Name mangling and linking C objects into C++ code

March 17, 2016

example C code

```c
// hellofunc.c
#include <stdio.h>
void myPrintHelloMake (void)
{
  printf("Hello makefiles!\n");
}
```

header for `hellofunc.h`

```
// hellomake.h
#ifdef __cplusplus
extern "C"{
#endif
void myPrintHelloMake (void);
#ifdef __cplusplus
}
#endif
extern "C" { void myprinthellomake_ (void); }
```

- the preprocessor macro __cplusplus is defined by default whenever using a C++ compiler
- the effect of the **#ifdef** __cplusplus conditional can be verified by running the preprocessor

example C++ code

```
// hellomake.cpp
#include <hellomake.h>
#include <iostream>
int main()
{
  // call function from other file
  myprinthellomake_ ();
  std::cout << "Hello from C++ as well!"
            << std::endl;
  return (0);
}
```

invoking the proprocessor

using g++ the lines between conditionals are included in the output of the preprocessor

```
$ g++ -E hellomake.h
```

```
# 1 "hellomake.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "hellomake.h"
extern "C"{
void myPrintHelloMake (void);
}
```

using gcc the lines between conditionals are *not* included in the output of the preprocessor

```
$ gcc -E hellomake.h
```

```
# 1 "hellomake.h"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "hellomake.h"
void myPrintHelloMake (void);
```

C vs. C++ linkage

Why is the **extern** ”C”{...} needed?

Let us compile hellofunc.c using the C compiler and inspect the symbols in the object file

```
$ gcc -c hellofunc.c
$ nm hellofunc.o | grep myPrintHelloMake

0000000000000000 T _myPrintHelloMake
```

The name of the compiled function in the object file is (almost) the same as defined in the C source code.

C vs. C++ linkage

Why is the **extern** ”C”{...} needed?

Let us compile hellofunc.c using the C++ compiler and inspect the symbols in the object file

```
$ g++ -c hellofunc.c
$ nm hellofunc.o | grep myPrintHelloMake

0000000000000000 T __Z16myPrintHelloMakev
```

The name of the compiled function in the object file is surrounded by additional unreadable charecters.

why?

C++ name mangling

Functions in C++ are distinguished not only by their name, but also by the type of their input parameters.

The C++ compiler encodes the full signature of the function in the name stored in the object file, this is called *name mangling*.

The name can be *demangled* using the c++ filt utility

```
$ g++ -c hellofunc.c
$ nm hellofunc.o | grep myPrintHelloMake | c++filt

0000000000000000 T myPrintHelloMake()
```

The information that the function requires no input parameters is also stored in the object file!

When invoking a C function in C++ code, the compiler must be told to search for a symbol that has C linkage and is given an unmangled label in the object file, this is the purpose of the **extern** "C" {...}

what happens specifying C linkage?

Let us run the compiler on hellomake.cpp

```
$ g++ -c hellomake.cpp -I.
$ nm hellomake.o | grep myPrintHelloMake
```

```
U _myPrintHelloMake
```

The object file has an *undefined symbol* with the *unmangled name*. This means the linker expects the symbol with the *unmangled name* to exist when linking the final binary. If we run the C compiler on hellofunc.c this symbol will be defined in hellofunc.o so we can link the two object files and form an executable

```
$ gcc -c hellofunc.c
$ g++ -c hellomake.cpp -I.
$ g++ hellomake.o hellofunc.o -o hellomake
$ ./hellomake
```

```
Hello makefiles!
Hello from C++ as well!
```

what happens without specifying C linkage?

Let us comment out the lines specifying C linkage in the hellomake.h header

```
//#ifdef __cplusplus
//extern "C"{
//#endif
void myPrintHelloMake (void);
//#ifdef __cplusplus
//}
//#endif
```

and run the compiler on hellomake.cpp

```
$ g++ -c hellomake.cpp -I.
$ nm hellomake.o | grep myPrintHelloMake

                 U __Z16myPrintHelloMakev
```

The object file has an *undefined symbol* with the *mangled name*. This means the linker expects the symbol with the *mangled name* to exist when linking the final binary, and will error out if it isn't!

what happens without specifying C linkage?

Let us try to link the executable

```
$ gcc -c hellofunc.c
$ g++ -c hellomake.cpp -I.
$ g++ hellofunc.o hellomake.o -o hellomake

Undefined symbols for architecture x86_64:
  "myPrintHelloMake()", referenced from:
      _main in hellomake.o
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
```

what happens without specifying C linkage?

If I had compiled the C file with the C++ compiler linking would have worked

```
$ g++ -c hellofunc.c
$ g++ -c hellomake.cpp -I.
$ g++ hellofunc.o hellomake.o -o hellomake
$ ./hellomake

Hello makefiles!
Hello from C++ as well!
```

This cannot always be done (especially if linking to a C library that is not compiled by yourself!)

example F77 code

It is (unfortunaley) still common to find numerical libraries written in fortran, so let us consider a simple fortran 77 subroutine

```
c       hellofunc.f
        subroutine myPrintHelloMake ()
        write (*,*) 'Hello makefiles!'
        end
```

linking fortran objects

The situation when linking to fortran77 objects is similar, but has some differnces

```
$ nm hellofunc.o | grep -i myPrintHelloMake
$ gfortran -c hellofunc.f

0000000000000000 T _myprinthellomake_
```

- fortran is case insensitive
- an additional underscore has been added to the object name (this depends on the compiler)

To use the f77 subroutine in a C++ program we need to define in the header

```
extern "C" { void myprinthellomake_ (void); }
```