Mixing Code in C, C++, and FORTRAN on Unix

It is becoming increasingly common for engineers to write different parts of a final program in different languages. For example, you might use legacy FORTRAN code for calculations, C++ code for writing a graphical interface to the program, and C for other system functions. This page is intended to show simple examples of how to write programs and functions that can be accessed from other languages.

Major points to keep in mind when mixing languages include:

Call By Value / Call By Reference

C and C++ default to passing arguments by value; FORTRAN defaults to passing arguments by reference. In other words, the normal way that FORTRAN subroutines or functions are called allows them to modify their argument variables inside the subroutine code, while C and C++ do not. C and C++ subroutines use a slightly different syntax to allow for modification of arguments, and this syntax is consistently used in the following examples.

Splitting Code Into Multiple Files

Normally, as your program code grows larger, you'll want to separate it into several files, one per function or subroutine. Then each source code file is compiled into an object file (a .o file in Unix, or a .obj file in Windows), and the various object files are linked together into a final single executable.

The advantages of splitting your code up this way include:

- Being able to use different languages for different portions of the program
- Being able to delegate the writing of different functions to different people
- More efficient compilation, since a change to one source file only requires its object file to be recompiled and the object files to be relinked, rather than recompiling the entire body of code from scratch.

Once you exceed the two or three source code files we use in the following examples, you'll almost definitely want to investigate using the make utility to automate the build process on your program.

Internal Function Names

When the compiler turns source code into object files, it might change the internal name of the function, for example, by appending or prepending underscores. In Unix, you can use the `nm` command to list those internal names:

```
mwr@ch208m:~$ nm cppfunction.o
cppfunction.o:

  [Index]   Value      Size    Type  Bind  Other Shndx   Name

[2]     |         0|       0|SECT |LOCL |0    |4      |
[5]     |         0|       0|SECT |LOCL |0    |2      |
[6]     |         0|       0|SECT |LOCL |0    |3      |
[4]     |         0|       0|NOTY |LOCL |0    |4      |__FRAME_BEGIN__
[7]     |         0|      76|FUNC |GLOB |0    |2      |cppfunction
[1]     |         0|       0|FILE |LOCL |0    |ABS    |cppfunction.C
[3]     |         0|       0|NOTY |LOCL |0    |2      |gcc2_compiled.
mwr@ch208m:~$ nm ffunction.o
```

```
ffunction.o:

[Index]   Value      Size    Type  Bind  Other Shndx   Name

[3]     |         0|       0|SECT |LOCL |0    |2      |
[4]     |         0|       0|SECT |LOCL |0    |3      |
[1]     |         0|       0|FILE |LOCL |0    |ABS    |ffunction.f
[5]     |         0|      68|FUNC |GLOB |0    |2      |ffunction_
[2]     |         0|       0|NOTY |LOCL |0    |2      |gcc2_compiled.
mwr@ch208m:~$
```

Note that the FORTRAN compiler appended a single underscore to the function name, while the C++ compiler left the name intact.

Example 1: Main Program in C, with Subroutines in C, C++, and FORTRAN

The C program is nothing out of the ordinary: it defines two variables, and calls various functions that change those variables' values. C requires that we use a "call by reference" syntax to make these changes persistent, rather than its default "call by value" method. Note that the name of the FORTRAN function called from the C program is `ffunction_`, a name we extracted via the nm command shown above. Note also that the C++ function has an `extern "C"` directive above the code of the function, indicating not that `cppfunction()` is written in C, but that it is called from a C-style interface instead of a C++ interface.

File cprogram.c:

```
#include <stdio.h>
int main(void) {
  float a=1.0, b=2.0;

  printf("Before running Fortran function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  ffunction_(&a,&b);
  printf("After running Fortran function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);

  printf("Before running C++ function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  cppfunction(&a,&b);
  printf("After running C++ function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);

  printf("Before running C function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);
  cfunction(&a,&b);
  printf("After running C function:\n");
  printf("a=%f\n",a);
  printf("b=%f\n",b);

  return 0;
}
```

File ffunction.f:

```
      subroutine ffunction(a,b)
      a=3.0
      b=4.0
      end
```

File cppfunction.C:

```
extern "C" {
  void cppfunction(float *a, float *b);
}

void cppfunction(float *a, float *b) {
  *a=5.0;
  *b=6.0;
}
```

File cfunction1.c:

```
void cfunction(float *a, float *b) {
  *a=7.0;
  *b=8.0;
}
```

Compilation Steps: each program is compiled into an object file using the appropriate compiler with the `-c` flag. After all the object files are created, the final `gcc` command links the object files together into a single executable:

```
  mwr@ch208m:~$ gcc -c cprogram.c
  mwr@ch208m:~$ g77 -c ffunction.f
  mwr@ch208m:~$ g++ -c cppfunction.C
  mwr@ch208m:~$ gcc -c cfunction1.c
  mwr@ch208m:~$ gcc -o cprogram cprogram.o ffunction.o cppfunction.o cfunction1.o
  mwr@ch208m:~$
```

Though this example problem does not require it, many of the math functions (for example, `sin`, `cos`, `pow`, etc.) require that you also link in the `libm` math library. Add a `-lm` flag to the final `gcc` command above to link in the math library.

Results:

```
mwr@ch208m:~$ ./cprogram
Before running Fortran function:
a=1.000000
b=2.000000
After running Fortran function:
a=3.000000
b=4.000000
Before running C++ function:
a=3.000000
b=4.000000
After running C++ function:
a=5.000000
b=6.000000
Before running C function:
a=5.000000
b=6.000000
After running C function:
a=7.000000
b=8.000000
mwr@ch208m:~$
```

Example 2: Main Program in C++, with Subroutines in C, C++, and FORTRAN

The only differences between the C++ code in a normal program is that we have to account for the C and FORTRAN subroutines expecting to be called with a C-style interface, and also that the FORTRAN compiler will append an underscore to the FORTRAN function name. Also, since the C++ function is held in a separate file from the C++ main program, we need to declare a function prototype before the main program code.

File cppprogram.C:

```
#include <iostream.h>

extern "C" {
  void ffunction_(float *a, float *b);
}

extern "C" {
  void cfunction(float *a, float *b);
}

void cppfunction(float *a, float *b);

int main() {
  float a=1.0, b=2.0;

  cout << "Before running Fortran function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  ffunction_(&a,&b);

  cout << "After running Fortran function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  cout << "Before running C function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  cfunction(&a,&b);

  cout << "After running C function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  cout << "Before running C++ function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  cppfunction(&a,&b);

  cout << "After running C++ function:" << endl;
  cout << "a=" << a << endl;
  cout << "b=" << b << endl;

  return 0;
}
```

File cfunction1.c:

```
void cfunction(float *a, float *b) {
  *a=7.0;
  *b=8.0;
}
```

File ffunction.f:

```
      subroutine ffunction(a,b)
      a=3.0
      b=4.0
      end
```

Compilation Steps:

```
mwr@ch208m:~$ g++ -c cppprogram.C
mwr@ch208m:~$ gcc -c cfunction1.c
mwr@ch208m:~$ g++ -c cppfunction1.C
mwr@ch208m:~$ g77 -c ffunction.f
```

```
mwr@ch208m:~$ g++ -o cppprogram cppprogram.o cfunction1.o cppfunction1.o ffunction.o
mwr@ch208m:~$
```

Results:

```
mwr@ch208m:~$ ./cppprogram
Before running Fortran function:
a=1
b=2
After running Fortran function:
a=3
b=4
Before running C function:
a=3
b=4
After running C function:
a=7
b=8
Before running C++ function:
a=7
b=8
After running C++ function:
a=5
b=6
mwr@ch208m:~$
```

Example 3: Main Program in FORTRAN, with Subroutines in C, C++, and Fortran

Though the non-FORTRAN subroutines don't have any underscores after their names in the main FORTRAN program, running the `nm` command on fprogram.o shows that the FORTRAN program expects that they'll have underscores appended to the function name internally. Therefore, in both the C and C++ files, we need to write the function prototype statement accordingly, with an underscore after the function name. We also must use the `extern "C"` directive in the C++ file, indicating that the C++ function is called with a C-style interface, similar to the first example.

File fprogram.f:

```
      program fprogram
      real a,b
      a=1.0
      b=2.0

      print*,'Before Fortran function is called:'
      print*,'a=',a
      print*,'b=',b
      call ffunction(a,b)
      print*,'After Fortran function is called:'
      print*,'a=',a
      print*,'b=',b

      print*,'Before C function is called:'
      print*,'a=',a
      print*,'b=',b
      call cfunction(a,b)
      print*,'After C function is called:'
      print*,'a=',a
      print*,'b=',b

      print*,'Before C++ function is called:'
      print*,'a=',a
      print*,'b=',b
      call cppfunction(a,b)
      print*,'After C++ function is called:'
      print*,'a=',a
      print*,'b=',b
```

```
      stop
      end
```

File ffunction.f:

```
      subroutine ffunction(a,b)
      a=3.0
      b=4.0
      end
```

File cppfunction2.C:

```
extern "C" {
  void cppfunction_(float *a, float *b);
}

void cppfunction_(float *a, float *b) {
  *a=5.0;
  *b=6.0;
}
```

File cfunction2.c:

```
void cfunction_(float *a, float *b) {
  *a=7.0;
  *b=8.0;
}
```

Compilation Steps:

```
mwr@ch208m:~$ g77 -c fprogram.f
mwr@ch208m:~$ g77 -c ffunction.f
mwr@ch208m:~$ g++ -c cppfunction2.C
mwr@ch208m:~$ gcc -c cfunction2.c
mwr@ch208m:~$ g77 -lc -o fprogram fprogram.o ffunction.o cppfunction2.o cfunction2.o
mwr@ch208m:~$
```

Results:

```
mwr@ch208m:~$ ./fprogram
 Before Fortran function is called:
 a=  1.
 b=  2.
 After Fortran function is called:
 a=  3.
 b=  4.
 Before C function is called:
 a=  3.
 b=  4.
 After C function is called:
 a=  7.
 b=  8.
 Before C++ function is called:
 a=  7.
 b=  8.
 After C++ function is called:
 a=  5.
 b=  6.
mwr@ch208m:~$
```

Filed under: C, C++, FORTRAN