

How to use C code in Free Pascal projects

G. Marcou, E. Engler, A. Varnek

October 26, 2006

1

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 summerizes 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.

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 [7].

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) [6].

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 behavior. For example, it is up to the user to correctly convert from C or C++ types to Free Pascal. An error in this conversion

¹Université Louis Pasteur,
Institut de Chimie de Strasbourg,
4 rue Blaise Pascal,
67000 Strasbourg,
FRANCE

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 `verblchello.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. Mantione², 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 useful 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].

²a regular contributor to Free Pascal

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 `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 initialize this string to the null string and an overloaded constructor to initialize 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.cc`. 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

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 [8].

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 FHELLO_H
#define FHELLO_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 /*FHELLO_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 analyzing the code is a C++ compiler and false otherwise [7, 6]. 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 analyze 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 [6, 8].

The third protected statement only to be processed by the C++ compiler just defines a new type `helloHandle` as a pointer 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 only to be seen only by the C compiler so that it does not interfere with the C++ instructions [8].

Another important difference of behavior 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 [7].

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 C. 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 `fhello.c`:

```
#include "fhello.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 fhello.c -o fhello_c.o`. The C++ body of the functions is in the file `fhello.cc`:

```
#include "fhello.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);
};
};

```

Here again the `extern "C" {}` statement specifies the linking convention [7]. The body of the functions is obviously C++ so to cast C++ types and classes into standard C types. In this example, there is commands like `string str(nme)` while `nme` is a `char*`.

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

Following this procedure, it shall be noted that on contrary of 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 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 [6]. 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 the one of 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 fhello_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.

4.4 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.

```

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

{$link fhello_c.o}
{$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.

```

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 [8]. 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.

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}`, link the C definition and the Free Pascal definition of the procedures and functions with a `c` prefix [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 class [4, 5]. It also performs the link to the standard `libc` and `libc++` libraries.

5 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.

5.1 Conclusion

This small tutorial will 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. In fact, 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.

This tutorial is not complete. For example, 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 [8].

Some work also has to be done to check how exceptions can be handled through those mixed code.

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] GNU is Not Unix. Using the gnu compiler collection (gcc), 2006.
- [7] B. Stroustrup, editor. *The C++ programming language, special edition*. Pearson Education, 2003.
- [8] R. Velthuis. Using c++ objects in delphi, 2006.