Creating Pascal bindings for C

v1.0

W Hunter

August 2016

Contents

1 Purpose of document

The purpose of this document is to show how to use C libraries from Object Pascal, more specifically using Free Pascal (FPC). It is inspired by [1], but showing a more up to date approach as per the *Free Pascal Programmer's Guide* http://www.freepascal.org/docs-html/prog/progse28.html, and with help from 'Thaddy' on the FPC forum.

I suggest you read everything in this document (that means the whole document), including the source files, because I've also put explanations and hints in them.

In all the examples that follow I'll first show the C code and how to compile it, and then the same for the Pascal code. Each example has three C source code files: a Header, a Module and finally the C program itself.[1]

If you're not familiar with C you might wonder why one would go through all the trouble of creating three separate files, especially for something as trivial as adding two numbers?

One reason is to keep the function definition separate from the actual C program, this way you can have many different C programs that can use the same functions (and possibly other things). If a function is defined in a C program and you wanted to use it in another program, you'd have to rewrite (or copy) that piece of code every time. But you probably know this.

Another reason, and relevant in the context of this document, is that C libraries generally take this form, namely, they consist of a Header file and the Module; the examples in this document tries to mimic that.

The approach is one of the following methods:

1. Compile the C module into an object file (*.o file) and link to the object file, or
2. Create a library (static) from the object files and then link to the library or
3. Create a library (shared) from the object files and then link to the library.

I'll show how to do all three methods.

2 Prerequisites

You need to have a fairly good knowledge of the C language, and of course of Object Pascal too. Where I refer to Pascal in this document, it also means Object Pascal, although in the context of this document it makes little or no difference.

If you spot an error or a typo or an improvement (I'm a hobbyist programmer), please let me know. You can mail me at whunter.za+pbc@NOSPAMgmail.com (delete the NOSPAM bit).

[1] If you want to read up about C programming, have a look at the *Deitel* books on C programming, I've also found *C for Linux Programming* (by Castro Lopo, Aitken and Jones) to be a good book, especially if you're going to be using *gcc*.

3 Development environments

3.1 Windows 7/8.1/10

You'll need the following two compilers:

- FPC (Free Pascal Compiler) – `http://www.freepascal.org/`
- GCC (MinGW GNU C Compiler) – `http://www.mingw.org/`

Setting up MinGW (Minimalist GNU for Windows) on Windows can be painful (to say the least). If on 64-bit, I suggest you download `mingw-w64-install.exe` instead of `mingw-get-setup.exe`, the latter didn't work for me on *Windows 10*.

3.2 Linux (Ubuntu)

All the code in this document was compiled and ran successfully in *Ubuntu 14.04 LTS 64-bit*, using FPC and GCC. The approach on Linux is a little different for Shared Objects (Dynamic Libraries) compared to Windows, in particular, you may have to do the following or a variant thereof:

The C modules must be compiled as "position-independent code" using gcc's 'fPIC' flag, for example:

```
gcc -c -fPIC -o sum.o sum.c
gcc -c -fPIC -o hello.o hello.c
```

Create a Shared Object

```
gcc -shared -fPIC sum.o hello.o -o liball.so
```

Set permissions (if required)

```
chmod 755 liball.so
```

Copy lib to a standard location

```
sudo cp liball.so /usr/local/lib/
```

Copy headers to a standard location

```
sudo cp sum.h /usr/local/include/ sudo cp hello.h /usr/local/include/
```

We need to copy the header files to the same location as the library so that function prototypes are available to someone who wants to link to the library, unless the prototypes are known otherwise.

Ask loader to update cache

```
sudo ldconfig
```

3.3 Compiler versions

The code examples in this document was last tested and worked on *Windows 10 64-bit*, using

- `Free Pascal Compiler version 3.0.0 [2015/11/16] for i386`
- `gcc (x86_64-win32-seh, Built by MinGW-W64 project) 6.1.0`

3.4 Conventions

Both Pascal programs and units uses the 'pas' file extension. If using the *Lazarus IDE* to compile your files, you may want to use 'lpr' for your Pascal source files (but apparently not for units).

Static Libraries uses the 'a' file extension (referring to the fact that they're merely archives).

Shared Objects (*.so) and Dynamic Libraries (*.dll) refer to the same thing, although I read somewhere that it's not a bad idea to keep the 'so' extension on Windows machines too if using *gcc*. That way users know that the library was probably compiled using *MinGW* tools.

You should know that Pascal code is not case sensitive, but C is, so keep this in mind when linking to external code, since your function prototypes have to be *identical*.

4 Linking to Object files

4.1 The sum of two numbers

A trivial example showing how to calculate the sum of two integer numbers and print the result to standard output.

4.1.1 C source files

Header file 'sum.h'

This really just contains the function prototype, the rest is a C "include guard" (see for example http://en.wikipedia.org/wiki/Include_guard)

```
/*
 * Filename: sum.h
 *
 */

#ifndef SUM_H
#define SUM_H

// Function prototype(s)
int sum(int x, int y);

#endif
```

Module 'sum.c'

This is where the actual function's workings are defined, the function needs to correspond to the prototype in the Header file.

```
/*
 * Filename: sum.c
 *
 */

#include "sum.h"

int sum(int x, int y)
{
    return x + y;
}
```

Program 'main1.c'

Note that in the program we only include the Header file, and then simply call the function. We don't actually need to know how the function is implemented, we just need to know what the function definition looks like (by inspecting the Header file) so that we can use it.

```c
/*
 * Filename: main1.c
 *
 */

#include <stdio.h>
#include "sum.h"

int main(void)
{
    int a = 17;
    int b = 19;
    printf("From C: The sum of %d and %d is %d\n", a, b, sum(a,b));

    return 0;
}
```

Compiling the C code

First create an object-code file from the C module; in a terminal (for example, `ConsoleZ` on `Windows`) type:

```
gcc -c sum.c
```

This should produce an object-code file `sum.o`. Now create a program called `main1.exe` by compiling and linking the object-code file and creating an executable:

```
gcc sum.o main1.c -o main1.exe
```

This should produce a file `main1.exe` (the 'exe' extension is only required on `Windows` machines). If you run the program (by typing its name in the terminal), you should get:

```
From C: The sum of 17 and 19 is 36
```

That just proves that the C code compiled and works as expected, which is obviously important.

4.1.2 Pascal source files

Two files are required, namely a Pascal unit and the Pascal program.

Unit 'unit1.pas'

We link to the C object file via the following Pascal unit.

```
unit unit1;

{$link sum.o} // link to C object-code, same as {$L sum.o}

interface
// leave this empty

uses ctypes; // need to specify this else FPC won't compile

function sum(x, y : cint32) : cint32; cdecl; external;
// function args and types to match the ones as defined in sum.h

implementation
// leave this empty

end.
```

Program 'prog1.pas'

```
program prog1;

{$mode objfpc}{$H+}

uses
  unit1;

var
  a, b: integer;

begin
  a := 12; // 17 in C program
  b := 51; // 19 in C program
  writeln('From Pascal: The sum of ', a, ' and ', b, ' is ', sum(a, b));
end.
```

NOTE:- Nowhere in the above Pascal code did we define how to add two integers (since that would be pointless as it's already defined in the C code).

Compiling the Pascal code

We can now call the "C generated" `sum.o` object file from Pascal code. As you can see from the Pascal code above, the `sum` function isn't defined anywhere **except** in the C source code. And that's the whole idea, the intention is to use the "C generated" object-file from Pascal and not having to define (write the code for) the function again because it's already been done in C.

Compile the Pascal program (output the file as `prog1.exe`, `fpc` will compile and link the files):

```
fpc prog1.pas -oprog1.exe
```

If you're on a 64-bit machine, you may have to type this instead (**this is true for the rest of the examples too**):

```
fpc -Px86_64 prog1.pas -oprog1.exe
```

This should produce a file `prog1.exe`. If you run the program (by typing its name in the terminal), you will get

```
From Pascal: The sum of 12 and 51 is 63.
```

NOTE:- Since we've passed different values (12 and 51) to the `sum` function in the Pascal code, we obviously expect a different answer.

4.2 Printing to screen

A slightly more complicated example showing how to print to standard output.

For this example to work I had to link to `libmsvcrt.a` in my MinGW directory (just search for it). I just copied the library file to the same directory as my source files.

4.2.1 C source files

Header file 'hello.h'

```
/*
 * Filename: hello.h
 *
 */

#ifndef HELLO_H
#define HELLO_H

void printhello(void);
void printhelloperson(char *);

#endif
```

Module 'hello.c'

```
/*
 * Filename: hello.c
 *
 */

#include <stdio.h>
#include "hello.h"

void printhello(void)
{
    printf("Hello, World!");
    printf("\n");
}

void printhelloperson(char *name)
{
    printf("Hello ");
    printf("%s\n", name);
}
```

Program 'main2.c'

```c
/*
 * Filename: main2.c
 *
 */

#include <stdio.h>
#include "hello.h"

int main(void)
{
    char *name = "John Smith"; // same as: char name[] = "John Smith";
    printf("From C:\n");
    printhello();
    printhelloperson(name);

    return 0;
}
```

Compiling the C code

As before, first create an object-code file from the C module; in a terminal type:

```
gcc -c hello.c
```

This should produce an object-code file `hello.o`. Now create a program called `main2.exe` by compiling and linking the object-code file and creating an executable:

```
gcc main2.c hello.o -o main2.exe
```

This should produce a file `main2.exe`. If you run the program (by typing its name in a terminal), you will get:

```
From C:
Hello, World!
Hello John Smith
```

4.2.2 Pascal source files

Unit 'unit2.pas'

We link to the C object file via the following Pascal unit.

```
unit unit2;

{$link hello.o} // link to C object-code

{$ifdef WINDOWS}
  {$linklib libmsvcrt}
{$else} // Linux
  {$linklib libc} // or try {$linklib c}
{$endif}

{Note the difference in use of $link and $linklib above,
the first is for object files, the latter for libraries.}

interface
// leave this empty

uses ctypes;

procedure printhello; cdecl; external;
procedure printhelloperson(name : pchar); cdecl; external;
// function prototypes to match that of hello.h exactly, even the case!

implementation
// leave this empty

end.
```

Program 'prog2.pas'

```
program prog2;

{$mode objfpc}{$H+}

uses
  unit2;

var
  name: pchar; // ^char will also work

begin
  name := 'Joe Public';
  writeln('From Pascal, calling C functions:');
  PrintHello;
  PrintHelloPerson(name);
end.
```

Compiling the Pascal code

Compile the Pascal program (output the file as `prog2.exe`):

```
fpc -Px86_64 prog2.pas -oprog2.exe
```

This should produce a file `prog2.exe`. If you run the program (by typing its name in the terminal), you will get

```
From Pascal, calling C functions:
Hello, World!
Hello Joe Public
```

4.3 Simple array functions

An example showing how to work with 1-d arrays (vectors)using **fun**ctions that operate on **arr**ays.

4.3.1 C source files

Header file 'arrfun.h'

```
/*
 * Filename: arrfun.h
 *
 */

#ifndef ARRFUN_H
#define ARRFUN_H

// Function prototype(s)

void multelem(double u[], double v[], double w[], int len_w);
// Multiply arguments element-wise; vectors must be same length

double sumelem(double v[], int len_v);
// Sum of elements

#endif
```

Module 'arrfun.c'

```
/*
 * Filename: arrfun.c
 *
 */

void multelem(double u[], double v[], double w[], int len_w)
// Multiply arguments element-wise; vectors must be same length
{
    int i;

    for (i = 0; i < len_w; i++)
    {
        w[i] = u[i] * v[i];
    }
}

double sumelem(double v[], int len_v)
// Sum of elements
{
    double ans = v[0];
    int i;

    for (i = 1; i < len_v; i++)
    {
  ans = ans + v[i];
    }

    return ans;
}
```

Program 'main3.c'

```c
/*
 * Filename: main3.c
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "arrfun.h"

int main(void)
{
    int i;

    // two vectors
    double a[] = {1,2,3,4,5};
    double b[] = {6,2,4,7,5};
    double c[] = {0,0,0,0,0};

    // determine length of vectors
    int veclen = sizeof(a) / sizeof(double);

    // multiply vectors element-wise
    multelem(a, b, c, veclen);

    // print result 1
    printf("From C: The multiplied elements are:\n");
    for (i = 0; i < veclen; i++)
    {
       printf("Item %d: %f\n", i, c[i]);
    }

    // print result 2
    printf("From C: The sum of the elements is:\n");
    printf("Sum: %f\n", sumelem(c, veclen));

    return 0;
}
```

Compiling the C code

As before, first create an object-code file from the C module; in a terminal type:

```
gcc -c arrfun.c
```

This should produce an object-code file `arrfun.o`. Now create a program called `main3.exe` by compiling and linking the object-code file and creating an executable:

```
gcc main3.c arrfun.o -o main3.exe
```

This should produce a file `main3.exe`. If you run the program (by typing its name in a terminal), you will get:

```
From C: The multiplied elements are:
Item 0: 6.000000
Item 1: 4.000000
Item 2: 12.000000
Item 3: 28.000000
Item 4: 25.000000
From C: The sum of the elements is:
Sum: 75.000000
```

4.3.2 Pascal source files

Unit 'unit3.pas'

We link to the C object file via the following Pascal unit.

```
unit unit3;

{$link arrfun.o}

interface
// leave this empty

uses ctypes;

procedure multelem(u, v, w:pcdouble; len_w:cint32); cdecl; external;
// multiply arguments element-wise; vectors must be same length

function sumelem(v:pcdouble; len_v:cint32): cdouble; cdecl; external;
// sum of elements

implementation
// leave this empty

end.
```

Program 'prog3.pas'

```
program prog3;

{$mode objfpc}{$H+}

uses
  unit3;

var
  a: array[0..4] of double = (1, 0.5, 1/3, 0.25, 0.2);
  b: array[0..4] of double = (5, 3.2, 8.5, 4.05, 1.4);
  c: array[0..4] of double = (0, 0, 0, 0, 0);
  veclen: integer = length(a);
  i: integer;
  ans : double;

begin
  writeln('From Pascal: The multiplied elements are:');
  multelem(a, b, c, veclen);
  for i := 0 to veclen - 1 do
    writeln('Item ', i:1, ': ', c[i]:1:2);

  writeln('');
  writeln('From Pascal: The sum of the elements is:');
  ans := sumelem(b, veclen);
  writeln('Sum :', ans:5:1);
end.
```

Compiling the Pascal code

Compile the Pascal program (output the file as `prog3.exe`):

```
fpc -Px86_64 prog3.pas -oprog3.exe
```

This should produce a file `prog3.exe`. If you run the program (by typing its name in the terminal), you will get

```
From Pascal: The multiplied elements are:
Item 0: 5.00
Item 1: 1.60
Item 2: 2.83
Item 3: 1.01
Item 4: 0.28

From Pascal: The sum of the elements is:
Sum : 22.2
```

5 Linking to Static and Shared Object C libraries

The code in this section calculates *Fibonacci* numbers using recursion, it also mixes in some array functions used in the previous section.

5.1 C source files for calculating Fibonacci numbers

Header file 'fibonacci.h'

```
/*
 * Filename: fibonacci.h
 *
 */

#ifndef FIBONACCI_H
#define FIBONACCI_H

/* Function prototypes */
int fib(int a);
void fibseries(int series[], int len);

#endif
```

Module 'fibonacci.c'

```
/*
 * Filename: fibonacci.c
 *
 */

int fib(int val)
// Recursive function to compute Fibonacci number, counting from 0
{
    if (val < 0) //  'Wrong' input will return -1
        return - 1;
    else if (val == 0)
        return 0;
    else if ((val == 1)||(val == 2))
        return 1;
    else
        return fib(val - 2) + fib(val - 1);
}

void fibseries(int series[], int len)
// Compute nth Fibonacci number for series of values
{
    int i;
    for (i = 0; i < len; i++)
    {
    series[i] = fib(series[i]);
    }
}
```

Start by creating object-code files, as before:

```
gcc -c fibonacci.c
```

You should end up with a ‘fibonacci.o’ object file.

5.2 Creating a Static C library (aka an archive)

In a terminal, type:

```
ar cr libfibonacci.a fibonacci.o
```

You should end up with a static library named `libfibonacci.a`. A static library is little more than an archive of object-code file(s).

5.2.1 Using the Static Library from Pascal

You can now call the functions in the ’libfibonacci.a’ library as follows:

Unit ’ulibfibStat.pas’

We link to the C object file via the following Pascal unit.

```
unit ulibfibStat;

{$linklib 'libfibonacci.a'} // link to our static library libfibonacci.a

interface
// leave this empty

// The function prototypes in fibonacci.h (the C header file) are:
// int fib(int a);
// void fibseries(int series[], int len);

// The equivalent Pascal functions are:
function fib(a:longint):longint; cdecl; external;
procedure fibseries(series:plongint; len:longint); cdecl; external;

implementation
// leave this empty

end.
```

Program 'fibStat.pas'

```
program fibStat;

{$mode objfpc}{$H+}

uses
  ulibfibStat;

var
  i: integer;
  vec: array[0..3] of integer = (3, 5, 7, 17);
  copyvec: array[0..3] of integer = (3, 5, 7, 17);
  len: integer = length(vec);

begin
  WriteLn('From Pascal, referencing the C Static Library:');

  for i := 0 to 17 do
    WriteLn('Fib(', i, ') = ', fib(i));

  fibseries(vec, len);
  WriteLn();
  WriteLn('Calculate Fibonacci numbers for the following vector of values:');
  Write('[ ');
  for i := 0 to (len - 1) do
    Write(copyvec[i], ' ');
  WriteLn(']');

  for i := 0 to (len - 1) do
    WriteLn('Fib(vec[', i, ']) = ', vec[i]);
end.
```

Compiling the Pascal code

Compile the Pascal program (output the file as `fibStat.exe`):

```
fpc -Px86_64 fibStat.pas -ofibStat.exe
```

This should produce a file `fibStat.exe`. If you run the program, you will get

```
From Pascal, referencing the C Static Library:
Fib(0) = 0
Fib(1) = 1
Fib(2) = 1
Fib(3) = 2
Fib(4) = 3
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(8) = 21
Fib(9) = 34
Fib(10) = 55
Fib(11) = 89
Fib(12) = 144
Fib(13) = 233
Fib(14) = 377
Fib(15) = 610
Fib(16) = 987
Fib(17) = 1597

Calculate Fibonacci numbers for the following vector of values:
[ 3 5 7 17 ]
Fib(vec[0]) = 2
Fib(vec[1]) = 5
Fib(vec[2]) = 13
Fib(vec[3]) = 1597
```

You can test that it's static by simply renaming (or deleting) the 'libfibonacci.a' file and running the program again. If it runs as before, it's proof.

5.3 Creating a Shared C library (aka a Shared Object file)

In a terminal, type:

```
gcc -shared fibonacci.o -o libfibonacci.so
```

You should end up with a Shared Object (dynamic lib) named `libfibonacci.so` that you can link to, just as before. For large programs you may notice that the final executable is smaller, because you've linked dynamically to the library, not statically, which means the library's code isn't included in the executable. For small programs you won't notice this.

5.3.1 Using the Shared Library from Pascal

You can now call the functions in the 'libfibonacci.so' library as follows:

Unit 'ulibfibDyn.pas'

We link to the C object file via the following Pascal unit.

```
unit ulibfibDyn;

interface
// leave this empty

// The function prototypes in fibonacci.h are:
// int fib(int a);
// void fibseries(int series[], int len);

// The equivalent Pascal functions are (and link to external shared object):
function fib(a:longint):longint; cdecl; external 'libfibonacci.so';
procedure fibseries(series:plongint; len:longint); cdecl; external 'libfibonacci.
    so';

implementation
// leave this empty

end.
```

Program 'fibDyn.pas'

```
program fibDyn;

{$mode objfpc}{$H+}

uses
  ulibfibDyn;

var
  i: integer;
  vec: array[0..3] of integer = (18, 15, 17, 23);
  copyvec: array[0..3] of integer = (18, 15, 17, 23);
  len: integer = length(vec);

begin
  WriteLn('From Pascal, referencing the Shared Object (Dynamic Library):');

  for i := 8 to 23 do
    WriteLn('Fib(', i, ') = ', fib(i));

  fibseries(vec, len);
  WriteLn();
  WriteLn('Calculate Fibonacci numbers for the following vector of values:');
  Write('[ ');
  for i := 0 to (len - 1) do
    Write(copyvec[i], ' ');
  WriteLn(']');

  for i := 0 to (len - 1) do
    WriteLn('Fib(vec[', i, ']) = ', vec[i]);
end.
```

Compiling the Pascal code

Compile the Pascal program (output the file as `fibDyn.exe`):

```
fpc -Px86_64 fibDyn.pas -ofibDyn.exe
```

This should produce a file `fibDyn.exe`. If you run the program, you should get

```
From Pascal, referencing the Shared Object (Dynamic Library):
Fib(8) = 21
Fib(9) = 34
Fib(10) = 55
Fib(11) = 89
Fib(12) = 144
Fib(13) = 233
Fib(14) = 377
Fib(15) = 610
Fib(16) = 987
Fib(17) = 1597
Fib(18) = 2584
Fib(19) = 4181
Fib(20) = 6765
Fib(21) = 10946
Fib(22) = 17711
Fib(23) = 28657

Calculate Fibonacci numbers for the following vector of values:
[ 18 15 17 23 ]
Fib(vec[0]) = 2584
Fib(vec[1]) = 610
Fib(vec[2]) = 1597
Fib(vec[3]) = 28657
```

You can test that it has linked dynamicly by simply renaming (or deleting) the 'libfibonacci.so' file and running the program again. If it doesn't run as before, it's proof, on Windows you should get a dialogue as shown below:

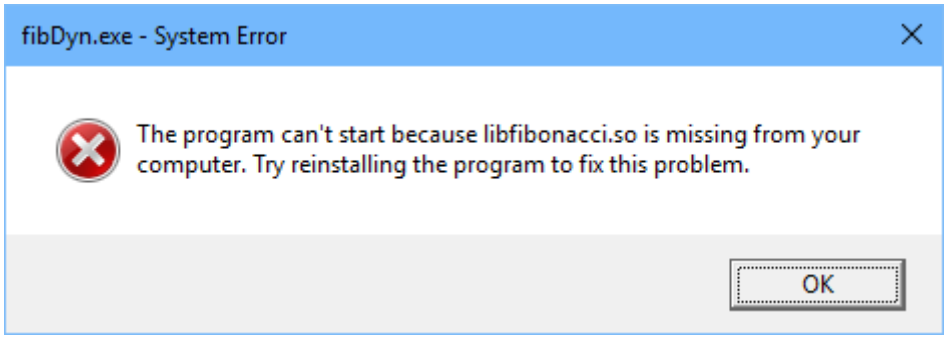

Figure 1: Windows error dialogue for missing library

References

[1] Marcou, Engler and Varnek. *How to use C code in Free Pascal projects*. University of Strasbourg, 23 July 2009.