Mixing Pascal, Fortran, and C with GCC

Contents

Introduction
Installation
Run-Time Libraries
Makefile
Names
Simple Arguments
Array Arguments
Procedural Arguments
Wrapper Routines
Global Variables
Demonstration
Fortran Arguments
Conclusion

Introduction

We describe how to use the GNU Compiler Collection (GCC) to mix C, C++, Fortran, and Pascal code in a single executable file. Effective mixing allows us to link existing well-tested code in Fortran and Pascal, to modern code written in C. Effective mixing also allows programmers who prefer one language to work on new projects with programmers who prefer another. With GCC, effective mixing is practical. The GCC compilers all share the same object format. Each can link to objects and libraries created by any of the other compilers. Furthermore, the GCC compilers are available for Windows, Linux, MacOS, and UNIX. Once you get your mixing to work on one platform, it will be portable to other platforms.

Suppose your main program is in C, and you want it to call routines written in C, Pascal, and Fortran. To mix the three languages with GCC, you compile the C, Pascal, and Fortran files into objects with their respective compilers. Each object refers to its own source language's run-time library. You link the objects and the run-time libraries together to make your executable or shared library.

The steps required for effective mixing with GCC are as follows.

1. install the compilers
2. find the correct run-time libraries
3. match the actual and expected names of routines in the objects
4. match the actual and expected argument types in the objects

In the sections, below, we describe how we take each of these steps. We find that mixing is now straightforward for us, and we have no doubt that it will be for you too.

Installation

Download our collection of files, which you will find in Mixing.zip. Here is a list of files in the zip archive, with a short description of each.

File	Description
C_Library.c	C source code for the example C library.
Pascal_Library.p	Pascal source code for the example Pascal library.
Fortran_Library.f	Fortran source code for the example Fortran library.
Maine.c	C main program that calls routines from all libraries.

Makefile	Makefule for main program.
Examples/LWDAQ_Makefile	Make file for lwdaq.so.
Examples/Minuit_Main.c	C program that uses Minuit.h to call Minuit.
Examples/Minuit_Main.p	Pascal program that uses Minuit.p to call Minuit.
Examples/Minuit_Makefile	Make file for Minuit_Main.c and Minuit_Main.p.
Examples/Minuit.h	C interface with Fortran Minuit library.
Examples/Minuit.p	Pascal interface with Fortran Minuit library.
Examples/F_easyio.f	Fortran I/O example program.
Examples/F_linfit.f	Fortran Minuit-calling example program.
Examples/F_pass.f	Fortran argument-passing example program.

Table: Files in Mixing.zip.

Main.c calls routines from C_Library.c, Fortran_Library.f, and Pascal_Library.p. To compile Main.c, you must first compile each of these files with its corresponding compiler. We compile C_Library (and Main.c) with gcc, the GNU Compiler Collection C compiler. We compile Fortran_Library.f with g77, the GNU Fortran 77 compiler. We compile Pascal_Library.p with gpc, the GNU Pascal Compiler. Each of these compilers is available on Windows (with Cygwin), MacOS, Linux, and many varieties of UNIX.

Install the GNU Compiler Collection's Fortran, Pascal, and C-compilers. Most likely you have the C-compiler already. There is a choice of Fortran compilers: Fortran 95 or Fortan 77. We use Fortran 77, because it compiles the really old, fast, Fortran code still in use in High Energy Physics. We will not attempt to provide you with a comprehensive set of links to Pascal, Fortran, and C compilers. If you Google GCC, Fortran, Pascal, and your operating system name, you will find them eventually. If you're working on Linux, MacOS, or UNIX, you almost certainly have the C-compiler installed already. On Windows, we recommend you install the Linux Interface progam for Windows, called Cygwin. When you install Cygwin, you can equip it with the C, Fortran, and Pascal compilers.

Run-Time Libraries

Not only will you need the three compilers, you will also need the C, Pascal and Fortran run-time libraries to make the final link with possible. You will need libgpc.a and libf2c.a for our demonstrations. These are the Pascal and Fortran run-time libraries for GCC. You will have to compile or find these libraries for yourself.

If your main program is in C or Pascal, and you are calling Fortran routines, you will need to name the Fortran run-time library explicitly when you link the main program. If you have g77 installed, you should have a library called libg2c.a. This is the Fortran run-time library. You can use this instead of the one we refer to in our Makefile. The one we refer to in our Makefile is libf2c.a, which we obtain by comiling the f2c source code with gcc. The f2c run-time library is more compact than libg2c.a. Perhaps it is missing some routines, and some of you will need to use libg2c.a.

If your main program is in C or Fortran, and you are calling Pascal routines, you will have to name the Pascal run-time library when you link the main program. If you have gpc installed, you will have a library called libgpc.a.

If your main program is in Fortran or Pascal, and you are calling C routines, you may have to name the C run-time library when you link the main program. Try linking without naming the C run-time library. The Pascal compiler has a C run-time library installed, so it should work without an explicit mention of the library. The Fortran compiler is also likely to work without explicit mention of the C run-time library. If you get an error, your gcc installation will have a library called libc.a, libc.dylib, or libc.so. Search around to find the one that works.

Makefile

Here are the contents of our makefile.

```
products: Main.exe

Main.exe: Main.o Fortran_Library.o Pascal_Library.o C_Library.o
	/usr/bin/gcc -o Main.exe Main.o \
		Fortran_Library.o libf2c.a \
		Pascal_Library.o libgpc.a \
```

```
                C_Library.o

Main.o: Main.c
        /usr/bin/gcc -c -o Main.o Main.c

Fortran_Library.o: Fortran_Library.f
        /usr/local/bin/g77 -c -o Fortran_Library.o Fortran_Library.f

Pascal_Library.o: Pascal_Library.p
        /usr/bin/gpc -c -o Pascal_Library.o Pascal_Library.p

C_Library.o: C_Library.c
        /usr/bin/gcc -c -o C_Library.o C_Library.c

clean:
        rm *.o Main.exe *.gpi
```

The Makefile file describes how the *make* utility should compile the source code and link objects in order to create the final executeable, Main.exe.

Open Makefile and replace our gcc, g77, and gpc commands with ones with path names that work on your system. Replace our references to libf2c.a and libgpc.a with ones that will work on your system.

Type *make* and see if the compile works. It probably won't. Make sure you have the *make* utility installed. Work on your library names until it does work.

Once *make* works without error, type *make clean* to get rid of the objects in anticipation of our turorial, which begins immediately below. We will show you how to call routines written in C, Fortran, and Pascal from a C main program. If you want your main program to be Fortran or Pascal instead, you will have to solve some additional problems, but they are not arduous.

Names

There are two things you have to get right when you are calling an external routine. One is its name, and the other is the size and format of its arguments. Each compiler makes it easy for you to match names and arguments with those of routines declared or used in source code that it also compiles. So it's easy to link a routine declared in one C-program with a routine declared in another C-program. But when you go between programming languages, you have understand how each compiler handles routine names, so you can match up the names, and you have to know how each compiler assigns memory to variable types.

Let's start with getting the names right. Enter the following command at your UNIX, Darwin, Linux, or Cygwin prompt.

```
gcc Main.c
```

You will see a list of external routines that the linker cannot find. The names in this list are the actual names the linker expects to find in an object file. You may think it's redundant for me to say "actual names", but it's not. Each programming language has a way for you to declare routines you would like to export to other objects, and routines you would like to import from other objects. In each such declaration, you type in a name for the routine. But the compiler does not use the name you type in. Depending upon the compiler, it might convert all letters to lower-case, or add underscores before and after the routine name. In other words, the *source name* differs from the *object name*. The *object name* is what is commonly referred to as the *symbolic name* of the routine, but we are going to use the source/object terminology, to make sure that there is no confusion in our discussion.

For example, look at C_Library.c. It contains the declaration of a routine called C_Sqr. The routine's source name is C_Sqr. The following command generates C_Library.o from C_Library.c. Try it, so we can see what C_Sqr's object name will be.

```
gcc -c -o C_Library.o C_Library.c
```

We now use the *nm* command to get a list of all the routines declared in C_Library.o.

```
nm C_Library.o
```

The output for our initial version of C_Library.o is:

```
00000000 T _C_Sqr
```

So the object name of C_Sqr is _C_Sqr. The gcc compiler preserved the case of the letters in the routine name, but added an underscore to the beginning of the name. In the program below, we declare C_Sqr as an external routine, like so:

```
extern double C_Sqr(double x);
```

When we compile this declaration with gcc, the resulting object contains an undefined routine named _C_Sqr. Whatever transformation gcc applies to source names in order to generate object names, it applies equally to the given names in routine declarations and external declarations, so the C- programmer never has to worry about how the object names differ from the source names.

But each compiler has its own rules for transforming source names into object names. In Fortran_Library.f, you will find a subroutine with the name F_Sqr. Compile this with your Fortran compiler. On our MacOS machine we type the following.

```
/usr/local/bin/g77 -c -o Fortran_Library.o Fortran_Library.f
```

Now see what's inside the object file.

```
nm Fortran_Library.o
```

You will see one line like this:

```
00000000 T _f_sqr__
```

The Fortran compiler converts all letters in a routine name to lower-case. It adds an underscore to the beginning of the name, and two to the end. Before we conclude that Fortran always adds two underscores to the end of a name, look at the routine called NoUnderscores in Fortran_Library.f. Its object name is _nounderscores_. Meanwhile, Fortran routine F_hello has object name _f_hello__. Any time we put one or more underscores in a source name, its object name has two underscores at the end. If there are no underscores, the object name has only one underscore at the end. Perhaps there are more rules the Fortran compiler uses to generate its object names. Our policy is to resolve linker errors by inspecting object names with *nm*.

So, how do we call F_Sqr from C? The object names must match, but each compiler has its own transformation from source name to object name. This means the source names are not the same. If we declare an external routine called F_Sqr in C, its object names comes out as _F_Sqr. We want to work our way towards the Fortran object name, which is _f_sqr__. So we start by converting to lower case, and we add two underscores to the end.

```
extern void f_sqr__(double* x, double* y);
```

In Pascal_Library.p, there is a routine called P_Sqr, declared as follows.

```
function P_Sqr(x:longreal):longreal; attribute(name='P_Sqr');
```

In Pascal, you get the opportunity to specify the source name for the routine's external declaration. In the line above, this source name is the same as the Pascal routine name, just to avoid confusion. But the compiler still adds an underscore to the specified source name. Compile this into an object file and see what P_Sqr's object name turns out to be, using the following lines.

```
/usr/bin/gpc -c -o Pascal_Library.o Pascal_Library.p
nm Pascal_Library.o
```

You will see that P_Sqr appears as _P_Sqr. It appears that gpc and gcc share the same rules for transforming source names into object names.

The C and Fortran compilers take our routine names, add their underscores and adjust the letter case, and place use this modified name in the object code they create. But the Pascal and C++ compilers use a two-level namespace, where a string of letters and numbers identifying the source file are added to the routine name. The external name for the Pascal_Library.pas initialization routine is _p__M14_Pascal_library_init. The two-level namespace allows us to us the same routine name in different source files for local operations without giving rise to an error at link time.

When you call a routine in one C++ object from another C++ object, the C++ compiler sorts out the routine names for you. You might call an external routine *foo*, which is declared in another C++ object. The actual name of the routine in the object is *__Z9fooPKc*, but you don't know that, nor do you have to know. Similarly, in Pascal, you might call an external routine *foo* without knowing that it's actual name is *__p__M6_Foo*.

But when you call Pascal routines from C, C++, or Fortran, the non-Pascal compiler does not know how to construct the two-level name of a Pascal routine. That's why you have to use the *attribute (name='Foo')* command with the Pascal routines you want to export to a non-Pascal compiler. The *attribute* command forces the Pascal compiler to assign a specific name to the routine in the object code. Similarly, when calling a non-Pascal routine from Pascal, we use the *external name 'Foo'* command to force the Pascal compiler to refer to the external routine by a specific name in the object code.

In C++ there are two commands that perform the same function as Pascal's *attribute* and *external* name comands. Here is how you declare an external routine with the name Foo_Import, which will be referred to as _Foo_Import in the object code.

```
extern "C" int Foo_Import();
```

This command forces the C++ compiler to use the C-compiler's naming system. When you export a routine, you use the same command. You place a function prototype at the top of your file, and then define the function later.

```
extern "C" int Foo_Export();
```

The Foo_Export routine will be place in the compiled object code, and be given the name _Foo_Export.

Simple Arguments

Fortran functions and subroutines take only variable arguments, meaning they can be changed by the routine. The value of the argument is not pushed onto the stack. Instead, a pointer to a location in memory containing the value of the argument is pushes onto the stack. In Fortran, you can pass a constant to a subroutine, but the compiler makes a copy of the constant in memory and passes a pointer to this constant. For more discussion, see below. Your C or Pascal compiler will not know to do the same thing before passing a constant to a Fortran subroutine, so your C or Pascal code must do the same thing explicitly.

C	Pascal	Fortran
double	longreal	real*8
float	real	real*4
int	integer	integer
char	char	character

Table: Equivalent Argument and Function Types.

Array Arguments

In ANSI Pascal, a string is an array of characters preceeded by a byte giving its length. In GNU Pascal, the header before the characters in the string is longer. The strings can be longer. Pascal supports schema types that neither C nor Fortran support. Our solution to the problem of passing arrays is to assume we are working with an array consisting of consecutive variables of the same type in memory, and pass a pointer to the first element in the array as if it were a single instance of the type. So we pass a pointer to the first number in an array when we pass the array.

When we pass a character string to a Fortran routine, we have to pass its length as a separate argument. In the Fortran code, there is no sign of this length. The compiler adds an additional argument to the routine for each string, in the order the string is mentioned in the declaration. When you call such a Fortran routine from C or Pascal, you must declare the routine with one additional length argument at the end of the argument list for each string. This length argument must be a four-byte integer, and it is passed directly, not by reference. See our F_Strings routine as an example. We call it from Main.c.

```
subroutine F_strings (a_string,a_integer,b_string,a_real)
integer a_integer
real*4 a_real
character*(*) a_string, b_string
write(6,*) 'Fortran: F_string, ',a_string,', ',b_string
return
end;
```

We declare F_strings with the following line in C.

```
extern void f_strings__(char *a_string, int *a_integer,
```

```
    char *b_string, float *a_real,
    int len_a, int len_b);
```

C-programmers may think it's pretty crazy of Fortran to go pulling this kind of extra-argument trick upon us, but just look at the trouble C gives us when we call F_Strings from C:

```
char astr[40]="A LONG STRING OF CAPITALS";
char bstr[2]="X";
f_strings__(astr,&i,bstr,&w,strlen(astr),strlen(bstr));
```

Even though we are passing a pointer to a string to F_Strings, we have to name the string *without* a pointer symbol (&). If we put pointer symbols in front of *astr* and *bstr*, we get compiler warnings, and the code does not work. Nevertheless, Fortran expects a pointer to the string, so our C call to the procedure must be passing a pointer to each string. If we want to prevent F_Strings from changing astr or bstr, you will have to restrict F_Strings with a *parameter* statement (see below), which serves the same purpose as C's *const* directive.

The ANSI C standard says that C will pass all arguments longer than four bytes by passing a pointer to the variable rather than a copy of the variable. But notice that our *bstr* is only two characters long, plus a null character, that makes three characters. The C compiler must have assigned more space to *bstr* and then passed a pointer automatically. Because C does not give us direct control over the manner in which it passes parameters, you will have to fool around with & symbols until your call works. But you will get there in the end.

Pascal can, in theory, present us with similar problems. If you pass a large argument directly, the compiler can copy the argument into the stack, and pass a pointer to the stack copy through a microprocessor register.

If we stick to our policy of a passing an array as a pointer to the first element, we can call F_Strings in a more reliable way in both C and Pascal. In C, the call to F_Strings would look like this:

```
f_strings__(&astr[0],&i,&bstr[0],&w,strlen(astr),strlen(bstr));
```

The above code works, it makes sense, and it's clear to everyone what is going on. No change in the compiler and the way it optimizes argument-passing is going to stop it from working.

Procedural Arguments

If you are a C or Pascal programmer, you may be wondering how Fortran can support recursive functions if it provides only variable arguments. The answer is that Fortran does not support recursive functions. It does, however, allow you to pass functions as arguments to other functions. In our Fortran Library we have the following.

```
subroutine F_call (aprocedure)
external aprocedure
integer i,j
i = 1
j = 2
write(6,*) 'Fortran: Calling aprocedure.'
call aprocedure(i,j)
return
end
```

The *aprocedure* argument is an externally-declared procedure. The compiler does not procedural type checking. When we call F_call from C (or Pascal), we must pass it a pointer to the first instruction of a void function (or procedure). In Main, we declare F_call as follows.

```
extern void f_call__(void aprocedure(int *i,int *j));
```

Meanwhile, in Pascal Library we have:

```
procedure P_Add(var i,j:integer); attribute(name='P_Add');
```

And we pass this procedure to the F_Call with the following line of C.

```
f_call__(P_Add);
```

void f()	procedure f()	subroutine ()
int f()	function f():integer	integer function ()

Table: Equivalent Argument and Function Types.

Wrapper Routines

Having figured out how to call a routine compiled in another language, we may find that the routine is still awkward to use. Fortran routines will not accept constants as arguments when called from C or Pascal, which means the C or Pascal programmer has to declare dummy variables in which to pass constants. When it comes to passing strings to Fortran routines, things start to get awkward. When it comes to receiving strings from Fortran routines, things get downright inconvenient. Our solution to these problems is to write *wrapper routines* for each of the original Fortran routines. The wrapper routine accepts arguments convenient to the native language, and performs all necessary translation for the foreign language before and after it calls the foreign routine.

We used wrapper routines to provide a convenient interface between the Minuit library of routines, which is compiled from Fortran, and both C and Pascal. You will find our C wrapper routines in Minuit.h, and our Pascal wrapper routines in Minuit.p. We show how the wrappers are used in Minuit_Main.c and Minuit_Main.p. If you want to try the example files yourself, you will need the Minuit library, libmyminuit.a. Download the Minuit source code and compile with Fortran 77.

Because the GCC Fortran compiler adds an underscore to its object names, we can give our wrapper routines the same names as they have in Fortran, and we know that their object names will not conflict with those of the compiled Fortran routines. Now the C or Pascal program can call the Fortran routines with lines of code that look almost identical to those in Fortran programs call the same routines. The only difference will be the presence of the occasional & or @ symbol to make pointers to local variables.

Global Variables

The Pascal Library contains a global variable declaration (P_GlobalVar). It is an integer. The C compiler knows P_GlobalVar is an integer, and not a function, because we declare it as an external like this:

```
extern int P_GlobalVar;
```

This differs from our declaration of a procedure that takes no arguments only in the absense of paranthases.

```
extern void P_ReadVar();
```

Demonstration

Compile the executable by entering *make* at your command prompt. Now run *Main.exe* with.

```
./Main.exe
```

You may not need the *./* on your platform, but we need it on Darwin (MacOS UNIX). The program runs, and shows some results. It stops in the Pascal library, asking you to enter a value for P_GlobalVar. It shows the results of P_Cube, which calls C_Sqr to calculate the cube of a P_GlobalVar. Although P_GlobalVar is an *int*, and P_Cube accepts a *double*, the C-compiler type casts it for the Pascal routine.

Fortran Arguments

We took the following from an e-mail sent to us by Peter Hurst. You will find the file he refers to, pass.f in the Examples folder of the Mixing collection. In the e-mail, Peter is explaining to us how to call the Fortran version of CERN's MINUIT fitting routine.

I see that you've got answers that you're happy with, but I'll still send this along. The argument passing thing is a very interesting question, and one that I haven't thought about in a long time. I think the answer has several parts. First, it is absolutely legal in FORTRAN to pass either a constant (like 5) or a variable (like i) into a subroutine. Second, the subroutine is allowed to change the arguments and pass them back (sorting subroutines, for example, typically recieve a jumbled array from some main routine, shuffle the contents of the array around, and simply pass the same array back nicely ordered). As

you've seen, there is potential trouble when you had a subroutine a constant and it tries to change the constant. I think this is expressly forbidden by the FORTRAN standard. What invariably happens when you try to do this is that the program comes to a crashing halt with some kind of segmentation fault. Exactly what the segmentation fault is and how it is signaled to the system depends on the compiler, and some amusing things are observable if one goes looking for them.

I grew up programming on various Digital VAX machines. I'm fairly sure that the VAX passed addresses only; if your code called a subroutine with a constant, the compiler would create a temporary variable, load the value of the constant into this temporary variable, and pass the address of the temporary variable into the subroutine. If the subroutine tried to change this variable the code would get angry and crash, but it wouldn't corrupt any memory. The compiler that comes with my current linux release behaves differently, and much more like a C compiler. (In fact, I suspect that the "FORTRAN compiler" is really just a FORTRAN-to-C translator followed by the C compiler.) In this case when you pass a constant (let's say '3') into a subroutine what you actually do is pass the address of the memory where the system itself stores the value of the constant. Now, if the subroutine tries to change this variable it would actually change the bit pattern that the system thinks represents '3', and much craziness could ensue. This can be demonstrated to happen, though the system eventually realizes its mistake and shuts things down.

I've included a scrap of code that contains a subroutine that swaps the values of three variables. The main program calls this subroutine twice, once by sending it variables and once by sending it constants. It works fine when variables are sent but crashes hard when it tries to swap constants. If one runs it in the debugger, though, one can actually see it change the system values of the constants 1, 2, and 3 before it gets wise (or realizes that it's confused) and crashes.

In practical terms I've always been able to get away with sloppy coding on this point and have come to no harm. The compiler will protect you against this sort of thing by crashing if you make a boo-boo. It's probably good form, though, to make a habit of only passing variables, particularly when using somebody else's code when you're not sure what they're doing.

It's also sometimes convenient to explicitly protect a variable from being changed. If, for example, you have a variable PI which you want to set to 837.5 and want to prevent well-meaning folk from trying to reset it to 3.1416 you can use the 'parameter' statement:

```
implicit none
real pi
parameter (pi = 837.5)
```

After this I think the code will crash if you try to reset pi in any way, shape, or form.

For the specific case of MINUIT's FCN function, I never call FCN myself and make MINUIT do all the work. I also take the extra precaution of transferring all the arguments passed into FCN by MINUIT into variables of my own and never alter the MINUIT values (except, of course, for 'fval').

Conclusion

Because the GCC compilers use the same linker ("back end"), it is easy to bring their source code compilers ("front ends") into harmony, provided that we know how to match the object names generated by each front end from the source names of variables and functions. It is straightforward to match the argument types by paying attention to how many bytes each variable takes up in memory. We don't have to worry about big-endian or little-endian confusion, because all the compilers use the same format on any given platform. Nor to we have to worry about the floating point formats.

We hope that this manual will make it possible for you to avoid re-writing Fortran or Pascal code for C users. We ourselves plan to call CERN's MINUIT minimization routines from C, and so compile ARAMyS as a shared library extension to our LWDAQ software. If you are a High Energy Physics user, you may be interested in the MINUIT source code, which you will find here.

We have not described how to make a Pascal or Fortran main program that links to C, Pascal, and Fortran code, but we believe this Manual will give you a head start. We know that it is possible in Pascal. Our LWDAQ software uses a shared library called *lwdaq.so*, which we compile for all platforms from our Pascal libraries, and which links to the TCL/TK libraries. The TCL/TK libraries are compiled from C. To take a look at these libraries, and our Makefile, download our LWDAQ software for your platform, and consult the Software Installation section of our LWDAQ Manual. You will find installation instructions for Linux, Windows, Unix, and MacOS. They tell you how to compile *lwdaq.so* for each platform. For more details on the use of DLLs (dynamic link libraries) on Windows, see Using Dynamic Link Libraries.