How to use C code in Free Pascal projects

G. Marcou, E. Engler, A. Varnek

Université de Strasbourg,
Faculté de Chimie de Strasbourg,
4 rue Blaise Pascal,
67000 Strasbourg,
FRANCE

July 23, 2009

Contents

1 Introduction

Free Pascal provides a robust, portable and powerful compiler[5]. On the other hand, the C language benefits of a very long life time and therefore of millions of lines of codes. Hopefully, the Free Pascal compiler supports linking with objects and shared objects compiled in C. Reusing this material is therefore often desirable in Free Pascal projects.

Furthermore, the C++ provides also a very large number of very useful libraries. But it is not possible yet to access them from Free Pascal directly[5, 4]. Therefore, it is necessary to write in C code, procedures and functions to access the C++ ones. Linking Free Pascal to C code is therefore even more important since C will act as a "glue" between Free Pascal code and C++ -or other languages.

This article summarizes our experience about this field It will be extended, I hope, by ourself as we continue to gain experience with this problem, and by others. Here, we focus on GNU/Linux systems. For Windows users, we hope this will help too. But, certainly, the Delphi community will be of great help.

2 Dangers of mixing code

The support of other languages is not a characteristic of any language. It is a property of compilers. Therefore, instructions for doing so are dependent of the runtime environment. This includes the system and, sometimes, the compiler chosen. Characteristics of the definitions of the languages also plays a role [10].

In the following, the environment used is the Free Pascal Compiler (fpc 2.0.4) [5] and the GNU Compiler Collection (gcc 4.1.2) [7] on Linux.

Incompatibilities will play a role at nearly every stage of the -executable- binary generation. The less dangerous are those that will prevent compilation or linking. The most delicate ones will generate unpredictable behaviour. For example, it is up to the user to correctly convert from C or C++ types to Free Pascal. An error in this conversion will not necessarily generate compilation or linking errors, neither the executable will crash immediately. Such bug can become extremely difficult to spot.

Free Pascal compiler is meant to produce as straight as possible, binaries and executable code [5]. This leads to some differences compared to GCC compilers suite. The main difference is that Free Pascal users should not have to deal with a Makefile. Therefore, they should not have to compile the program by modules and then design a linking section.

This explains why the pre-compiler will have to deal with commands dedicated to the linker and it helps understand the logic with the `Unit` key word of Free Pascal.

3 Hello in C

This section is a tutorial showing, in a simple case, how it is possible to use C written objects in a Free Pascal project.

3.1 Hello object

The first thing to do is to produce an C written library to be compiled as a static object. Static objects, for the Linux world, are a legacy of the `a.out` binary format. Objects

are pieces of code already compiled that can be reused in other programs. Linking with a program which calls a function or procedure in an object file must be complete before execution time [3].

Here is proposed a small C code as a classical “hello” example. First a small header (file `chello.h`) defines what is the library composed of.

```
#ifndef CHELLO_H
#define CHELLO_H
#include <stdio.h>
void PrintHello();
void PrintHelloS(int);
#endif
```

This is not compulsory for C, but header files are highly encouraged since they simplify interface and helps linking the objects. Here are defined two procedures that write a “hello” message the second passing an integer to be written. The body of the program follows (file verb|chello.c|).

```
#include "chello.h"
void PrintHello(){
  printf("Hello\n");
  return;
};
void PrintHelloS(int nme){
  printf("Hello\n");
  printf("%i",nme);
  return;
};
```

This library is compiled using the command `gcc -c chello.c`. The result of the compilation is a binary file `chello.o`.

3.2 Free Pascal wrapping

In a C compatible world, it should be possible to pass the C header and let the compiler link the library definition of the procedures in the object files to the main program. Free Pascal does not understand C, so the header has to be rewritten in standard Free Pascal [5, 4]. This is the purpose of the following Free Pascal unit (file `helloU.pas`), wrapping the C written object in a standard Free Pascal code:

```
unit helloU;

{$link chello.o}
{$linklib c}

interface

uses CTypes;

procedure PrintHello; cdecl; external;
```

```
procedure PrintHelloS(nme : ctypes.cint32); cdecl; external;

implementation

end.
```

Several parts shall be noted. First there is only definition of the procedures. The implementation is empty. Even more, the definition is followed by the `external` keyword. This means the Free Pascal compiler should not wait for the body of the procedure since it shall find it at linking time [5, 4].

Second, the CTypes unit is called by the command `uses CTypes`. This unit is poorly documented, but its purpose is precisely to help for correct conversion between Free Pascal native types and C native types. In fact, it is quite difficult: the numerical limits of types like integers or floats are not defined by a language but by the compilers. As pointed by D. Mantione[1], with GCC, an `int` is $80000000..$7fffffff (32 bits), while a Pascal integer is defined as $8000..$7fff (16 bits). Thus, it would be required to use `longint` Free Pascal type to match it. Here we call the `ctypes.cint32` type converter. This method shall be preferred.

For type conversion of C types it is usefull to take a look at the C/C++ header `limits.h`. On Linux systems it is usually located in `/lib/include/`. For Free Pascal conversion, it shall be safer to take a look to the documentation [5, 4].

Then, there is two instructions for the linker. The first one `{$link chello.o}` means that the linker should link the program with the `chello.o` file [5, 4]. So the Free Pascal compiler will never know about what are truly those procedures, it will be the job of the linker to find them in the object file.

Doing so, a lot of references will appear to some functions typical of C. For example, it will find references to the `print` function and others defined in the `<stdio.h>` included. Those are defined in a shared library file, `libc.so`.

Shared libraries are libraries that are not linked at linking time. So the linking of the program is not complete. The program will not run until it will have found the definition of, at least, the standard C functions. In fact, at execution time, a special program, `ld-linux.so` is called to charge in memory the binary definition of the missing procedures [3].

The linker needs to know at compile time, where to find not yet defined functions and procedure. First, to be able to build in the program binary instructions relative to which shared library has to be charged in memory. Second to test that at least, at linking time, every single procedure and function will be defined.

Here we tell the linker to search in the file `libc.so` for missing definitions. It is the purpose of the instruction `{$link c}`.The linker will add a prefix `lib` and a suffix `.so`, in the Linux world to find the correct shared object file [5, 4].

It is usually necessary to use also the `{$link gcc}`to link with `libgcc.so` if the C objects were created with GCC. Nonetheless this library might not have been produced during GCC compilation and therefore can be missing. Therefore, linking errors might occurs if this linking statement is present as well as if this statement is missing.

One last thing, is that the Free Pascal definition has a different protocol as how to pass arguments in functions and procedures compared to C. This is what the keyword

[1] a regular contributor to Free Pascal

`cdecl` does. The `cdecl` warns the compiler and the linker that the arguments are to be understood as if they were in C[5, 4]. If this keyword is omitted it is possible that the program will compile and link but as soon as procedure will be called, it will crash. It might as well not compile any more if one of the wrapped procedures or function is called.

3.3 Free Pascal program

Now it is time to get a program that will call our C written functions. This is done in the file `hello.pas`:

```
uses
  helloU;

begin
   PrintHello;
end.
```

It is compiled using the command `fpc hello.pas`. This Free Pascal code only knows about the Free Pascal wrapper of the C object. So the compiler will compile first the `helloU` unit, then the main program. This part will produce `.ppu` and `.o` files. The first ones are Free Pascal binaries and the second ones, a translation in the `a.out` Linux binary format.

The linker is called that will link statically the `.o` it knows the existence of. It will then generate an executable binary with all informations about the shared libraries to be dynamically loaded in memory at execution time.

4 Hello in C++

This section is a tutorial showing how, in a simple case, it is possible to use C++ written objects in a Free Pascal project.

4.1 Hello object, again...

In this hello project in C++, a class hello is defined. This class contains a private structure and several public methods to access it or to perform some actions. Here is the header, in the file `hello.hpp`:

```
#ifndef HELLO_HPP
#define HELLO_HPP
#include<iostream>
#include<string>
using namespace std;
class hello{
public:
  hello(){
    string vide="";
    hello::SetName(vide);
  };
```

```
  hello(string &name){
    hello::SetName(name);
  };
  void HelloPrint();
  void SetName(string&);
  string GetName();
private:
  string name;
};
#endif
```

The private structure is a C++ string containing a name. There is a constructor to initialise this string to the null string and an overloaded constructor to initialise it to an arbitrary value.

The other public methods include setting or retrieving the value of the private structure. The last method just prints in the current console a friendly message.

The body of the methods defined in the header is in the following file, `hello.cpp`:

```
#include "hello.hpp"
void hello::HelloPrint(){
  cout << "Hello " << GetName() << endl;
  return;
};
void hello::SetName(string &str){
  hello::name=str;
  return;
};
string hello::GetName(){
  return(hello::name);
};
```

This set of file, the header and the body of the methods, is written in pure C++. It is compiled using the command `g++ -c hello.cpp`. The compilation provide an object file `hello.o` in which there is many references to C and C++ libraries, in particular standard template libraries -for the C++ string.

As stated before, Free Pascal cannot handle C++ objects [5, 4]. They have a very particular structure since they are linked to C++ shared libraries and here, have class structure. It is therefore necessary to build a C interface between this C++ code and Free Pascal.

4.2 C wrapping: first solution

As the C language does not know about C++ object oriented structure it is not possible to keep it. This means that the main job of the C wrapper will be to flatten the C++ class structure to a set of C functions [11].

Another important point is that C does not know about C++ standard template libraries and evolved structures like C++ strings, vectors, maps, etc. So another function of the wrapper will be to cast those C++ classes in standard C types [1].

To do this, it is useful to create a special header, as shown in the file `chello.h`:

```
#ifndef CHELLO_H
#define CHELLO_H
#ifdef __cplusplus
#include "hello.hpp"
#define EXPORTCALL __attribute__((stdcall))
typedef hello *helloHandle;
#else
typedef struct hello *helloHandle;
#define EXPORTCALL
#endif

#ifdef __cplusplus
extern "C"
{
#endif
  extern helloHandle EXPORTCALL NewHello();
  extern helloHandle EXPORTCALL NewHelloC(char*);
  extern void EXPORTCALL DeleteHello(helloHandle);
  extern void EXPORTCALL HelloPrint(helloHandle);
  extern void EXPORTCALL SetName(helloHandle, char*);
  extern const char* EXPORTCALL GetName(helloHandle);
  //
  extern helloHandle EXPORTCALL cNewHello();
  extern helloHandle EXPORTCALL cNewHelloC(char*);
  extern void EXPORTCALL cDeleteHello(helloHandle);
  extern void EXPORTCALL cHelloPrint(helloHandle);
  extern void EXPORTCALL cSetName(helloHandle, char*);
  extern const char* EXPORTCALL cGetName(helloHandle);
#ifdef __cplusplus
}
#endif
#endif/*CHELLO_H*/
```

This header can be interpreted both by a C and a C++ compiler. And two body files will be implemented, one in C++ code and the other one in standard C code.

There is several instruction to help to deal with this. The first one is to use the macro definition `#ifdef __cplusplus` and the corresponding `#endif`. It is true if the compiler analysing the code is a C++ compiler and false otherwise [10, 7]. This macro definition is widely spread among compilers but it is not part of any standard. Therefore, it might be necessary to define it manually through an option of the compiler.

This macro definition first protects the C++ header that needs to be included to define the hello object. If it is not protected, a C compiler will refuse the code since it would try to analyse the C++ code of the `hello.hpp` header.

The second statement that is protected so that only a C++ compiler can see it, is the definition of the `EXPORTCALL` to `__attribute__((stdcall))`. This value is interpreted by GCC and ensures the argument passing of the function to be compatible with C definition. This is why it is not set for the C compiler [7, 11].

The third protected statement only to be processed by the C++ compiler just defines a new type `helloHandle` as a pointer to the `hello` class.

For the C compiler, `helloHandle` will still be a pointer to a structure `hello` that is not yet defined. This definition is of course protected so that only the C compiler can see it. In such a way it does not interfere with the C++ instructions [11].

Another important difference of behaviour whether the header has to be processed by the C or the C++ compiler comes form the instruction `extern ''C''{}`. This is a C++ instruction and as such needs to be hidden for the C compiler. This instruction specifies a linking convention: how to access memory, integrated type formats, etc. This command does not target specifically the C language: it is also used to link Fortran to C++ for example [10].

In the following, it shall be noted that each method of the `hello` class is replaced by a function. To allow the function to call a method of the class for a given instance of the class, at least a pointer to this instance shall be passed. Constructors are an exception since they only receive this pointer.

The last unusual point of this header is the double definition of all the methods. It is so to cast C++ types in C standard types. In our example, the methods of the `hello` class uses C++ strings that cannot be understood by a C compiler. As the functions definitions has to be understood by both C and C++ compiler, we had to pass only C types as argument of the functions -a `char*`.

Functions written with a `c` prefix will be fully written in C. Therefore, they can only make calls to C functions. Functions that has not the prefix will have a body in a separate file, written in C++.

The C body of the functions is the file `chello.c`:

```
#include "chello.h"
helloHandle cNewHello(){
  return(NewHello());
};
helloHandle cNewHelloC(char* nme){
  return(NewHelloC(nme));
};
void cDeleteHello(helloHandle h){
  return(DeleteHello(h));
};
void cHelloPrint(helloHandle h){
  return(HelloPrint(h));
};
void cSetName(helloHandle h, char* nme){
  return(SetName(h,nme));
};
const char* cGetName(helloHandle h){
  return(GetName(h));
};
```

It can be compiled with the command `gcc -c chello.c -o chello_c.o`. The C++ body of the functions is in the file `chello.cc`:

```
#include "chello.h"
extern "C"{
helloHandle EXPORTCALL NewHello(){
```

```
  return new hello;
};
helloHandle EXPORTCALL NewHelloC(char *nme){
  string str(nme);
  return new hello(str);
};
void EXPORTCALL DeleteHello(helloHandle handle){
  delete handle;
};
void EXPORTCALL HelloPrint(helloHandle handle){
  handle->HelloPrint();
};
void EXPORTCALL SetName(helloHandle handle, char *nme){
  string str(nme);
  handle->SetName(str);
};
const char* EXPORTCALL GetName(helloHandle handle){
  return handle->GetName().c_str();
};
}
```

Here again the `extern "C"{}` statement specifies the linking convention [10]. The body of the functions is obviously C++: this convention cast C++ types and classes into standard C types. In this example, there are commands like `string str(nme)` or `string.c_str()` for `string` and `char*` interconvertions.

This code can be compiled using the command
`g++ -c chello.cc -o chello_cc.o`.

Following this procedure, it shall be noted that on contrary to what is sometime said [1], it is not necessary to compile the C wrapper with the C++ compiler. The final program can be linked using the two separate objects created.

4.3 C/C++ wrapping: second solution

The above described strategy is overly complicated. In fact, it is not necessary to define two times each flatten methods. To do so, the idea is to define wrapper function is a C compatible way and to write the body of the functions in C++.

First, the header `chello.h` is simplified:

```
#ifndef CHELLO_H
#define CHELLO_H
#ifdef __cplusplus
#include "hello.hpp"
#define EXPORTCALL __attribute__((stdcall))
typedef hello *helloHandle;
#else
typedef struct hello *helloHandle;
#define EXPORTCALL
#endif

#ifdef __cplusplus
```

```
extern "C"
{
#endif
  extern helloHandle EXPORTCALL NewHello();
  extern helloHandle EXPORTCALL NewHelloC(char*);
  extern void EXPORTCALL DeleteHello(helloHandle);
  extern void EXPORTCALL HelloPrint(helloHandle);
  extern void EXPORTCALL SetName(helloHandle, char*);
  extern const char* EXPORTCALL GetName(helloHandle);
#ifdef __cplusplus
}
#endif
#endif/*CHELLO_H*/
```

This time, each flattening function is defined once. Yet, the header is still written in a way such that it can be compiled by both the C++ and the C compiler. As above, the body `chello.cc` of the functions is written in C++. It is reproduced here:

```
#include "chello.h"
extern "C"{
helloHandle EXPORTCALL NewHello(){
  return new hello;
};
helloHandle EXPORTCALL NewHelloC(char *nme){
  string str(nme);
  return new hello(str);
};
void EXPORTCALL DeleteHello(helloHandle handle){
  delete handle;
};
void EXPORTCALL HelloPrint(helloHandle handle){
  handle->HelloPrint();
};
void EXPORTCALL SetName(helloHandle handle, char *nme){
  string str(nme);
  handle->SetName(str);
};
const char* EXPORTCALL GetName(helloHandle handle){
  return handle->GetName().c_str();
};
}
```

Again, this file is compiled using the command:
`g++ -c chello.cc -o chello_cc.o`.

As above, the goal is to use a C written interface to link Free Pascal code with flattened C++ methods. This is why the header alone will be compiled now. Because this file is a header -the name of the file ends by a `.h`- the default behaviour of the GCC compiler will change. It will not produce an object file but a precompiled header [7]. Yet, the C code in this header is enough to call the flattened C++ methods in a Free Pascal software. To produce an object file from the header, the compiler has to be

forced to compile the file as C source code using the `-x c` switch:
`gcc -x c -c chello.h -o chello_c.o`.

The following steps depend slightly of the above flattening strategies.

4.4 linking

The task of integrating those methods in Free Pascal is not over yet. Compilation of C++ objects is another compiler dependent task. In this tutorial, `gcc` is used as compiler. During compilation of this set of compiler, a shared version of the `libgcc` might not be compiled [7]. Therefore, it might not be possible to find such version to link the Free Pascal code with the C++ object. In brief, this task might be possible but can be more difficult than linking with pure C written objects.

Another solution is to link the C++ and the C objects in advance into a shared library. This can be done with the command
`gcc -shared -o libhello.so hello.o chello_cc.o`. Here, it is the `gcc` compiler tools that will find and set the correct dependencies of the libraries.

The shared library can be used as before in the Free Pascal Code.

4.5 Free Pascal wrapping

The Free Pascal wrapper is a Free Pascal unit defining the functions and procedures to be called in a Free Pascal project. If the first strategy was used, then, the Free Pascal wrapper should be:

```
unit helloFLAT;

interface

uses
sysutils;

type
   helloHandle = type pointer;

function cNewHello:helloHandle; cdecl;
function cNewHelloC(nme :PChar):helloHandle; cdecl;
procedure cDeleteHello(handle : helloHandle); cdecl;
procedure cHelloPrint(handle : helloHandle); cdecl;
procedure cSetName(handle : helloHandle;
                   nme : PChar); cdecl;
function cGetName(handle : helloHandle):PChar ; cdecl;

implementation

{$link chello_c.o}
{$linklib hello.so}
{$linklib c}
{$linklib stdc++}

function cNewHello:helloHandle; cdecl; external;
```

```
function cNewHelloC(
         nme :PChar):helloHandle; cdecl; external;
procedure cDeleteHello(
          handle : helloHandle); cdecl; external;
procedure cHelloPrint(
          handle : helloHandle); cdecl; external;
procedure cSetName(
          handle : helloHandle;
          nme : PChar); cdecl; external;
function cGetName(
         handle : helloHandle):PChar ; cdecl; external;

end.
```

Using the second strategy, functions call are done without a `c` prefix:

```
unit helloFLAT;

interface

uses
sysutils;

type
   helloHandle = type pointer;

function NewHello:helloHandle; cdecl;
function NewHelloC(nme :PChar):helloHandle; cdecl;
procedure DeleteHello(handle : helloHandle); cdecl;
procedure HelloPrint(handle : helloHandle); cdecl;
procedure SetName(handle : helloHandle;
                  nme : PChar); cdecl;
function GetName(handle : helloHandle):PChar ; cdecl;

implementation

{$link chello_c.o}
{$linklib hello.so}
{$linklib c}
{$linklib stdc++}

function NewHello:helloHandle; cdecl; external;
function NewHelloC(
         nme :PChar):helloHandle; cdecl; external;
procedure DeleteHello(
          handle : helloHandle); cdecl; external;
procedure HelloPrint(
          handle : helloHandle); cdecl; external;
procedure SetName(
```

```
          handle : helloHandle;
          nme : PChar); cdecl; external;
function GetName(
         handle : helloHandle):PChar ; cdecl; external;

end.
```

It shall be noted that the functions and procedures have to be declared with the `cdecl` key word. It ensures compatibility between C and Free Pascal declarations of functions and procedures arguments [5, 4]. Only the C functions are defined.

Those declarations need also a new type to be defined: `helloHandle`. Free Pascal does not really need to know about the true nature of this type: it will be linked to the C pointer [11]. As stated by D. Mantione, with this definition, `helloHandle` is not compatible to a pointer, but still is a pointer, which is exactly what we want. This definition makes `helloHandle` incompatible to a pointer because of the `=type` construction.

Implementation of the functions and procedures are referred as `external`. This means that the Free Pascal compiler knows it has to wait to find them: they will be defined only at linking time or later [5, 4].

The `cdecl` instruction guarantees the argument passing protocol to be compatible with C style. It should also be noted that to correctly map the C style `char*` type, the Free Pascal `PChar` type was used. Failing to map correctly the C type can be not detected at compilation or linking time, leading the generated binary to crash.

At last, as explained before, two sets of pre-processor instructions are defined. The first `{$link fhello_c.o}`,links the C definition and the Free Pascal definition of the procedures and functions [5, 4].

The second set of pre-processor instructions, `{$linklib hello.so}`, `{$linklib c}`,`{$linklib stdc++}`,pass instructions to the linker so that it can link the C functions with the C++ implementations of functions and classes [4, 5]. It also performs the link to the standard `libc` and `libc++` libraries.

4.6 Free Pascal program

An application resulting of this work can be found in the file `helloP.pas`:

```
uses
  helloFLAT;

var
   h : helloHandle;

begin
   h:=cNewHelloC('Gilles');
   cHelloPrint(h);
   cDeleteHello(h);
end.
```

It uses the `helloFLAT` unit. Thus it uses the newly defined type `helloHandle`. It will use it to construct an instance of the `hello` class for which the private string

structure is immediately set. It will use it to print a message and clean the memory before leaving.

During compilation, the linker might return an error saying that it cannot find `-lstdc++`. This is because depending on the version of the C++ compiler, several versions of this library might exist. They are named `libstdc++.so.x`, where `x` stands for the version number of the library; there might not exist a file without such version number. Therefore, a symbolic link named `libstdc++.so` shall be created to point to the desired version of the library. Ideally, this link shall be located in a directory searched by the linker.

5 A practical example: a Free Pascal unit to call GSL functions

The GNU Scientific Library (GSL) is a powerfull library written in C and readily usable in C++ code [6]. It includes complex numbers, special functions, permutations, linear algebra (based on BLAS), statistics, random distributions, interpolation, least square fitting, minimisation, root finding, differential equations, functional transformation (Fourier, Hankel, wavelets), and much more. This library is available for GNU/Linux and Windows system.

In this section, only the Legendre special functions will be discussed. Legendre polynomials are easily defined by the following relation:

$$P_0(x) = 1 \tag{1}$$

$$P_1(x) = x \tag{2}$$

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x) \tag{3}$$

They constitute an important family of orthogonal polynomial because they appear naturally in multipole expansions. They appear also in the definition of associated Legendre polynomial, spherical harmonics and many others which are included into the GSL and covered by the Legendre polynomial section of the library.

5.1 The headers

The Legendre polynomial functions are defined in the file `gsl_sf_legendre.h`. The following is an excerpt of this file:

```
#ifndef __GSL_SF_LEGENDRE_H__
#define __GSL_SF_LEGENDRE_H__

#include <gsl/gsl_sf_result.h>

#undef __BEGIN_DECLS
#undef __END_DECLS
#ifdef __cplusplus
# define __BEGIN_DECLS extern "C" {
# define __END_DECLS }
#else
```

```
# define __BEGIN_DECLS /* empty */
# define __END_DECLS /* empty */
#endif

__BEGIN_DECLS

int gsl_sf_legendre_Pl_e(const int l, const double x,
                         gsl_sf_result * result);
double gsl_sf_legendre_Pl(const int l, const double x);
int gsl_sf_legendre_Pl_array(
  const int lmax, const double x,
  double * result_array
  );
int gsl_sf_legendre_Pl_deriv_array(
  const int lmax, const double x,
  double * result_array,
  double * result_deriv_array
  );
int gsl_sf_legendre_P1_e(double x,
                         gsl_sf_result * result);
int gsl_sf_legendre_P2_e(double x,
                         gsl_sf_result * result);
int gsl_sf_legendre_P3_e(double x,
                         gsl_sf_result * result);
double gsl_sf_legendre_P1(const double x);
double gsl_sf_legendre_P2(const double x);
double gsl_sf_legendre_P3(const double x);
int gsl_sf_legendre_Q0_e(const double x,
                         gsl_sf_result * result);
double gsl_sf_legendre_Q0(const double x);
int gsl_sf_legendre_Q1_e(const double x,
                         gsl_sf_result * result);
double gsl_sf_legendre_Q1(const double x);
int gsl_sf_legendre_Ql_e(const int l, const double x,
                         gsl_sf_result * result);
double gsl_sf_legendre_Ql(const int l, const double x);
int gsl_sf_legendre_Plm_e(const int l, const int m,
                          const double x,
                          gsl_sf_result * result);
double gsl_sf_legendre_Plm(const int l, const int m,
                           const double x);
int gsl_sf_legendre_Plm_array(
  const int lmax, const int m, const double x,
  double * result_array
  );
int gsl_sf_legendre_Plm_deriv_array(
  const int lmax, const int m, const double x,
  double * result_array,
  double * result_deriv_array
  );
```

```
int gsl_sf_legendre_sphPlm_e(const int l, int m,
                             const double x,
                             gsl_sf_result * result);
double gsl_sf_legendre_sphPlm(const int l, const int m,
                              const double x);
int gsl_sf_legendre_sphPlm_array(
  const int lmax, int m, const double x,
  double * result_array
  );
int gsl_sf_legendre_sphPlm_deriv_array(
  const int lmax, const int m, const double x,
  double * result_array,
  double * result_deriv_array
  );
int gsl_sf_legendre_array_size(const int lmax,
                               const int m);
int gsl_sf_conicalP_half_e(const double lambda,
                           const double x,
                           gsl_sf_result * result);
double gsl_sf_conicalP_half(const double lambda,
                            const double x);
int gsl_sf_conicalP_mhalf_e(const double lambda,
                            const double x,
                            gsl_sf_result * result);
double gsl_sf_conicalP_mhalf(const double lambda,
                             const double x);
int gsl_sf_conicalP_0_e(const double lambda,
                        const double x,
                        gsl_sf_result * result);
double gsl_sf_conicalP_0(const double lambda,
                         const double x);
int gsl_sf_conicalP_1_e(const double lambda,
                        const double x,
                        gsl_sf_result * result);
double gsl_sf_conicalP_1(const double lambda,
                         const double x);
int gsl_sf_conicalP_sph_reg_e(const int l,
                              const double lambda,
                              const double x,
                              gsl_sf_result * result);
double gsl_sf_conicalP_sph_reg(const int l,
                               const double lambda,
                               const double x);
int gsl_sf_conicalP_cyl_reg_e(const int m,
                              const double lambda,
                              const double x,
                              gsl_sf_result * result);
double gsl_sf_conicalP_cyl_reg(const int m,
                               const double lambda,
                               const double x);
```

```
int gsl_sf_legendre_H3d_0_e(const double lambda,
                            const double eta,
                            gsl_sf_result * result);
double gsl_sf_legendre_H3d_0(const double lambda,
                             const double eta);
int gsl_sf_legendre_H3d_1_e(const double lambda,
                            const double eta,
                            gsl_sf_result * result);
double gsl_sf_legendre_H3d_1(const double lambda,
                             const double eta);
int gsl_sf_legendre_H3d_e(const int l, const double lambda,
                          const double eta,
                          gsl_sf_result * result);
double gsl_sf_legendre_H3d(const int l, const double lambda,
                           const double eta);
int gsl_sf_legendre_H3d_array(const int lmax,
                              const double lambda,
                              const double eta,
                              double * result_array);

#ifdef HAVE_INLINE
extern inline
int
gsl_sf_legendre_array_size(const int lmax, const int m)
{
  return lmax-m+1;
}
#endif /* HAVE_INLINE */


__END_DECLS

#endif /* __GSL_SF_LEGENDRE_H__ */
```

The beginning of the header is designed for C++ compatibility as described earlier. Then another header is included `gsl_sf_result.h`, which is required to define data structures that are usefull for all special functions in the GSL. It is reproduced hereafter:

```
#ifndef __GSL_SF_RESULT_H__
#define __GSL_SF_RESULT_H__

#undef __BEGIN_DECLS
#undef __END_DECLS
#ifdef __cplusplus
# define __BEGIN_DECLS extern "C" {
# define __END_DECLS }
#else
# define __BEGIN_DECLS /* empty */
```

```
# define __END_DECLS /* empty */
#endif

__BEGIN_DECLS

struct gsl_sf_result_struct {
  double val;
  double err;
};
typedef struct gsl_sf_result_struct gsl_sf_result;

#define GSL_SF_RESULT_SET(r,v,e) do { (r)->val=(v);
        (r)->err=(e); } while(0)

struct gsl_sf_result_e10_struct {
  double val;
  double err;
  int    e10;
};
typedef struct gsl_sf_result_e10_struct gsl_sf_result_e10;

int gsl_sf_result_smash_e(const gsl_sf_result_e10 * re,
                          gsl_sf_result * r);

__END_DECLS

#endif /* __GSL_SF_RESULT_H__ */
```

Fortunately, this header is short and does not call for any other include file.

Calling GSL functions in Free Pascal projects will therefore need two Free Pascal wrappers, one for `gsl_sf_result.h` and the other `gsl_sf_legendre.h`.

5.2 gsl_sf_result

First, the key words `__BEGIN_DECLS` and `__END_DECLS` are conventions keyword for C++ compilers only. They must be read as `extern ''C''{` and `}` respectively [8]. Therefore, It is not a concern for the development of the wrapper.

The main difficulty here, compared to the previous cases of the tutorial, is the presence of C structures. The Free Pascal unit wrapping C headers will have a systematic name composed of the original name of the header with the prefix `pas_`. The Free Pascal wrapper of the `gsl_sf_result.h` header is named `pas_gsl_sf_result.pas`:

```
unit pas_gsl_sf_result;

{$mode objfpc}{$H+}

interface

uses
```

```
  CTypes;

{$linklib gsl}
{$linklib gslcblas}
{$linklib m}

Type
{$IFDEF FPC}
{$PACKRECORDS C}
{$ENDIF}
    Ppas_gsl_sf_reslt = ^pas_gsl_sf_reslt;
    pas_gsl_sf_reslt = record
      val: ctypes.cdouble;
      err: ctypes.cdouble;
    end;
    Ppas_gsl_sf_reslt_e10 = ^pas_gsl_sf_reslt_e10;
    pas_gsl_sf_reslt_e10 = record
      val: ctypes.cdouble;
      err: ctypes.cdouble;
      e10: ctypes.cint32;
    end;

function gsl_sf_result_smash_e(re: Ppas_gsl_sf_reslt_e10;
    r: Ppas_gsl_sf_reslt):ctypes.cint32; cdecl; external;

implementation

end.
```

Note the usage of the `CType` unit. This unit is not documented in Free Pascal but proposes robust type conversion rules between C types and Free Pascal types. The required library to link with a Free Pascal project using the GSL are also present: `{$linklib gsl}`,`{$linklib gslcblas}`, and `{$linklib m}`. They are the GSL library of course and the CBLAS and the mathematical library. The CBLAS library can be replaced by better optimised libraries when efficiency becomes crucial -such as computer-intensive applications.

Then the header defines several structures such as `gsl_sf_result_struct`, which are then translated as types, such as `gsl_sf_result`. For the wrapper, the important point is to mimic the data structure. A solution is to use records (using the keyword `record`), which are native data structures of Free Pascal that looks like C structures. The problem is that the byte alignment of data in these structures can be different in Free Pascal and in C -it seems to be also compiler dependant. It affects the byte position of each element of the record and the total size of the record. The directive `{$PACKRECORDS C}` forces byte alignment to be similar to GCC compilers [5].

Since many functions might call for pointers to these structures, they are defined here too. For instance, the wrapper of the function `gsl_sf_result_smash_e` calls this function in the library and pass as input pointers to Free Pascal records that are, hopefully, perfectly mimicking the C structures. Any differences in the management of these memory space between the Free Pascal code and the C code might result in

uncontrolled behaviour of the software.

To resume, Free Pascal translation of a C structure can be a record if each element type of the record is compatible with each element type of the C record and if the byte packing is the same. When this cannot be achieved, a solution is to write another set of C functions for read/write access to such structures...

5.3 gsl_sf_legendre

Once the problem C structures is solved, the wrapper for Legrendre becomes straightforward.

```
unit pas_gsl_sf_legendre;

{$mode objfpc}{$H+}

interface

uses
  CTypes,pas_gsl_sf_result;

{$linklib gsl}
{$linklib gslcblas}
{$linklib m}


  function gsl_sf_legendre_Pl_e(l: cint32; x: cdouble;
     reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_Pl(l: cint32;
     x: cdouble): cdouble; cdecl; external;
  function gsl_sf_legendre_Pl_array(lmax: cint32;
     x: cdouble; reslt: pcdouble): cint32;cdecl;external;
  function gsl_sf_legendre_Pl_deriv_array(lmax: cint32;
     x: cdouble; result_array: pcdouble;
     result_deriv_array: pcdouble): cint32;cdecl;external;
  function gsl_sf_legendre_P1_e(x: cdouble;
     reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_P2_e(x: cdouble;
     reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_P3_e(x: cdouble;
     reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_P1(
     x: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_P2(
     x: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_P3(
     x: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_Q0_e(
     x: cdouble;
     reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_Q0(
```

```
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_legendre_Q1_e(x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_legendre_Q1(
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_legendre_Ql_e(l: cint32; x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_legendre_Ql(l: cint32;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_legendre_Plm_e(l: cint32; m: cint32;
   x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_legendre_Plm(l: cint32; m: cint32;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_legendre_Plm_array(lmax: cint32;
   m: cint32; x: cdouble;
   reslt_array: pcdouble): cint32;cdecl;external;
function gsl_sf_legendre_Plm_deriv_array(lmax: cint32;
   m: cint32; x: cdouble; reslt_array: pcdouble;
   reslt_deriv_array: pcdouble): cint32;cdecl;external;
function gsl_sf_legendre_sphPlm_e(l: cint32; m: cint32;
   x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_legendre_sphPlm(l: cint32; m: cint32;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_legendre_sphPlm_array(lmax: cint32;
   m: cint32; x: cdouble;
   reslt_array: pcdouble): cint32;cdecl;external;
function gsl_sf_legendre_sphPlm_deriv_array(lmax: cint32;
   m: cint32; x: cdouble; reslt_array: pcdouble;
   reslt_deriv_array: pcdouble): cint32;cdecl;external;
function gsl_sf_legendre_array_size(lmax: cint32;
   m: cint32): cint32;cdecl;external;
function gsl_sf_conicalP_half_e(lambda: cdouble;
   x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_conicalP_half(lambda: cdouble;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_conicalP_mhalf_e(lambda: cdouble;
   x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_conicalP_mhalf(lambda: cdouble;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_conicalP_0_e(lambda: cdouble;
   x: cdouble;
   reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
function gsl_sf_conicalP_0(lambda: cdouble;
   x: cdouble): cdouble;cdecl;external;
function gsl_sf_conicalP_1_e(lambda: cdouble;
   x: cdouble;
```

```
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_conicalP_1(lambda: cdouble;
    x: cdouble): cdouble;cdecl;external;
  function gsl_sf_conicalP_sph_reg_e(l: cint32;
    lambda: cdouble; x: cdouble;
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_conicalP_sph_reg(l: cint32;
    lambda: cdouble; x: cdouble): cdouble;cdecl;external;
  function gsl_sf_conicalP_cyl_reg_e(m: cint32;
    lambda: cdouble; x: cdouble;
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_conicalP_cyl_reg(m: cint32;
    lambda: cdouble; x: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_H3d_0_e(lambda: cdouble;
    eta: cdouble;
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_H3d_0(lambda: cdouble;
    eta: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_H3d_1_e(lambda: cdouble;
    eta: cdouble;
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_H3d_1(lambda: cdouble;
    eta: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_H3d_e(l: cint32;
    lambda: cdouble; eta: cdouble;
    reslt: Ppas_gsl_sf_reslt): cint32;cdecl;external;
  function gsl_sf_legendre_H3d(l: cint32;
    lambda: cdouble;
    eta: cdouble): cdouble;cdecl;external;
  function gsl_sf_legendre_H3d_array(lmax: cint32;
    lambda: cdouble; eta: cdouble;
    reslt_array: pcdouble): cint32;cdecl;external;

implementation

end.
```

Again, the GSL, CBLAS and mathematical libraries are called at link and execution time. The unit `pas_gsl_sf_result` is also called. Then the remaining calls to the C functions is systematic. Yet, attention must be paid to the variables types that are passed to the functions. They should match as close as possible those used by the C functions and the C structures must be replaced by their record counterpart.

6 Conclusion

This small tutorial will hopefully help new comers to Free Pascal to build applications that are able to take advantage of the huge libraries available in C and C++. This shall help saving time as first not having to write again what was already written in a quite

optimal manner. Besides, reusing code prevent the need of testing it. For instance, the Borland's Kylix project (that is by now, cold dead) was build on the TrollTech's QT libraries [2]. Those libraries were originally written in C for use in C++ code.

There exists some Free Pascal utilities developed for helping designing correct wrappers. This is precisely the goal of the utility `h2pas` [5, 4]. It is nonetheless essential to get knowledge of the details of the protocol to be able to use it correctly. Beside, it is wise to look at the Free Pascal code generated.

This tutorial is not complete. For instance, linking C++ library through a layer of C is still a rather complicated task. It might be possible to do better than what is proposed here. Also, there shall be a way to use the C++ class structure into Free Pascal without flattening it. It has been reported previously for Delphi7 and Windows [11]. Finally, this tutorial focus essentially on GNU/Linux systems. For Windows users, this might require the use of Cygwin [9]. Again, the Delphi community had to deal with the same kind of problems. Such resources as [11] can provide very good tips. Some work also has to be done to check how exceptions can be handled through those mixed code.

7 Thanks

We want to thanks D. Mantione and F. Lombardi for their interest on this text. They have reviewed this text and added essential contributions to it.

References

[1] M. Cline. How to mix c and c++, 2006.

[2] O. Dahan and P. Toth, editors. *Delphi7 Studio*. Eyrolles, 2004.

[3] U. Drepper. How to write shared libraries, 2006.

[4] freepascal project, http://community.freepascal.org. *freepascal community forum*, 2006.

[5] freepascal project, http://community.freepascal.org. *freepascal programmers guide*, 2006.

[6] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. B. Gough, 2nd edition version 1.8 edition, 2006.

[7] GNU is Not Unix. Using the gnu compiler collection (gcc), 2006.

[8] G. Lehey. *Porting UNIX Software: From Download to Debug*. G. Lehey, 2005. http://www.lemis.com/grog/Documentation/PUS/.

[9] Red Hat Cygwin Product, http://www.cygwin.com/. *Cygwin*.

[10] B. Stroustrup, editor. *The C++ programming language, special edition*. Pearson Education, 2003.

[11] R. Velthuis. Using c++ objects in delphi, 2006.