Calling Foreign Functions with Pharo

Esteban Lorenzano, Guillermo Polito and Stéphane Ducasse

May 14, 2017
master @ 528d1e0*

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Unified FFI

Foreign Function Interface (FFI) represents the way to call functions/procedures and data structures written in C.

In this document you will learn about the new FFI framework named Unified FFI, or shortly, uFFI.

We will present how to invoke functions written in C, access C struct, define callbacks and other frequent situations that you face when you want to interact with an external library. uFFI has been developed by E. Lorenzano.

1.1 Calling a simple external function

A Foreign Function Interface (FFI) is the mechanism used by a programming language to invoke or call functions written in other programming language, most commonly C. To illustrate what this means, and how uFFI achieves it, we will start with an example. Suppose that you want to know the amount of time the image has been running by calling the underlying OS function named `clock`. This function is part of the standard C library (`libc`). Its C declaration is:

```
clock_t clock(void);
```

For the sake of simplicity, let's consider that `clock`'s return type is a `unit` instead of `clock_t`. We will discuss about types, conversions and typedefs in the following sections. This would result in the following function signature.

```
uint clock (void)
```

To call `clock` from the image, we need to define a binding between a Smalltalk method and the corresponding function. FFI bindings are normal smalltalk

methods that use the `ffiCall:module:` message to specify which function they will call. Let's then define a new class `FFITutorial` and define in it a binding to the `clock` function:

```
Object subclass: #FFITutorial
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

FFITutorial class >> ticksSinceStart
  ^ self ffiCall: #( uint clock () ) module: 'libc.so.6'
```

Note that in this example we specify `libc` as it exists in linux systems. For this code to run in other platforms you need to replace e.g., the `'libc.so.6'` string by `'libc.dylib'` in Mac OS or `'msvcrt.dll'` in windows.

For Mac OS: [[[language=smalltalk FFITutorial class » ticksSinceStart ^ self ffiCall: #(uint clock ()) module: 'libc.dylib']]]

For Windows: [[[language=smalltalk FFITutorial class » ticksSinceStart ^ self ffiCall: #(uint clock ()) module: 'msvcrt.dll']]]

In the example above, we define a new empty class, and a method named `ticksSinceStart` on its class side. The method `ffiCall:module:` is provided by UnifiedFFI and defined in `Object`, so we can be use it either on the instance or the class-side.

> **Note** the message `#ffiCall:module:` cannot be used as a normal expression in a playground or in the middle of a method. It must only be sent by methods that define a callout, because it does some

context mangling behind the scenes.

To test if our code works, execute the method and print its result:

```
FFITutorial ticksSinceStart
```

If everything went ok, this expression will return the number of native clock ticks since the Pharo proces started.

1.2 Analyzing the FFI callout

To understand what happenned and how it worked, let us look at the binding definition again:

```
FFITutorial class >> ticksSinceStart
  ^ self ffiCall: #( uint clock () ) module: 'libc.so.6'
```

The binding we just did is also called an FFI callout, as it performs a call of a function in the *outside world* (the C world). The method `ffiCall:module:` is

the one in charge of making the callout. In other words, it first transforms (marshalls) all arguments from Smalltalk to C, pushes them to the C stack, performs a call to the external function, and finally converts (marshalls) the return value from C to Smalltalk. To do all this, `ffiCall:module:` uses the function description provided as argument. The first argument is the *signature* of the function we want to call, described with an array, and the second argument is the *module* or *library* where uFFI will look for it.

In our example, the signature is as follows: `#(uint clock ())`

- Its first element is the C return type, here `uint`.
- Its second is the name of the called function, here `clock`.
- Its third element is an array describing the function parameters if there's any.

Basically, if you strip down the outer #(), what is inside is a C function prototype, which is very close to normal C syntax. This is intentionally done so, that in most cases you can actually copy-and-paste a complete C function declaration, taken from header file or from documentation, and it is ready for use. The second argument, the module or library, is in our example `'libc.so.6'`, the name of the library where to find the function. The avid reader will notice that our binding is platform dependent, as it will only run from a linux machine. We will explore how to define bindings in a platform-independent in the following section.

Note is this next sentence necessary here?

Note that there exist other messages to define callouts that will be discussed later.

1.3 A note on marshalling

In the `clock` callout example above, we specified the return type as `uint` and not `SmallInteger`. You will see as you go on in the chapter that this is the same as well with function arguments. Types in bindings are all described in terms of C language. In general you don't have to worry too much about this, because uFFI knows how to map standard C values to Smalltalk objects and vice-versa. So, a callout is executed, the return value will be converted into a `SmallInteger`. You can also define new mappings from C types to Smalltalk objects. This is useful when wrapping libraries that define new C types as we will show later.

This conversion process for types from different languages is called *marshalling*. We will see more examples of automatic conversions (by automatic, we mean that they are already defined in uFFI) in this Chapter.

1.4 Modules and libraries

We saw before that a callout requires to specify a module or library. uFFI uses this information to look up the given function. In our previous example, we were looking up the `clock` function inside the standard C library, i.e., `libc.so.6` in my unix system.

We saw until now how modules in uFFI can be expressed by using a library name. However, this has portability issues as a library will not have the same name in different platforms, or not be located in the same place. uFFI solves this problem by allowing also library objects. A library object defines dinamically the real name of the library in the current platform. A library in uFFI is defined as a subclass of `FFILibrary` and defining the methods `macModuleName`, `unixModuleName` and `win32ModuleName`. uFFI will dispatch to the library to get the correct module name given the current platform.

Let's for example consider the library `LibC` already provided by uFFI:

```
FFILibrary subclass: #LibC
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'UnifiedFFI-Libraries'

LibC>>macModuleName
  ^ 'libc.dylib'

LibC>>unixModuleName
  ^ 'libc.so.6'

LibC>>win32ModuleName
  "While this is not a 'libc' properly, msvcrt has the functions we
    are defining here"
  ^ 'msvcrt.dll'
```

With this platform aware library implementation, we can re-implement our binding in a platform-independent fashion:

```
FFITutorial class >> ticksSinceStart
  ^ self ffiCall: #( uint clock () ) module: LibC
```

As you can see, it is very easy to express different paths and names, depending on different conditions. For example, here you can see how the Cairo library is defined for unix platforms:

```
CairoLibrary>>unixModuleName
    "On different flavors of linux the path to library may differ
    depending on OS distro or whether system is 32 or 64 bit."

    #(
      '/usr/lib/i386-linux-gnu/libcairo.so.2'
      '/usr/lib32/libcairo.so.2'
```

```
      '/usr/lib/libcairo.so.2')
    do: [ :path |
      path asFileReference exists ifTrue: [ ^ path ] ].

    self error: 'Cannot locate cairo library. Please check if it
    installed on your system'
```

This is very useful and a recommended way of dealing with libraries, better than hardcoding module names.

Finally, note that any class can also define a method `ffiModuleName` to provide a default module for bindings in that class. Callouts may not specify a module to use this default module.

```
FFITutorial class>>ffiLibraryName
  ^ LibC

FFITutorial class>>ticksSinceStart
  ^ self ffiCall: #( uint clock () )
```

> **Note** `ffiLibraryName` is a terrible name I forget to change. It should be just `ffiLibrary` (but now change is not so easy...)

1.5 Passing arguments to a function

The previous `clock` example was the one of the simplest possible. It executes a function without parameters and we got the result. Now let's look how we can call functions that take arguments.

1.6 Passing a method parameter

Let's start with a really simple function: `abs()`, which takes an integer and returns its absolute value.

C header.

```
int abs ( int n );
```

Smalltalk binding.

```
FFICExamples class>>abs: anInteger
  ^ self ffiCall: #( int abs (int anInteger) ) module: LibC
```

Compared to the previous example, we changed the name of the function and added an argument. When functions have arguments you have to specify two things for each of them:

1. their type and the object that you want to be sent to the C function. That is, in the arguments array, we put the type of the argument (`int` in this example), and

2. the *name of the Smalltalk variable* we pass as argument.

Here `anInteger` in `#(int abs (int anInteger)` means that the variable is bound to the `abs:` method parameter and will be converted to a C int, when executing a call.

This type-and-name pairs will be repeated, separated by comma for each argument, as we will show in the next examples.

Now you can try printing this:

```
FFICExamples abs: -42.
```

1.7 About arguments

In the callout code/binding declaration, we are expressing not one but *two* different aspects: the obvious one is the C function signature, the other one is the objects to pass as arguments to the C function when the method is invoked. In this second aspect there are many possibilities. In our example the argument of the C function is the method argument: `anInteger`. But it is not always necessary the case. You can also use some constants, and in that case it's not always necessary to specify the type of the argument. This is because FFI-NB automatically uses them as C 'int's: `nil` and `false` are converted to 0, and `true` to 1. Numbers are converted to their respective value, and can be positive and negative.

1.8 Passing literals

Imagine that we want to have a wrapper that always calls the `abs` function with the number -42. Then we directly define it as follows:

```
FFICExamples class>>absMinusFortyTwo
  ^ self ffiCall: #( int abs (-42) ) module: LibC
```

Note that we omitted the type of the argument and directly write `int abs (-45)` instead of writing `int abs (int -45)` since by default arguments are automatically converted to C int.

But, if the C function takes a float/double as argument for example, you must specify it in the signature:

```
FFICExamples class>>floor: aFloat
  ^ self ffiCall: #( double floor(double aFloat) ) module: LibC
```

Note Luc Examples with more complex literals: arrays, ...

1.9 Passing variables

Often some functions in C libraries take flags as arguments whose values are declared using #define in C headers. You can, of course take these constant values from header and put them into your callout. But it is preferable to use a symbolic names for constants, which is much less confusing than just bare numbers. To use a symbolic constant you can create an instance-variable, a class-variable or a variable in shared pool, and then use the variable name as an argument in your callout.

For example, imagine that we always pass a constant value to our function that is stored in a class variable of our class:

```
Object subclass: #FFICContantExamples
  ...
  classVariables: 'TheAnswer'
..
```

Then don't forget to initialize it properly:

```
FFICContantExamples class>>initialize
  TheAnswer := -42.
```

And finally, in the callout code, we can use it like following:

```
FFICContantExamples class>>absMinusFortyTwo
  ^ self ffiCall: #( int abs ( TheAnswer ) ) module: LibC
```

You can also pass `self` or any instance variable as arguments to a C call. Suppose you want to add the `abs` function binding to the class `SmallInteger` in a method named `absoluteValue`, so that we can execute `-50 absoluteValue`.

In that case we simply add the `absoluteValue` method to `SmallInteger`, and we directly pass `self` as illustrated below.

```
SmallInteger>>absoluteValue
  ^ self ffiCall: #( int abs (int self) ) module: LibC
```

It is is also possible to pass an instance variable, but we let you do it as an exercise :)

1.10 Passing strings

As you may know strings in C are sequences of characters terminated with a special character: `\0`. It is then interesting to see how FFI-NB deals with them since they are an important data structure in C. For this, we will call the very well known `strlen` function. This function requires a string as argument and returns its number of characters.

C header.

```
int strlen ( const char * str );
```

Smalltalk binding.

```
FFICExamples class>>stringLength: aString
  ^ self ffiCall: #( int strlen (String aString) ) module: LibC
```

1.11 Example analysis

You may have noticed that the callout description is not exactly the same as the C function header.

In the signature `#( int strlen (String aString) )` there are two differences with the C signature.

- The first difference is the const keyword of the argument. For those not used to C, that's only a modifier keyword that the compiler takes into account to make some static validations at compile time. It has no value when describing the signature for calling a function at runtime.
- The second difference, an important one, is the specification of the argument. It is declared as `String aString` instead of `char * aString`. With `String aString`, FFI-NB will automatically do the arguments conversion from Smalltalk strings to C strings (null terminated). Therefore it is important to use String and not `char *`. In the example, the string passed will be put in an external C `char` array and a null termination character will be added to it. Also, this array will be automatically released after the call ends. This automatic memory management is very useful but we can also control it as we will see later. Using `(String aString)` is equivalent to `(someString copyWith: (Character value:0)` as in `FFICExamples stringLength: (someString copyWith: (Character value:0)`. Conversely, FFI-NB will take the C result value of calling the C function and convert it to a proper Smalltalk Integer in this particular case.

1.12 Passing two strings

We will now call the `strcmp` function, which takes two strings as arguments and returns -1, 0 or 1 depending on the relationship between both strings.

C header

```
  int strcmp ( const char * str1, const char * str2 );
```

Smalltalk binding

```
FFICExamples class>>stringCompare: aString with: anotherString
  ^ self ffiCall: #( int strcmp (String aString, String
    anotherString) ) module: LibC
```

Notice that you can add arguments by appending them to the arguments array, using a comma to separate them. Also notice that you have to explicitly tell which object is going to be sent for each argument, as already told. In this case, aString is the first one and anotherString is the second one.

1.13 Getting return value from a function

Symmetrically to arguments, returned values are also marshalled, it means that C values are converted to Smalltalk objects.

We already saw that implicitly through multiple examples since the beginning of the chapter. For example in the `abs` example, the result is converted from an int to a SmallInteger. In the `floor` example, the result is converted from a double to a Float.

But FFI-NB can also convert types a bit more complex than atomic types, like String

```
FFICExamples>>#getEnv: aString
  ^ self ffiCall: #( String getenv (String string) ) module: LibC
```

There is a mapping defined for each atomic type. About a bit more complex objects (like external objects or structures), we will talk in following sections.

1.14 Returning "void *"

Take this call as an example:

```
FFICExamples class>>malloc: aNumber
  ^ self ffiCall: #( void * malloc ( int aNumber ) )
```

This is a special case of return: when there is a function who answers a *void *. In this case, since FFI-NB cannot know which kind of object it represents, it will answer an instance of `ExternalData` (we will see this in next section).

> **Note** Luc illustrate that when NULL (a pointer with value 0) is returned, it is automatically converted to nil

1.15 External address

External addresses (contained in the class `ExternalAddress`) is the way we represent any kind of data *outside* Smalltalk. That means data (pointers, structures, arrays) who are allocated in the heap.

An `ExternalAddress` can be:

- an allocation of memory (you can use `ExternalAddress class>>allocate:` or `ExternalAddress class>>gcallocate:`). Note that in

case of `#allocate:` you will need to `#free` the external address later (`#gcallocate:` does that work for you).

- the result of a function call (usually it will come as part of an `ExternalData`)

`ExternalData` represents an `ExternalAddress` with an C type associated. For example,

Both `ExternalAddress` and `ExternalData` can be used as arguments when declaring functions with pointers as parameters, for example:

```
LibC>>memCopy: src to: dest size: n
  ^ self ffiCall: #(void *memcpy(void *dest, const void *src, size_t
    n)
```

1.16 External objects

`FFIExternalObject` represents an object in the heap. An external object is a reference to any kind of data allocated in the heap mapped to a Smalltalk object.

This is confusing, so I will try to explain it better: When you allocate a region of memory in the heap, you get a pointer to that location, which does not represent anything. But often, frameworks will allocate structures, pointers, etc. which actually represents ”an object” (not in the same sense as a Smalltalk object, but can be interpreted like one). For example, to create a cairo surface, you can call this:

```
AthensCairoSurface class>>primImage: aFormat width: aWidth height:
    aHeight
  ^ self ffiCall: #(AthensCairoSurface * cairo_image_surface_create
    ( int aFormat,
  int aWidth, int aHeight) )
```

This will call the cairo function `cairo_image_surface_create` but instead answer an `ExternalAddress` it will create an instance of `AthensCairoSurface`, so you can treat the allocated pointer as an object.

Any class in the system can be an external object as long as either:

- it inherits from `FFIExternalObject`; or
- it implements in its **class side** the method `asExternalTypeOn:`.

You can check for implementors of `asExternalTypeOn:` for examples, but they usually looks like this one:

```
AthensCairoCanvas class>>asExternalTypeOn: generator
  "use handle ivar to hold my instance (cairo_t)"
  ^ FFIExternalObjectType objectClass: self
```

Note if you want to add the resource to an automatic free mechanism (to make GC frees also the external object), you need

to call `autoRelease` (in case of children from FFIExternalObject) or implement similar mechanism.

Note FFIExternalObject replaces NBExternalObject

1.17 How autoRelease works

Sending #autoRelease message of an object registers object for finalisation with a particular executor. Then behaviour is divided:

- A.1) for ExternalAddresses, it just registers in regular way, who will call `finalize` on GC
- A.2) finalize will just call a free assuming ExternalAddress was allocated (which is a malloc)
- B.1) for all FFIExternalReference, it will register for finalisation what `resourceData` answers (normally, the handle of the object)
- B.2) finalisation process will call the object class » finalizeResourceData: method, with the`resourceData` result as parameter
- B.3) each kind of external reference can decide how to free that data (by default is also just freeing).

An example of this is how AthensCairoSurface works.

Note add an example

1.18 Structures

The `FFIExternalStructure` object is used to manipulate C structures from Pharo. From the standard C library we can use the `div()` function as an example. Given a numerator and denominator, it returns a structure holding the quotient and remainder. In stdlib.h we find these definitions:

```
typedef struct
{
  int quot; /* Quotient.  */
  int rem;  /* Remainder.  */
} div_t;

div_t div (int __numer, int __denom)
```

Converting these to FFI definitions we get...

```
FFIExternalStructure subclass: #Div_t
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'UnifiedFFI-ExampleLibC'
```

```
Div_t class >> fieldsDesc
  "self rebuildFieldAccessors"
  ^ #(
    int quot;
    int rem;
    )
```

```
LibC class >> div_numer: numer denom: denom
  ^ self ffiCall: #( Div_t div( int numer, int denom ) ) module: LibC
```

So lets try it out.

```
LibC div_numer: 7 denom: 2
  "Div_t ( quot: 3  rem: 1)"
```

Now for another example, imagine you need to manipulate the following C structure from Pharo:

```
struct My_Structure {
  uint8 id;
  char * name;
  uint name_length;
}
```

The first step is to create a subclass of `FFIExternalStructure` that we name `MyStructure`:

```
FFIExternalStructure subclass: #MyStructure
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFIDemo'
```

Then, we need to describe the structure. This is done by overriding the `FFIExternalStructure class>>#fieldsDesc` method. The syntax used to do it is pretty close to the C syntax:

```
MyStructure class>>#fieldsDesc
  ^ #(
  uint8 id;
  char * struct_name;
  uint name_length;
  )
```

Once the structure is described, you can generate the accessors by evaluating `MyStructure rebuildFieldAccessors`. This will generate accessors and mutators on instance side of `MyStructure`.

Now we are going to make the use of this structure a bit easier. As you saw before, the structure holds a char pointer to represent its name (as a `String`) and it also holds the length of this `String` in `name_length` field.

Right now, you can use the structure like this:

```
myStruct := MyStructure externalNew. "Use #externalNew to create the
    object on the external heap."

"Create an external array of type char to hold the name."
newName := FFIExternalArray externalNewType: 'char' size: 3.
'foo' doWithIndex: [ :char :i |
  newName at: i put: char ].

"Set the name."
myStruct struct_name: newName.
myStruct name_length: newName size.

"Get the name."
structName := String
  newFrom: (FFIExternalArray
        fromHandle: myStruct struct_name getHandle type: 'char'
    size: myStruct name_length)
```

So to get the actual string, you need to take the char pointer and read `name_length` char. We are going to wrap this procedure in a new method:

```
MyStructure>>#structName
  ^ String
    newFrom: (FFIExternalArray
          fromHandle: self struct_name getHandle type: 'char' size:
    self name_length)
```

We want the same thing for the mutator. We do not want to matter with `name_length` when setting `struct_name`:

```
MyStructure>>#structName: aString
  | externalArray |
  externalArray := FFIExternalArray externalNewType: 'char' size:
    aString size.
  aString doWithIndex: [ :char :i |
    externalArray at: i put: char ].
  self struct_name: externalArray.
  self name_length: aString size.
```

With the two preceding methods added, you can use `MyStructure` the same way you use any other Pharo object:

```
"Of course you still need to use #externalNew!"
myStruct := MyStructure externalNew
  structName: 'foo';
  id: 42;
```

```
  yourself.

myStruct structName. "foo"
myStruct id. "42"

myStruct structName: 'bar'.
myStruct structName. "bar"
```

1.19 Arrays

1.20 Callbacks

1.21 Handles (Windows)

1.22 How does it works?

In a very simplified way, old NativeBoost was taking the first calls to a method and transforming it into machine code, then reexecuting the method who now jumps and executes the code stored in the method.

To simplify, we do **the same** but instead injecting machine code, we create bytecodes to translate correctly input and output parameters, then we add a call to the function.

Note todo here design explanation.

1.23 Non conventional casts

Casting in C is trivial. You can do something like this:

```
void *var = 0x42000000.
```

And you will be creating a pointer who points to the address 0x42000000. This kind of declarations are used in certain frameworks, notably some Windows libraries.

In Pharo this ”casting” is not so easy, and we need to declare this kind of variables as ExternalAddresses. We do this as this:

```
var := ExternalAddress fromAddress: 16r42000000.
```

1.24 Conclusion