to turn this struct into a C++ class and make it easier to use with the following changes:

```
extern "C" {
  #include "buf.h"
}
class mybuf { // first attempt -- will it work?
public:
    mybuf() : data(0), count(0) { }
    void clear() { buf_clear((buf*)this); }
    bool print() { return buf_print((buf*)this); }
    bool append(const char* p, unsigned c)
        { return buf_append((buf*)this, p, c); }
private:
    char* data;
    unsigned count;
};
```

The interface to the `class mybuf` looks more like C++ code, and can be more easily integrated into an Object-Oriented style of programming -- if it works.

What happens when the member functions pass the `this` pointer to the `buf` functions? Does the C++ class layout match the C layout? Does the `this` pointer point to the `data` member, as a pointer to `buf` does? What if you add virtual functions to `mybuf`?

The C++ standard makes no promises about the compatibility of `buf` and `class mybuf`. This code, without virtual functions, might work, but you can't count on it. If you add virtual functions, the code will fail using compilers that add extra data (such as pointers to virtual tables) at the beginning of a class.

The portable solution is to leave `struct buf` strictly alone, even though you would like to protect the data members and provide access only through member functions. You can guarantee C and C++ compatibility only if you leave the declaration unchanged.

You can derive a C++ `class mybuf` from the C `struct buf`, and pass pointers to the `buf` base class to the `mybuf` functions. If a pointer to `mybuf` doesn't point to the beginning of the `buf` data, the C++

compiler will adjust it automatically when converting a `mybuf*` to a `buf*`. The layout of `mybuf` might vary among C++ compilers, but the C++ source code that manipulates `mybuf` and `buf` objects will work everywhere. The following example shows a portable way to add C++ and Object-Oriented features to a C struct.

```
extern "C" {
  #include "buf.h"
}
class mybuf : public buf { // a portable solution
public:
    mybuf() : data(0), count(0) { }
    void clear() { buf_clear(this); }
    bool print() { return buf_print(this); }
    bool append(const char* p, unsigned c)
        { return buf_append(this, p, c); }
};
```

C++ code can freely create and use `mybuf` objects, passing them to C code that expects `buf` objects, and everything will work together. Of course, if you add data to `mybuf`, the C code won't know anything about it. That's a general design consideration for class hierarchies. You also have to take care to create and delete `buf` and `mybuf` objects consistently. It is safest to let C code delete (free) an object if it was created by C code, and not allow C code to delete a `mybuf` object.

Accessing C++ Code From Within C Source

If you declare a C++ function to have C linkage, it can be called from a function compiled by the C compiler. A function declared to have C linkage can use all the features of C++, but its parameters and return type must be accessible from C if you want to call it from C code. For example, if a function is declared to take a reference to an IOstream class as a parameter, there is no (portable) way to explain the parameter type to a C compiler. The C language does not have references or templates or classes with C++ features.

Here is an example of a C++ function with C linkage:

```
#include <iostream>
extern "C" int print(int i, double d)
{
    std::cout << "i = " << i << ", d = " << d;
}
```

You can declare function `print` in a header file that is shared by C and C++ code:

```
#ifdef __cplusplus
extern "C"
#endif
int print(int i, double d);
```

You can declare at most one function of an overloaded set as `extern "C"` because only one C function can have a given name. If you need to access overloaded functions from C, you can write C++ wrapper functions with different names as the following example demonstrates:

```
int    g(int);
double g(double);
```

```
extern "C" int    g_int(int i)       { return g(i); }
extern "C" double g_double(double d) { return g(d); }
```

Here is the example C header for the wrapper functions:

```
int g_int(int);
double g_double(double);
```

You also need wrapper functions to call template functions because template functions cannot be declared as `extern "C"`:

```
template<class T> T foo(T t) { ... }
extern "C" int   foo_of_int(int t) { return foo(t); }
extern "C" char* foo_of_charp(char* p) { return foo(p); }
```

C++ code can still call the the overloaded functions and the template functions. C code must use the wrapper functions.

Accessing C++ Classes From C

Can you access a C++ class from C code? Can you declare a C struct that looks like a C++ class and somehow call member functions? The answer is yes, although to maintain portability you must add some complexity. Also, any modifications to the definition of the C++ class you are accessing requires that you review your C code.

Suppose you have a C++ class such as the following:

```
class M {
public:
    virtual int foo(int);
    // ...
private:
    int i, j;
};
```

You cannot declare class `M` in your C code. The best you can do is to pass around pointers to class `M` objects, similar to the way you deal with FILE objects in C Standard I/O. You can write `extern "C"` functions in C++ that access class `M` objects and call them from C code. Here is a C++ function designed to call the member function `foo`:

```
extern "C" int call_M_foo(M* m, int i) { return m->foo(i); }
```

Here is an example of C code that uses class `M`:

```
struct M; /* you can supply only an incomplete declaration */
int call_M_foo(struct M*, int); /* declare the wrapper function */
int f(struct M* p, int j) /* now you can call M::foo */
    { return call_M_foo(p, j); }
```

Mixing IOstream and C Standard I/O

You can use C Standard I/O, from the standard C header <stdio.h>, in C++ programs because C Standard I/O is part of C++.

Any considerations about mixing IOstream and Standard I/O in the same program therefore do not depend on whether the program contains C code specifically. The issues are the same for purely C++ programs that use both Standard I/O and IOstreams.

Sun C and C++ use the same C runtime libraries, as noted in the section about compatible compilers. Using Sun compilers, you can therefore use Standard I/O functions freely in both C and C++ code in the same program.

The C++ standard says you can mix Standard I/O functions and IOstream functions on the same target "stream", such as the standard input and output streams. But C++ implementations vary in their compliance. Some systems require that you call the `sync_with_stdio()` function explicitly before doing any I/O. Implementations also vary in the efficiency of I/O when you mix I/O styles on the same stream or file. In the worst case, you get a system call per character input or output. If the program does a lot of I/O, the performance might be unacceptable.

The safest course is to stick with Standard I/O or IOstream styles on any given file or standard stream. Using Standard I/O on one file or stream and IOstream on a different file or stream does not cause any problems.

Working with Pointers to Functions

A pointer to a function must specify whether it points to a C function or to a C++ function, because it is possible that C and C++ functions use different calling conventions. Otherwise, the compiler does not know which kind of function-calling code to generate. Most systems do not have have different calling conventions for C and C++, but C++ allows for the possibility. You therefore must be careful about declaring pointers to functions, to ensure that the types match. Consider the following example:

```
typedef int (*pfun)(int);  // line 1
extern "C" void foo(pfun); // line 2
extern "C" int g(int)      // line 3
...
foo( g ); // Error!        // line 5
```

Line 1 declares `pfun` to point to a C++ function, because it lacks a linkage specifier.
Line 2 therefore declares `foo` to be a C function that takes a pointer to a C++ function.
Line 5 attempts to call `foo` with a pointer to `g`, a C function, a type mis-match.

Be sure to match the linkage of a pointer-to-function with the functions to which it will point. In the following corrected example, all declarations are inside `extern "C"` brackets, ensuring that the types match.

```
extern "C" {
    typedef int (*pfun)(int);
    void foo(pfun);
    int g(int);
}
```

```
foo( g ); // now OK
```

Pointers to functions have one other subtlety that occasionally traps programmers. A linkage specification applies to all the parameter types and to the return type of a function. If you use the elaborated declaration of a pointer-to-function in a function parameter, a linkage specification on the function applies to the pointer-to-function as well. If you declare a pointer-to-function using a `typedef`, the linkage specification of that `typedef` is not affected by using it in a function declaration. For example, consider this code:

```
typedef int (*pfn)(int);
extern "C" void foo(pfn p) { ... }       // definition
extern "C" void foo( int (*)(int) ); // declaration
```

The first two lines might appear in a program file, and the third line might appear in a header where you don't want to expose the name of the private typedef. Although you intended for the declaration of `foo` and its definition to match, they do not. The definition of `foo` takes a pointer to a C++ function, but the declaration of `foo` takes a pointer to a C function. The code declares a pair of overloaded functions.

To avoid this problem, use typedefs consistently in declarations, or enclose the typedefs in appropriate linkage specifications. For example, assuming you wanted `foo` to take a pointer to a C function, you could write the definition of `foo` this way:

```
extern "C" {
    typedef int (*pfn)(int);
    void foo(pfn p) { ... }
}
```

Working with C++ Exceptions

Propagating Exceptions

What happens if you call a C++ function from a C function, and the C++ function throws an exception? The C++ standard is somewhat vague about whether you can expect exceptions to behave properly, and on some systems you have to take special precautions. Generally, you must consult the user manuals to determine whether the code will work properly.

No special precautions are necessary with Sun C++. The exception mechanism in Sun C++ does not affect the way functions are called. If a C function is active when a C++ exception is thrown, the C function is passed over in the process of handling the exception.

Mixing Exceptions with `set_jmp` and `long_jmp`

The best advice is not to use `long_jmp` in programs that contain C++ code. The C++ exception mechanism and C++ rules about destroying objects that go out of scope are likely to be violated by a `long_jmp`, with unpredictable results. Some compilers integrate exceptions and `long_jmp`, allowing them to work together, but you should not depend on such behavior. Sun C++ uses the same `set_jmp` and `long_jmp` as the C compiler.

Many C++ experts believe that `long_jmp` should not be integrated with exceptions, due to the difficulty of specifying exactly how it should behave.

If you use `long_jmp` in C code that you are mixing with C++, ensure that a `long_jmp` does not cross over an active C++ function. If you cannot ensure that, see if you can compile that C++ code with exceptions disabled. You still might have problems if the destructors of local objects are bypassed.

Linking the Program

At one time, most C++ compilers required that function `main` be compiled by the C++ compiler. That requirement is not common today, and Sun C++ does not require it. If your C++ compiler needs to compile the main function but you cannot do so for some reason, you can change the name of the C main function and call it from a wrapper version of C++ `main`. For example, change the name of the C main function to `C_main`, and write this C++ code:

```
extern "C" int C_main(int, char**); // not needed for Sun C++
int main(int argc, char** argv) { return C_main(argc, argv); }
```

Of course, `C_main` must be declared in the C code to return an int, and it will have to return an int value. As noted above, you do not need to go to this trouble with Sun C++.

Even if your program is primarily C code but makes use of C++ libraries, you need to link C++ runtime support libraries provided with the C++ compiler into your program. The easiest and best way to do that is to use the C++ compiler driver to do the linking. The C++ compiler driver knows what libraries to link, and the order in which to link them. The specific libraries can depend on the options used when compiling the C++ code.

Suppose you have C program files `main.o`, `f1.o`, and `f2.o`, and you use a C++ library `helper.a`. With Sun C++, you would issue the command

```
CC -o myprog main.o f1.o f2.o helper.a
```

The necessary C++ runtime libraries like libCrun and libCstd are linked automatically. The documentation for `helper.a` might require that you use additional link-time options. If you can't use the C++ compiler for some reason, you can use the `-dryrun` option of the CC command to get the list of commands the compiler issues, and capture them into a shell script. Since the exact commands depend on command-line options, you should review the output from `-dryrun` with any change of the command line.

For More Information

- Sun ONE Studio C/C++ Documentation
  Latest information on the Sun ONE C and C++ compilers and tools, including man pages and readme files.

About the Author

Steve Clamage has been at Sun since 1994. He is currently technical lead for the C++ compiler and the Sun ONE Studio Compiler Collection. He has been chair of the ANSI C++ Committee since 1995.