Unified FFI - Calling Foreign Functions from Pharo

Guillermo Polito, Stéphane Ducasse, Pablo Tesone and Ted Brunzie

February 12, 2020

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

Contents

Illustrations

CHAPTER 1

Introduction / Preface

This booklet contains a guide to the unified FFI framework (uFFI) for the Pharo programming language. The aim of this booklet is not to be just an API reference, but to thoroughfully document uFFI in an ordered way for both beginner and advanced users. We present the different concepts of uFFI, from simple call-outs and marshalling up to how to memory is managed, all chapters including examples of code and/or pictures to illustrate those concepts.

You'll find the source code of this booklet stored in https://github.com/SquareBracketAssocia Booklet-uFFI, where it has a companion bug tracker and list of releases. As of this first edition (v1.0.0), we cover the uFFI framework as it exists in Pharo 8.0, released in january 2020. Future editions will update this booklet for future Pharo versions, and document new features as they appear. Do not hesitate to open an issue if you find a problem.

Enjoy your reading, Guille

1.1 What is FFI?

A Foreign Function Interface (FFI) is a programming language mechanism that allows software written in one language to use resources (functions and data structures) that were written in and compiled with a different language. These "foreign" resources typically take the form of shared libraries, such as DLL files in Windows (or '.so' files in Linux and Mac) and can include run-time services made available by your operating system. A good example is a driver library provided by a vendor of a computer peripheral, such as a printer or a network card.

Code sharing and reuse via an FFI is fundamentally different from source

code ”includes”, calls to IDE libraries, or messages sent to the base classes provided by your language’s development environment. In those cases, your compiler can link function calls and share structured data while making convenient assumptions about the object code interface (known as the ABI, or Application Binary Interface). This is familiar programming: We follow published APIs (Application Programming Interfaces) as we write our code, and the compiler takes care of the low-level connections for us transparently.

However, if the compiled resources we wish to link to were built with a different language (or even a different compiler of the same language), the bits and bytes may not line up properly when we push arguments, make calls, and retrieve results, due to the different standards that the other code’s compiler followed when the borrowed code was produced. In this case, more detailed bookkeeping must be followed by our language to make ABI translations and ensure that differences in issues such as data alignment, byte ordering, calling conventions, garbage collection, pointer references, and so forth are all accounted for. Without this, we risk not only getting incorrect results – we could crash our process (or even the operating system).

While in theory FFI interfaces can be defined to link any pair of differing languages, most languages are designed to interact with libraries written in the C language. This is because C has a long heritage and widespread use, and (importantly) C has a predictable, standardized way to compile functions and structures. So with Pharo also choosing C as an FFI target we gain two major advantages: first, by basing ourselves on such ”standard” formats, we can build tools that simplify interoperability with an extensive array of existing C libraries. Second, since nearly every other language is doing the same, we can simply join those mutual C-standard ABIs together to form a bridge between different systems – allowing us to program at the source level of a C API.

1.2 Pharo, FFI and uFFI

From time to time it happens that we need to access a feature that is not available in Pharo standard libraries, nor in a community package. In such times, we have the choice of implementing such feature from scratch in an entirely Pharo-written library, or to reuse some existing implementation in another language. Taking which one to do may rely in many criteria such as personal taste, the ability to cope with the effort of implementing a new library from scratch, or the maturity, documentation and community of the existing libraries doing the job. Without FFI, the option of reusing an external existing library would not exist, or it would be constrained to expert developers extending the execution interpreter with modules like virtual machine plugins.

This booklet shows the Unified FFI framework for Pharo, or uFFI for short. The Pharo uFFI is an API framework that eases communication between

Pharo code and code that follows a C-style interface, making it possible to easily interact with external C libraries. Given that 'calling functions' is the ultimate objective of someone using an FFI, you will see that uFFI also helps with several other concerns, such as finding C function interfaces, transforming data between the Pharo world and the C world, accessing C structures, defining C-to-Pharo callbacks, and others.

1.3 How to read this booklet

We recommend beginners to read the booklet in order. Chapter 2 is a good introduction to how to use uFFI, introducing the concepts of callout and marshalling. Chapter 3 dives into the details of marshalling and its rules, which may be a bit overwhelming at the beginning, so do not hesitate to skip parts of it and come back to it when you have acquired more experience. Chapter 4 explaines different complex types, and can be read on the needs of a particular data-type. Chapter 5 discusses how developers can enhance and organize their uFFI library bindings, so experience using uFFI is assumed. Chapter 6 discusses the issues of memory management between Pharo managed memory and C managed memory, important to avoid some issues in the long term.

We invite those users already knowing FFI and/or uFFI to jump around and read on the need.

1.4 Acknowledgments

Pharo uFFI has been originally developed by E. Lorenzano, and was based on many ideas on the NativeBoost framework by Igor Stasenko. uFFI has received many contributions over the years from Pharo's open source community. The main backend of uFFI up to this day is the FFI implementation of the OpenSmalltalk VM, with contributors such as Eliot Miranda. This booklet was mainly written by Guille Polito and Pablo Tesone, with extensive reviews and edits from Stéphane Ducasse and Ted Brunzie.

CHAPTER 2

Foreign Function Interface and Call-Outs

This chapter presents a fair introduction to uFFI by introducing function call-outs: calling out an external function. We start by defining a Pharo uFFI binding to a C function. This example will guide us to how uFFI manages to find and load libraries, and how it looks up functions in it. Finally, when executing the binding, the returned value should be transformed to a Pharo object. Such transformation is called marshalling.

In the second part of this chapter, we refactor the initial example to extract the library into a `FFILibrary` object. A library object can cope with platform independent library lookup and smarther library searches.

2.1 Calling a simple external function

To illustrate the purpose and usage of uFFI, we will start with an example. Suppose we 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 declaration in C is:

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

For the sake of simplicity, let's treat `clock()`'s return type as an unsigned, `uint`, instead of `clock_t`. (We will discuss types, conversions, and typedefs in subsequent chapters.) This results in the following C function declaration:

```
uint clock( void );
```

To call `clock()` from Pharo using uFFI, we need to define a binding between a Pharo method and the `clock()` function. uFFI bindings are classes and

methods that provide an object-oriented means of accessing C libraries, implementing all the glue required to join the Pharo world and the C world.

To write our first binding, let's start by defining a new class, `FFITutorial`. This class will act as a module and encapsulate not only the functions we want to call but also any state we would like to persist. To access the `clock()` function, we then define a method in our `FFITutorial` class using the `ffiCall:library:` message to specify the declaration of the C function and indicate where it is defined. We will technically refer to this binding as a *call-out*, since it *calls* a function in the *outside* world (the C world).

If our Pharo code is hosted on a Linux system, we define this class and (class-side) method like so:

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

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

where we have simply copied the C declaration into a Pharo Array literal, then added `'libc.so.6'` as a reference to the current version of its C shared library on our Linux host system. (We can find out which version we have by entering `ls -1 /lib/*/libc.so*` in a Linux terminal window.)

To define the same binding for other host platforms, we need to replace the `'libc.so.6'` string by e.g., `'libc.dylib'` if we're running on MacOS, or `'msvcrt.dll'` if we use Windows. The equivalent definitions for MacOS and Windows are then:

```
FFITutorial class >> ticksSinceStart [
  ^ self ffiCall: #( uint clock() ) library: 'libc.dylib'
]
```

and

```
FFITutorial class >> ticksSinceStart [
  ^ self ffiCall: #( uint clock() ) library: 'msvcrt.dll'
]
```

Finally, we can use our freshly-created binding in a Pharo playground by inspecting or printing the following expression:

```
FFITutorial ticksSinceStart
```

If everything works as expected, this expression will return the number of native clock ticks since our Pharo process was launched.

2.2 Analyzing the FFI Call-Out

The simple example we ran in the previous section illustrates several important uFFI concepts. For starters, let's look at the binding definition again:

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

This call-out binding, a Pharo method, is called `ticksSinceStart` and happens to be named differently than the C function we are calling. Indeed, uFFI does not impose any restrictions as far as how to call your external functions. This can come in handy for decoupling your methods from underlying C-level implementation details.

We invoke the C function using the Pharo method `ffiCall:library:`, which is defined by uFFI. We provide the message arguments it needs usually by just copying and pasting the target C function declaration inside a Pharo Array literal, then referencing the name of the library in which it's defined (which in general will depend on our host platform).

uFFI interprets the declaration and performs all the necessary work needed to make the call-out and return the result. In general,

- uFFI searches for the specified library in the host system,
- On finding it, loads the C library into memory,
- Indexes the specified function within the library,
- Transforms and pushes Pharo arguments (if any) onto the stack,
- Performs the call to the C function,
- And finally transforms the return value into a Pharo object.

To form the first argument in our example, we render our C declaration for `clock()` in Pharo as a literal string array, like so:

```
#( uint clock() )
```

The first element of our array is `uint`, which is the function return *type*. This is followed by the function name, `clock`. Following the function name, we embed another Pharo Array to list the formal arguments the C function expects, in order. In this case, `clock()` takes no arguments, so we must provide an empty Array.

Another way to think of the declaration argument is this: If we look past the outer `#( )` wrapper, what we see inside is our C function prototype, appearing very similar to normal C syntax. This convenience is possible due to the coincidental nature of Smalltalk syntax: our use of strings and array notation in Pharo nicely mirrors how we write a C function declaration. uFFI was intentionally designed to take advantage of this so that in most cases we can

simply copy-paste a C function declaration taken from a header file or documentation, wrap it in `#( )`, and it's ready for use!

Our `ffiCall:library:` message also needs a second argument (`'libc.so.6'` in our Linux example), which is the name of the library in our host that contains the function. In many cases we do not need to provide a full path to the file in our host system. However, it should already be apparent that our bindings can be platform dependent if the library we need is also platform dependent. We will explore how to define bindings in a platform-independent way in a following section.

2.3 Notes on Value Marshalling

To fully understand the previous example, we still need to explain how the C `uint` return value (a non-object; a cluster of bytes popped off the stack) gets transformed into a Pharo `SmallInteger` *object*. Remember, C does not understand objects and does not do us the favor of returning values as attributes encapsulated within an object. We must somehow create an appropriate type of Pharo object, then migrate the C return value to become *its* value. Our code then receives this Pharo object.

This process of converting values between different internal representations is called *marshalling*, and in most cases is managed automatically in Pharo by uFFI. For example, uFFI internally maps the following standard C values to Smalltalk objects:

- Types `int`, `uint`, `long`, `ulong` are marshalled into Pharo integers (small or long integers, depending on the platform architecture).
- Types `float` and `double` are marshalled into Pharo floats.

Correct marshalling (and *demarshalling*) of values is therefore crucial for correct behavior of the bindings, particularly because the C language is so closely tied to underlying machine architecture. And yet, C values are merely "naked" bits and bytes in registers and memory; they have no inherent context or meaning. Consequently, they can be interpreted in many different ways, including by the Pharo run-time engine. The correct interpretation, involving such issues as byte ordering, type size, alignment requirements, string length/termination, etc. must be knowable, known, and properly handled. An object can tell you what it is, but a string of bits is just a string of bits...

As an example, consider the C integer value 0x00000000 (four contiguous '0x00' bytes). This can be interpreted as the small integer zero, as the *false* object, or as a null pointer – all depending on the marshalling rule selected for the inferred type. This means that the developer coding the binding method needs to *carefully* and *correctly* describe the types of argument bindings so uFFI will then correctly interpret and transform those values. This is

programming at the ABI level (binary representations), so precision counts! You are working side-by-side with the compiler, and inattention to detail can lead to crashes (or strange behavior that can be difficult to diagnose).

In the following chapters we will explore the marshalling rules more in detail and see how they apply not only for return values but also for arguments. Moreover, we will learn how to define our own user-defined data types and type mappings, allowing us to customize and fine-tune the marshalling rules to fit our particular needs.

2.4 Libraries

We saw earlier that a call-out binding requires us to specify a library that uFFI uses to locate and load the desired function. In our previous example, we indicated to Pharo that the `clock()` function we need was inside the standard C library, namely, the file `libc.so.6`. However, this form of the library exists in Linux systems, but not in Windows.

So we could say that this solution is not portable enough: One of the hallmark qualities of Smalltalk is supposed to be platform independence. But if we want to load and run *this* code on a different host platform, we are faced with changing the library name to match the name on our new host system. Worse, the libraries we need will all too often not have the same name, nor be located in the same place on all platforms. Not only that, we would need to be sure we catch every instance of these kinds of dependencies when we perform this ”migration”. Ugh!

One way to overcome this issue would be to define a set of bindings, one per platform, and decide which one to call based on which platform we detect at run-time, as follows:

```
FFITutorial class >> ticksSinceStart [
  self platform isUnix
    ifTrue: [ ^ self ticksSinceStartUnix ].
  self platform isOSX
    ifTrue: [ ^ self ticksSinceStartOSX ].
  self platform isWindows
    ifTrue: [ ^ self ticksSinceStartWindows ].
  self error: 'Non-supported platform'
]

FFITutorial class >> ticksSinceStartUnix [
  ^ self ffiCall: #( uint clock() ) library: 'libc.so.6'
]

FFITutorial class >> ticksSinceStartOSX [
  ^ self ffiCall: #( uint clock() ) library: 'libc.dylib'
]
```

```
FFITutorial class >> ticksSinceStartWindows [
  ^ self ffiCall: #( uint clock() ) library: 'msvcrt.dll'
]
```

But this solution means our binding code (which is essentially the same in all cases) gets repeated three times, and any changes to the binding design will require changing all three binding methods. This may look simple enough for our `clock()` binding, but repeating the code of complex bindings is likely not an optimal solution...

uFFI solves this problem by allowing us to use *library objects* instead of plain strings like we did earlier. A library object represents a library as an instance of `FFILibrary`, abstracting away any platform dependencies. This library class defines methods `macModuleName`, `unixModuleName`, and `win32ModuleName`; uFFI internally selects the correct library name at run-time after sensing the host platform. Bonus: This selection is a *process*, not a literal (a string), so it can now include behavior, such as the ability to dynamically search through different directories on your host system to locate the correct version of a library, as we will see shortly.

So for our example, we can now define such a library, `MyLibC`, as follows (being careful to note that the methods are *instance side* overrides):

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

MyLibC >> unixModuleName [
  ^ 'libc.so.6'
]

MyLibC >> macModuleName [
  ^ 'libc.dylib'
]

MyLibC >> win32ModuleName [
  "While this is not a proper 'libc', MSVCRT has the functions we
    need here."
  ^ 'msvcrt.dll'
]
```

To use this improved technique, we modify our *original* binding method (in 2.2) to substitute our library object (as a class) in place of the library name string:

```
FFITutorial class >> ticksSinceStart [
  ^ self ffiCall: #( uint clock() ) library: MyLibC
]
```

This version will run on all three platform types, *and* do so without us having to repeat the same code multiple times.

2.5 Library Searching

The `macModuleName`, `unixModuleName`, and `win32ModuleName` methods allow us, as developers, to employ different strategies to search for libraries and functions, depending on our host platform. If these methods return a relative path, library searching starts in common/default library directories on the system, or adjacent to the virtual machine executable. If they return an absolute path, default system locations will not be searched; only the specified path will be. In either case, if the library is not found or cannot be loaded, an exception is raised.

For example, an alternative override for `unixModuleName` can limit the search for `libc` to load only from the `/usr/lib/` directory on the host this way:

```
MyLibC >> unixModuleName [
  ^ '/usr/bin/libc.so.6'
]
```

Moreover, we are not constrained to simply return a string containing a path. The use of a method allows us to define and follow complex search rules, potentially locating needed libraries dynamically.

To take a real-world example, let's consider where the Cairo graphics library installs its resources on Unix-type systems. Although they are generally compatible, different '*nix' distros have evolved in ways that occasionally led to divergence in their file system structure, the placement of operating system files, and where they prefer to install packages the user may add. This is especially true (for historical reasons) where structure was added to avoid mixing 32-bit and 64-bit libraries. (Unix pre-dates the 8-bit micro-computer age. It may be older than you are!)

In the example below, the Cairo library search method for Linux checks for the existence of the library in each of `/usr/lib/i386-linux-gnu`, `/usr/lib32`, and `/usr/lib`, and if found, returns the absolute path to that file:

```
CairoLibrary >> unixModuleName [
    "On different flavors of Linux, the path to the library may
    differ, depending on the distro and whether the 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 that it
    is installed on your system.'
]
```

2.6 Sharing Libraries Between Bindings

To finish this chapter, let's define a new binding to another function in the same C library: `time`. The `time` function receives a (potentially null) C pointer and returns the current calendar time (as seconds of epoch), which we will once again assume is a `uint` for simplicity.

We can define our new binding as follows, providing a `NULL` pointer as the function argument. (We leave it to the reader to look up the definition of the C function and determine the purpose of this argument.)

```
FFITutorial class >> time [
  ^ self ffiCall: #( uint time( NULL ) ) library: MyLibC
]
```

Our new binding references the `MyLibC` library we defined earlier, so the above structure couples that code to both our bindings. To avoid such undesirable coupling, we can choose to refactor the class reference into a single class method in our `FFITutorial` class that can be used instead in both bindings.

To continue our example,

```
FFITutorial class >> myLibrary [
  ^ MyLibC
]

FFITutorial class >> ticksSinceStart [
  ^ self ffiCall: #( uint clock() ) library: self myLibrary
]

FFITutorial class >> time [
  ^ self ffiCall: #( uint time( NULL ) ) library: self myLibrary
]
```

This strategy, however, is still not as neat as we would like it to be. Further refactoring could clean this up, but fortunately for us, uFFI provides the support we need for sharing library definitions between bindings.

Any class defining a binding also has the option of defining a default library by overriding the `ffiLibrary` class method. Doing so allows us to omit a library definition altogether in our call-out bindings. The library will be automatically referenced by uFFI via the default method definition.

Let's see how this further simplies things for us:

```
FFITutorial class >> ffiLibrary [
  ^ MyLibC
]

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

FFITutorial class >> time [
  ^ self ffiCall: #( uint time( NULL ) )
]
```

Of course, bindings defining a library explicitly will necessarily override this mechanism, so you still have the option of creating a class with bindings that mix both mechanisms in any manner you wish.

2.7 Conclusion

In this chapter we have seen the basics of writing our own uFFI call-outs. We declare an FFI binding to a C function by specifying the name of the function, its return type, its arguments, and the library the function belongs to. uFFI uses this information to load the library in memory, look up the function, demarshall our Pharo arguments to C types and push them, call the function, and marshall any C return values back into Pharo objects.

```
FFITutorial class >> time [
  ^ self ffiCall: #( uint time( NULL ) )
]
```

Since different platforms work differently, uFFI provides extensions to define a library as an object. Library objects define per-platform strategies to search for C libraries in the host file system. By specifying relative paths we let uFFI search for the library in a platform's standard locations, while absolute paths override such behavior. In addition, this mechanism allows developers to write bindings that can dynamically search for their libraries in multiple locations.

The next chapter covers function arguments of various types. Although we glossed over its details on purpose, the `time` binding described in the previous section has a literal `NULL` pointer argument. We will see how literal arguments, which may be of different flavors, are very convenient syntactic sugar for specifying default argument values. They are not, however, the only means of conveying data from Pharo to the C world, as we *can* also send Pharo objects. Of course, great care must be taken when sending objects into the C world, so we shall revisit marshalling in more detail next.

CHAPTER 3

Marshalling, Types and Function Arguments

In the last chapter we looked at the basics of FFI call-outs to define a Pharo uFFI binding to a C function. These first examples introduced the concepts of function lookup, library references, and marshalling of return values. However, the idea of marshalling is not specific to the transformation of return values from C to Pharo: it also involves the transformation of argument values from Pharo to C, and the sharing of values between the two environments.

This chapter presents marshalling in more detail, focusing on function arguments. Our first examples show the capability of uFFI to use literals as default argument values. We then advance to other basic data types, from `Strings` and `ByteArrays` all the way to C pointers and how to manipulate them within Pharo. This chapter finishes by presenting the different marshalling rules in uFFI for basic types, particularly how to manage platform-specific types.

3.1 A First Function Argument

Our previous `clock()` example was the one of the simplest uFFI bindings possible, as it does not require passing any arguments, and because we could easily tweak the binding to map the return value to an unsigned integer. To understand how to deal with functions that require arguments, let's consider the C `abs()` function, which receives an integer argument and returns its absolute value. As with `clock()`, `abs()` is another function in the standard C library, so we will continue using our `FFITutorial` class to create its bind-

ing. Its prototype is declared as follows:

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

Creating a binding for such a function requires that we provide an `int` argument in our binding – specifying both type and value. To ease the construction of argument bindings, uFFI will attempt to match any parameter names in our calling method to corresponding function parameters that have the same names. In other words, we can define our binding for `abs()` as a method with a single `n` keyword argument as follows (remembering that our `FFITutorial` class now conveniently provides uFFI with the correct C library):

```
FFITutorial class >> abs: n [
  ^ self ffiCall: #( int abs ( int n ) )
]
```

Creating this binding does not really add extra complexity to what we did in our previous examples. We create a simple method that uses the `ffiCall:` message, providing it an array argument enclosing a copy of the function's prototype, just as we did before. (And as with our earlier `FFITutorial` class methods, we're assuming the continued use of our `ffiLibrary` override to eliminate the need to add the `library:` keyword).

The part that's new here is the introduction of the `n` parameter and its C type, but no additional work is needed on our part to transform the Pharo object to a C value. Also notice that uFFI knows to distinguish the argument's declared *type* from its formal parameter *name*, while simultaneously matching up the name with our Pharo method's formal argument.

We might even rename the parameter to use a more Pharo-*ish* naming style, such as "`anInteger`":

```
FFITutorial class >> abs: anInteger [
  ^ self ffiCall: #( int abs ( int anInteger ) )
]
```

Finally, we can call this binding with a Pharo `SmallInteger` by entering the following in a playground:

```
FFITutorial abs: -42.
=> 42
```

3.2 Marshalling

As we saw regarding return values in Chapter 1, Pharo's uFFI also manages the transformation of function arguments for us. The syntax we used may give the impression that uFFI simply copies the Pharo integer value to the C stack; however, the marshalling rules that uFFI must follow are not that

straight-forward. Pharo and C use different internal representations for their data types, which must be modified in order to be exchanged, potentially in different ways (usually depending on the host platform).

To illustrate how marshalling works, let's consider the case of transforming a Pharo `SmallInteger` to a C `int` type, as in our example. Internally, for efficiency purposes, Pharo represents `SmallIntegers` directly in binary, rather than as an object pointed to in the heap. Therefore, to differentiate integers from object pointers, Pharo tags `SmallInteger` "pointers" with an extra bit to signify this special interpretation.

Consider the `SmallInteger 2`; this value is represented in binary as the number `2r10`, but is internally represented in Pharo as `2r101`, where the *least* significant bit (LSB) is shifted in as the added tag. Since all Pharo object pointers are at least 32-bit aligned, we're guaranteed that their least significant bit will always be zero. This makes a non-zero LSB a reliable indicator that we're dealing with a `SmallInteger` rather than a heap pointer. (This is also why Pharo `SmallIntegers` are "only" 31 bits in 32-bit images and 61 bits in 64-bit images).

This *representation mismatch* requires that the uFFI transform `SmallIntegers` to C `ints` (and vice-versa) as follows (using the values in our particular example):

- A Pharo `SmallInteger` value transformed to a C value needs to be logically shifted to the right, `2r101 >> 1`, transforming `2r101` to `2r10`.
- A C integer value (representable in 31/61 bits or less) transformed to a Pharo `SmallInteger` needs to be shifted to the left and incremented, `(2r10 << 1) + 1`, transforming `2r10` to `2r101`.

Each type-to-type transformation has its own particular rule that uFFI follows to ensure that correct representation is always maintained.

Pharo-to-C Demarshalling

When demarshalling Pharo objects to C values, uFFI decides the transformation rule to use depending on two pieces of information. First, it considers the concrete type of the Pharo object, *its class*. Second, it considers the C type defined in the function binding as the target transformation type. At run time, when the binding method is executed, uFFI reads the type of the Pharo argument and transforms the argument object into the indicated C type representation (if possible), performing a type cast/coercion as necessary.

These transformation rules can have consequences, which we illustrate with the following cases, using our previous `abs:` binding example:

- `SmallIntegers` in 32-bit Pharo images are transformed to `ints` as expected (since 31 bits cannot overflow into 32 bits):

```
FFITutorial abs: -42.
=> 42
```

```
FFITutorial abs: SmallInteger maxVal negated.
=> 1073741823
```

- `SmallIntegers` in 64-bit Pharo images *can* overflow the size of a `C` `int` (still 32 bits on most 64-bit hosts), and so are coerced by truncating their value to fit, producing results similar to what a `C` program would produce in a similar situation. Since the `maxVal` of a 64-bit Pharo `SmallInteger` is 60 '1' bits (plus a sign bit), it truncates to 32 '1' bits, which, to `C`, is the two's complement value `-1`. Hence,

```
FFITutorial abs: SmallInteger maxVal negated.
=> 1
```

- Pharo `LargeIntegers`, by contrast, are "infinite precision" (no `maxVal`), and do not have a corresponding `C` type to convert into. Consequently, uFFI throws an error:

```
FFITutorial abs: SmallInteger maxVal * -10.
=> Error: Could not coerce arguments
```

- Pharo floats, when provided for `C` `int` arguments, will be truncated (mathematically) to produce an integer:

```
FFITutorial abs: Float pi.
=> 3
```

- But if the Pharo float is too large for a `C` integer, strange values can result due to coercion to 32 bits:

```
FFITutorial abs: SmallInteger maxVal * Float pi.
=> -2147483648  "in 32-bit Pharo images"
```

```
FFITutorial abs: SmallInteger maxVal * Float pi.
=> 2007355392  "in 64-bit Pharo images"
```

- Pharo objects that are incompatible with `C` `int` type are rejected, and an exception is thrown:

```
FFITutorial abs: Object new.
=> Error: Could not coerce arguments
```

C-to-Pharo Marshalling

A similar-yet-different story happens when marshalling `C` values to Pharo objects. In this case uFFI decides the marshalling rule based on just the specified return type. At run time, when the binding method is executed and the `C` function returns, uFFI transforms the (expected) return value into the closest Pharo type corresponding to the declared `C` type.

For example, a delared `int` return type will cause uFFI to interpret the returned value as either a `SmallInteger` or `Large(Positive|Negative)Integer`, depending on the size and sign of the data; a C `float` or `double` type will interpret the returned data as a Pharo `Float`.

For example, our above evaluation of the following in a 32-bit Pharo image,

```
FFITutorial abs: SmallInteger maxVal * Float pi.
=> -2147483648
```

returns a Pharo `LargeNegativeInteger`, since the binary return value, 2^31, will not fit in a Pharo `SmallInteger`. It requires *32* bits, so uFFI selects the object type it *will* fit in, which is a `LargeNegativeInteger` object.

Marshalling of Incorrectly Declared Types

The marshalling rules we have seen above show that the way in which we specify function types is *crucial* to the correct behavior of our bindings, and thus our applications. In other words, call-out bindings require that C types are correctly specified, otherwise run-time errors – or even worse, viable but incorrect value transformations – may happen.

Let's consider as an example what happens if we create a companion `abs:` binding to operate on a `Float` argument instead of an integer, but which still uses the same C `abs()` function:

```
FFITutorial class >> floatAbs: aFloat [
  ^ self ffiCall: #( int abs ( float aFloat ) )
]
```

Now let's we evaluate this version in a playground using a negative `Float` value:

```
FFITutorial floatAbs: -1.0.
=> 1082130432  "in 32-bit Pharo images"
```

```
FFITutorial floatAbs: -1.0.
=> 0  "in 64-bit Pharo images"
```

Although we expected the message to evaluate to `1` (because the return value is still of type `int`), this example returns `0` (in 64-bit images). To understand this result, we need to realize that our bindings, and the way we express their C types, are *independent* of the actual function implementation we are calling. In other words, even if we 'set' the type of `abs()`'s argument to `float`, the `abs()` function in our system remains built and compiled to work only on C `int` values. We're not *compiling* the C functions in Pharo, only attaching and calling them. So we must strictly and carefully adhere to their documented function declarations.

What happens "under the hood" in this example is that uFFI transforms our `-1.0` Pharo float into a C `float`, then pushes it on the stack and calls the

`abs()` function. The function uses that value, but considers it to be an integer. And it happens that C integers and floats have the same bit size (32 bits), but vastly different representations in C. This produces either hilarious or *dangerous* results...

A similar problem arises if the return type of a function is incorrectly specified. Let's take for example a slightly modified version of our original `abs:` binding, this time declaring a C `float` return type:

```
FFITutorial class >> floatReturnAbs: anInteger [
  ^ self ffiCall: #( float abs ( int anInteger ) )
]
```

When this call returns, uFFI will interpret the returned value as a C float, and try to marshall it to a Pharo `Float`:

```
FFITutorial floatReturnAbs: -3.
=> Float nan  "in 32-bit Pharo images"
```

```
FFITutorial floatReturnAbs: -3.
=> -1.07374176e8  "in 64-bit Pharo images"
```

Since the implementation of `abs()` actually returns an `int`, the bits are wrongly interpreted, producing not an error (at least not in the 64-bit case), but a strange value – *one that your application might not detect*. And if this kind of misinterpretation is only "slightly off", it can lead to buggy behavior that is maddeningly difficult to diagnose.

So while writing library bindings with uFFI is fun and simple, the binding developer needs to make sure that the types are correctly declared, and that the correct version of the library is being used. Fortunately, for mature libraries, most of the time it is sufficient to simply "copy and paste the function declarations".

3.3 Function Argument Bindings

We have seen in the earlier introductory example how to use method parameters as arguments when writing function bindings. In this section we explore other ways to define arguments – in particular literal objects, instance variables, and class variables.

Literal Object Arguments

From time to time we will find ourselves calling C functions that require many more parameters than the ones we are actually interested in providing. For example, C functions may have extra parameters to select or control certain options and configurations, or they may have parameters that are only necessary in particular cases (and which are ignored in others).

Although parameters such as these are deemed *optional*, we cannot leave them out of our binding definition – they still need to be there for the C call to execute correctly. To make it easier to deal with such optional parameters, uFFI allows Pharo literal objects to be provided as function arguments.

To see how this works, let's first imagine that for some reason we needed to call the `abs()` function with the `-42` argument exclusively. Using what we have learned up to this point, a simple way to define such a binding would be to define a normal Pharo method calling our binding with the hard-coded value `-42`:

```
FFITutorial class >> absMinusFortyTwo [
  ^ self abs: -42
]
```

This approach is convenient when we want both `abs:` and `absMinusFortyTwo` to be exposed to our user. It certainly benefits from the binding being declared only once, allowing us to isolate potential marshalling mistakes in a single location. However, we may not want to couple to the `abs:` method directly, as we're doing in this case. Instead, we want to provide `-42` directly to the C `abs()` function, as a literal (canned-in) value.

To provide for this, uFFI supports the use of literal arguments in C function bindings. Rather than using a method parameter as argument in the function binding, we can specify a literal value following its C type declaration:

```
FFITutorial class >> absMinusFortyTwo [
  ^ self ffiCall: #( int abs ( int -42 ) )
]
```

Notice that we don't just type in a Pharo integer for the expected argument and expect uFFI to perform implicit type conversion. Even literal values such as integers must be preceded by a C type declaration (`int` in the above example). This type information is needed by uFFI to correctly determine how to transform the Pharo integer, given the many different forms in which it could be rendered.

Consider that in the case of C integers, a number can be signed or unsigned, and can occupy different sizes such as 8, 16, 32, or 64 bits. Pharo can't guess; the C function expects a specific type to be provided, and the Pharo object containing the value doesn't reflect this. We have to explicitly 'type' the literal when we type it in.

Most of the literals accepted in Pharo code are accepted in uFFI call-outs too: e.g., integers, floats, strings, arrays. We will present more detail about the different data types and how they are marshalled by uFFI in a subsequent section.

Class Variables

In the method above we used a literal number as an argument in a uFFI call-out. Although handy, literal numbers fall into the category of so-called "magic numbers": Embedded literals in code that offer no explanation of where they came from, why they were chosen, or how they were calculated. (These are distinguished from *manifest constants*, which are more-or-less self-explanatory, such as using *pi* in angle calculations or '2' when we need to divide a quantity in half.)

Embedding magic numbers in methods is a *code smell* and should be avoided. One way to handle these kinds of values is to parameterize them: assign them names, usually as a variable or named constant. C libraries often define such constants using `#define` pre-processor statements such as:

```
#define MagicNumber -42
```

In Pharo, we can take a similar approach by defining such values in class variables. We only need to change the definition of our `FFITutorial` class to include a class variable such as `MagicNumber` (which is therefore capitalized), and then define a class-side `initialize` method to set its value (*and explain why*). Do **not** forget to execute this `initialize` method, otherwise the value won't get set!

```
Object subclass: #FFITutorial
  ...
  classVariableNames: 'MagicNumber'
  ...

FFITutorial class >> initialize [
  "Set this to -42 because.. Life, the Universe, and Everything."
  MagicNumber := -42.
]
```

Finally, we update our call-out binding to use our class variable, remembering that we still need to provide its type explicitly:

```
FFITutorial class >> absMinusFortyTwo [
  ^ self ffiCall: #( int abs ( int MagicNumber) )
]
```

Just as with class variables, uFFI integrates transparently with variables defined in shared pools. Shared pools are useful for grouping common constants that need to be shared between different classes, bindings, or even libraries.

The following code illustrates how we can modify our code to put our variable in a shared pool. Notice that the only code that changes is the class defining the variable. The binding using the variable remains unchanged.

```
SharedPool subclass: #FFITutorialPool
  ...
  classVariableNames: 'MagicNumber'
  ...

FFITutorialPool class >> initialize [
  "Set this to -42 because.. Life, the Universe, and Everything."
  MagicNumber := -42.
]

Object subclass: #FFITutorial
  ...
  poolDictionaries: 'FFITutorialPool'
  ...

FFITutorial class >> absMinusFortyTwo [
  ^ self ffiCall: #( int abs ( int MagicNumber ) )
]
```

Using a shared pool does not change the normal Pharo usage of uFFI. If you want to learn more about Pharo shared pools, we recommend you take a look at *Pharo by Example 8.0*.

Instance Variables

uFFI also supports using instance variables as arguments to a C call-out. When such a call-out is executed, the value of the instance variable is demarshalled to form the argument, using the type information in the binding definition.

The use of instance variables in uFFI bindings can come in handy when defining object-oriented APIs to C libraries. Since objects hold values as well as define behaviors, they can both encapsulate the state required to perform the uFFI calls while masking the call-outs as normal messages.

So we see that the syntax is essentially the same as what we saw for class variables. Note that we can can even hold the function name in a Pharo variable:

```
MyClass >> doSomething [
  ^ self ffiCall: #( int myFunction ( int myInstanceVariable ) )
    library: LibC
]
```

We will let you, the reader, experiment with these as an exercise.

Special Variables: The Case of `self`

We can also pass `self` as an argument to a C call. Suppose we want to add the `abs()` function call to an extension of the class `SmallInteger`, in a method

named `absoluteValue`. This would allow us to write expressions such as `-50 absoluteValue`.

To do this, we simply add an `absoluteValue` method to `SmallInteger` and directly pass `self` as a (typed) argument. Though a pseudo-variable, Pharo will demarshall `self` correctly as it does any instance variable:

```
SmallInteger >> absoluteValue [
  ^ self ffiCall: #( int abs ( int self ) ) library: LibC
]
```

Using `self` as an argument has several benefits beyond these uses as integers. In combination with complex types such as C `structs` and opaque objects, using `self` as argument is a powerful tool for writing OOP-flavored bindings to C libraries.

3.4 Marshalling C Pointers

In Pharo, the objects we send as method arguments are always conceptually ”passed by reference”. This means that every time we provide an object as an argument, the receiver and the sender hold references to the same object: the argument object is *shared*. (One exception to this is the set of Pharo `SmallIntegers`, which are Pharo primitives, and therefore passed by value. However, this implementation detail is completely transparent to Pharo code, and has no impact beyond the implementation of uFFI demarshalling rules, as we saw earlier).

In contrast to Pharo, many function arguments in C are, by default, ”passed by value”. For example, every time we send an integer or a float as an argument to a function, its value is *copied* (pushed onto the C stack) prior to calling the function. Passing by value, while trivial for simple types such as numbers, usually demands more consideration when dealing with complex *mutable* types, such as arrays and structures. If an invoked function were to modify a *copy* of data received as an argument, the original value held by the caller would remain unmodified – which might not be the intended outcome.

Therefore, in addition to *pass by value*, C too supports passing arguments by reference, in this case using **pointers** (a C type that equates to an address in memory). Use of a C pointer as an argument is often denoted explicitly in function declarations by prefixing the `*` character to a variable name (although some C types are implicitly of pointer type).

For example, a function `foo()` that receives an `int` pointer as an argument and returns a `char` pointer will have a function declaration similar to the following:

```
char *foo ( int *arg );
```

uFFI supports pointers by introducing a new kind of object: `ExternalAddress`. `ExternalAddresses` are objects that represent memory addresses, thus their possible values range from `NULL` to the maximum possible address the host operating system allows, either 2^32 or 2^64 for 32-bit or 64-bit systems, respectively.

Obtaining an `ExternalAddress`

The most common way of obtaining an `ExternalAddress` is to receive it as the return value of a called C function. A good example is the `libc` function `malloc()`, which takes an integer specifying the desired size of a heap buffer, tries to allocate a contiguous block of memory of the size requested, and, if successful, returns a pointer to the allocated region of memory.

The C declaration of such function reads as follows (from the `libc` manual):

```
void *malloc( size_t size );
```

We only need create a uFFI binding to it by copy-pasting the declaration as follows:

```
FFITutorial class >> malloc: aSize [
  ^ self ffiCall: #( void * malloc ( size_t aSize ) ) library: LibC
]
```

Notice that this function returns a "generic pointer", of type `void *` (meaning, what it points to in memory is *untyped*). In uFFI, this is marshalled to a `ExternalAddress` object. In other words, our `malloc()` binding yields an `ExternalAddress` in return (if successful).

For example, if we use the above binding to ask the system to allocate a buffer of 200 bytes for us, then on return we should have an address that points somewhere in the C heap:

```
FFITutorial malloc: 200
 => (void*)@ 16r7FFBDE0DE030
```

But if we ask for more memory than there is currently available, `malloc()` will fail and return a `NULL` pointer instead:

```
FFITutorial malloc: SmallInteger maxVal
 => (void*)@ 16r00000000
```

Consequently, we must always check the return value in cases like this, lest we invoke the infamous "null pointer assignment" bug.

Pointers as Arguments

Pointers can appear as function arguments, too. Consider for example the function `free()`, used to de-allocate a block of memory previously allocated with, e.g., `malloc()`. As the complement to memory allocation functions,

`free()` takes a memory pointer as an argument (which is **required** to have originated from a memory allocation system call!), de-allocates the memory pointed to by it, and returns nothing.

The declaration of `free()` looks like this:

```
void free( void *ptr );
```

As you might expect by now, uFFI supports pointer arguments by copying the C syntax for pointers to use as its notation in function bindings. And as expected, copy-pasting a C function declaration is usually enough to create bindings to such functions:

```
FFITutorial class >> free: ptr [
  ^ self ffiCall: #( void free( void *ptr ) ) library: LibC
]
```

Naturally, pointer arguments accept `ExternalAddress` objects. We have seen in the previous section that functions returning pointers provide us with such `ExternalAddresses`. So we are now able to allocate memory using a `malloc()` binding and give that memory back to the system using a `free()` binding, as shown in this example:

```
anExternalAddress := FFITutorial malloc: 200.

FFITutorial free: anExternalAddress.
```

Warning: Always be careful when manipulating memory, especially when accessing it and freeing it. For each `malloc()` that allocates memory, you must eventually call a corresponding `free()` to release it back to the system. Improper memory manipulation can lead to memory access errors and could cause your process to die (or your system to crash).

Optional arguments, `NULL` pointers, and `nil`

In general when using uFFI, we will not need to craft any pointers manually, other than `NULL` pointers. `NULL` pointers (which are interchangeable) can be easily created with the following expression:

```
ExternalAddress null
  => @ 16r00000000
```

`NULL` pointers are useful because they are used in many libraries that have optional arguments. For example, in the family of memory allocation functions, the function `realloc()` takes two arguments: a pointer to a region of memory and a size. This function tries to enlarge the allocated region of memory to the requested size and returns the same pointer when successful. However, if a `NULL` pointer is provided as argument, `realloc()` operates instead like `malloc()` and just allocates a new region of memory.

The declaration of `realloc()` looks like this:

```
void *realloc( void *ptr, size_t size );
```

And the uFFI binding to this function might look like:

```
FFITutorial class >> realloc: aPointer toSize: aSize [
  ^ self ffiCall: #( void * realloc ( void* aPointer,  size_t aSize
    ) ) library: LibC
]
```

As noted above, we can use this binding for buffer allocation by providing a `NULL ExternalAddress` as its first argument:

```
FFITutorial realloc: ExternalAddress null toSize: 200.
 => (void*)@ 16r7FFBDE0DE030
```

Finally, uFFI type marshalling allows us to use `nil` for arguments, too. When a pointer is expected, `nil` will be demarshalled to a `NULL` pointer. Thus we could write our previous example in this way, too:

```
FFITutorial realloc: nil toSize: 200.
 => (void*)@ 16r7FFBDE0DE030
```

3.5 Type Marshalling Rules for Basic Objects

All Pharo basic objects (integer, floats, strings, booleans, characters) are represented in memory differently in Pharo than in C, thus requiring special marshalling for each of them. Moreover, not all values in one language are seamlessly representable in the other. For example, while integers in Pharo can have arbitrary precision and will be enlarged on demand, integers in C do overflow and will be simply truncated. In this subsection we go over all pharo basic types and we explore all these rough corners the FFI developer should be aware of.

Integers

Integer types in C such as `short`, `int`, their long and unsigned versions, have fixed size. For example, `int`'s are 32 bits long in 32bit machines and 64 bits long in 64bit machines. This means that values that cannot be represented in that size will be truncated to fit. To illustrate the behaviour, consider the following two C functions: they initialize an integer value with the maximum possible value they can store, and then add one to it.

```
void overflowing_int(){
  int input=INT_MAX;
  printf("Overflowing int: %d + 1 = %d\n", input, input + 1);
}

void overflowing_uint(){
  unsigned int input=UINT_MAX;
```

```
  printf("Overflowing unsigned int: %u + 1 = %u\n", input, input +
    1);
}
```

Which yield for example:

```
Overflowing int: 2147483647 + 1 = -2147483648
Overflowing unsigned int: 4294967295 + 1 = 0
```

We observe that the results of executing such functions are (when overflows are admitted/defined behaviour by the compiler) mathematically incorrect. To explain it briefly, such behavior is due to the fact that C integers (and other numeric types) do not entirely follow mathematical semantics, but are instead tightly coupled to the semantics and limitations of the underlying machine and processor. We will refrain ourselves from further discussing the semantics of C, since they are outside the scope of this booklet, and they may vary depending on the C standard and compiler used.

On the other hand, integer values in Pharo have variable size. Indeed, Pharo integers storage size is dynamically decided to fit the stored value. For example, the following piece of code shows how we can obtain the maximum `SmallInteger` value, and that adding one to it yields a `LargePositiveInteger`. The former class is used to represent integers that fit integers up to a fixed size (of 31 bits in 32bit machines and 61 bits in 64bit machines), while the latter is used to represent integers that require more storage.

```
(SmallInteger maxVal + 1) class
 => LargePositiveInteger
```

While uFFI value marshalling takes care of most conversions (e.g., representation differences, two-complements...), the uFFI user still has to be careful when exchanging integers between Pharo and a C library. This mostly boils down to:

- the integer return type and the integer types of the arguments defined in the uFFI callout should correspond to the actual types in the library. E.g., mistaking a `long` by an `int`, or wrongly defining a signed int as not signed will make uFFI interpret such values wrongly.
- Pharo integer values used as arguments should fit in the target argument type. Otherwise such value will be truncated.

Floating Point Numbers

Floating point numeric types in C such as `float` and `double` are also restricted by the limitations of the underlying hardware. Since the hardware is not infinite, floating point numbers cannot be represented with infinite precision, being in fact the opposite: floating point numbers see their precision limited. The C type `float` is a precision floating point number, while `double` is a double precision floating point number. The following example

C code illustrates how precision is lost: it first stores a floating point number in variables of type `float` and `double`, it then prints it using the `printf` function and requiring a precision of 18 digits after the decimal dot.

```
void printFloatingPointNumbers(){
  //Both floating point numbers should have the same value
  float x = 3.141592653589793238;
  double y = 3.141592653589793238;

  printf("Float is: %20.18f\n", x);
  printf("Double is: %20.18f\n", y);
}
```

The example above prints the following in the standard output. Although we asked for a precision of 18 digits, and 18 digits were printed to the stdout, not all of those 18 digits correspond to the original digits specified in our code. Moreover, we see that using a single precision type loses precision before than the one using double precision.

```
Float is: 3.141592741012573242
Double is: 3.141592653589793116
```

Pharo floating point numbers are double precision floating point numbers as explained above. uFFI marshalling will then take care of the following conversions:

- If a Pharo float is sent as a `double` argument, the Pharo double precision floating point number will be copied as is.
- If a Pharo float is sent as a `float` argument, Pharo float will be previously converted to a single precision float before.
- C floats returned by functions will be transformed to double precision Pharo floats. If the function return type is `float` it will then add **noisy** digits to complete its decimal representation.

Characters

Characters in C are represented using the `char` type which is, although unintuive, a numeric type and not a textual type. This means that a `char` value does not actually represent a character, but some bytes encoding a character. And the actual character value of those bytes depend on how those bytes are actually interpreted.

The reason for this is that the C programming language predates the nowadays broadly-used text encoding standards such as Unicode and its encoding formats such as utf-8 and utf-32. Historically, the `char` type stores 1 byte values, and is used to store ascii character values. However, the ascii encoding is not sufficient to represent many languages which have more characters than can possibly fit in a single byte.

On the other hand, Pharo represents characters with their unicode code point, i.e., a unique identifier assigned to each character by the unicode standard. Unicode codepoints are integer numbers that may not fit the typical single byte of the C `char` type.

The uFFI marshalling rules of characters is as follows:

- A Pharo character sent as argument have their code point truncated and copied to the function argument.
- A `char` return value is interpreted as a codepoint, and a Pharo Character with such codepoint is returned.

However, care is needed with these conversions because the `char` type is actually a numeric type, and it mismatches with the unicode codepoints within Pharo. This turns the manipulation of character values into a potential source of bugs. We recommend the FFI developer to carefully study how `char` types are used in the used library. Several C libraries will interpret `char` values as ascii values, but unicode code points only match ascii values up to 127. For example, while in ascii the euro character (€) is represented with the 0x80 value, its unicode code point is 0x20AC. Other libraries will use `char` values to represent raw bytes encoding characters in one of its encodings such as utf-8.

Booleans

Since the C99 standard, C includes support for booleans with the `bool` type, and its `true` and `false` values defined in the `stdbool.h` header. Pharo booleans are transformed to C booleans seamlessly when using the `bool` type in the FFI callout signature.

```
void printBoolean(bool b){
  printf("Boolean: %d\n", b);
}
```

```
FFITutorial class >> printBoolean: b [
  ^ self ffiCall: #( void printBoolean ( bool b ) )
]
```

3.6 More Predefined Data Types

uFFI includes a set of already defined data types. These data types are accessible to all the FFI libraries and call-outs. These data types represent all the data types defined in the C standard and some useful aliases.

We will divide the data types in fixed-size types, aliases and ABI dependent types. The fixed-size have the same size in all the platforms. The ABI dependent types may change their size depending of the running platform. Also,

uFFI includes a set of useful aliases to ease the creation of FFI callout just by copying the function signature.

Fixed Size Types

uFFI includes a set of fixed size integer types. This types are platform independent. They came in two flavours: signed and unsigned.

Type	Byte Size	Signed
int8	1	Yes
int16	2	Yes
int32	4	Yes
int64	8	Yes

Type	Byte Size	Signed
uint8	1	No
uint16	2	No
uint32	4	No
uint64	8	No

Floating-point Types

uFFI provides the data types for handling IEEE-754floating point data types. The following table presents the data-types usable in FFI bindings.

Type	IEEE 754	Mantissa	Exponent	Common Name
float16	binary16	11 bits	5 bits	Half Precision
float32	binary32	24 bits	8 bits	Single Precision
float64	binary64	53 bits	11 bits	Double Precision
float128	binary128	113 bits	15 bits	Quadruple precision

These types are correctly translated to a `Float` instance in Pharo. A `Float` instance in Pharo has a precission equivalent to `float64`.

Platform Dependant Types

uFFI is designed to allow the developers to write code that is portable to different platforms. As there are types with sizes depending of the platform, uFFI implements the following platform dependant types.

This types are automatically handled correctly depending on the running platform.

Type	32 bits size	Windows (64 bits)	Linux (64 bits)	OS X (64 bits)
size_t	4	8	8	8
long	4	4	8	8
ulong	4	4	8	8

Aliases

One of the nicest features of uFFI is the ability to copy and paste the definitions of the C Functions to implement the FFI calls.

To have a more extensive support for copying and pasting the definitions, uFFI includes a series of aliases from common used data types to uFFI types.

The following table presents the integer aliases:

Alias	Target Type
unsignedByte	uint8
unsignedChar	uint8
uchar	uint8
byte	uint8
sbyte	int8
schar	int8
signedByte	int8
signedChar	int8
unsignedShort	uint16
ushort	uint16
short	int16
signedShort	int16
uint	uint32
int	int32
signedLong	int32
unsignedLong	uint32
longlong	int64
ulonglong	uint64

Also there are aliases for floating point types:

Alias	Target Type
shortFloat	float16
float	float32
double	float64

3.7 Conclusion

In this chapter we have studied the basics of uFFI call-out arguments and their marshalling. The arguments of a function can be the the arguments of

the method they are declared in, an instance variable, a class variable, self, or a literal Pharo object. Arguments should declare a type, so uFFI knows how to transform that object to a C equivalent. Failing in correctly declaring the type can lead to misbehaviours in an application due to incorrect transformations.

Basic types covered by uFFI include integer values, floating point values, booleans, characters and pointers. Pointers are used through the special FFI class `ExternalAddress`, and its special value *NULL* can be acquired with the `null` class side message. Finally other basic types have special marshalling rules that apply when transforming Pharo values to C values. In particular we have seen that integers suffer from truncations and overflows in C, floating point numbers suffer from lose of precision, and characters in Pharo are interpreted as unicode code points while each C library can interpret characters as they please.

The next chapter explores other data types that can be built from these basic data types: arrays, structures, enumerations and opaque objects.

CHAPTER 4

Complex Types

We have seen in the last two chapters that types are a big deal in FFI call-outs, since they govern how values are marshalled between C and Pharo. The previous chapter explored the marshalling rules of basic types such as integers, floating point numbers, booleans and characters. But in addition to these basic types, existing libraries may make use of more complex data-types, i.e., data-types that are built from simpler data-types. This chapter builds on top of the knowledge of previous chapters to introduce these more complex types.

This chapter starts by showing how to define type aliases. Type aliases are user-defined alternative names for other types, usually used to improve the readability of the code. In addition, you'll see further in this chapter how uFFI exploits them to define more complex types.

uFFI provides support to map C complex data-types,such as arrays, structures, enumerations and unions. Arrays are data-types defining a bound sequence of values of a single data-type, e.g., a sequence of 10 integers. Structures are data-types defining a collection of values of heterogeneous data-types, e.g., a group of an integer and a two booleans. Enumerations are data-types defining a finite set of named values, e.g., the characters between a and z. Unions are data-types defining a single value that can be interpreted with different internal representations, e.g., we may want to see a float as an int to extract its mantissa.

4.1 Defining Type Aliases

A type alias is a user-defined alternative name for a type, which are useful in many different scenarios. As one example, type aliases are useful to improve

code readability, by creating domain specific types, e.g., `age => unsigned int`. As another example, external libraries can come with their own user-defined and types and aliases, so having the same aliases in our FFI bindings can simplify writing those bindings.

A first type alias

In uFFI, type aliases are created with class variables: the class variable name is the alias name, while its value is the aliased type. For example, we define our `age => unsigned int` alias as follows in our `FFITutorial` class, and then execute the initialize method to make it run.

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

FFITutorial class >> initialize
  Age := #uint
```

Once our type alias is defined, and the class side `initialize` executed, we can use that type alias anywhere in our bindings in that hierarchy below `FFITutorial`. For example, we can define our `abs()` binding as follows.

```
FFITutorial class >> abs: n [
  ^ self ffiCall: #( Age abs ( Age n ) )
]
```

Valid values for type aliases

As we have seen above, uFFI type aliases are defined by normal assignments to class variables:

```
Age := #uint
```

uFFI type alias names, on the left of the assignment, can have any name accepted as a class variable name. Although other names are be accepted by Pharo class definitions, the convention tells class variables should be capitalized. The value of a type alias, on the right of the assignment, could be either:

- a symbol with a type identifier to be resolved by uFFI, e.g., `#'int'`, a shown above or;
- an already resolved type object, which we will study in subsequent sections

Before executing a call-out, uFFI verifies all types used in the call-out can be resolved to valid types, and will throw an exception if an error occurs while resolving it.

Sharing type aliases with shared pools

uFFI does not enforce how bindings should be structured by a developer. A developer could choose to put all bindings in a single class, or, he could organize them in several classes even amongst several packages. Regardless of how the code is structured, it is most of the times useful to have user-defined type aliases available in all classes using the bindings. For this purpose, uFFI supports structuring type aliases in shared pools.

A typical usage of shared pools to define uFFI types is to define a `MyLibraryTypes` shared pool as follows, as we did before to define constants:

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

FFITutorialTypes class >> initialize
  Age := #uint
```

We can then import the type aliases in the shared pool by

```
Object subclass: #FFITutorial
  ...
  poolDictionaries: 'FFITutorialTypes'
  ...
```

4.2 Arrays

Arrays are a bound sequence of contiguous values. In Pharo, an array object can contain any object, and specially, a single array can contain objects of different types. For example, the following code snippet shows how a single array can contain integers, floats, strings and others.

```
anArray := Array new: 7.
anArray at: 1 put: 3.1415.
anArray at: 5 put: 42.
anArray at: 7 put: 'Hello World'.

anArray.
 => #(3.1415 nil nil nil 42 nil 'Hello World')
```

In addition, Pharo arrays can be safely accessed without producing buffer over/underflows, because it performs bound checks on each array access. In other words, accessing outside the bounds of a Pharo array yields an exception instead of accessing data **outside** the array.

```
anArray := Array new: 1.

anArray at: 2
 => Out of Bounds Exception!
```

C arrays behave partially like Pharo arrays: they are contiguous sequences of values. However, C arrays are constrained to contain values of a single type, and accessing outside of their bounds is not checked before the access, producing buffer over/underflows.

Because of these differences and differences in their internal representation, uFFI does not automatically marshall Pharo arrays into C arrays. Instead, uFFI provides a specialized array class to manipulate arrays: the `FFIArray` class.

Creating `FFIArrays`

`FFIArrays` are created using the `newType:size:` or the `externalNewType:size:` instance creation methods. The former will allocate an array in the Pharo heap, while the latter one will allocate the array in the C heap.

```
"In the pharo heap"
array := FFIArray newType: #char size: 10.

"In the C heap"
array := FFIArray externalNewType: #char size: 10.
```

`FFIArrays` allocated in the C heap are not moved and their memory is not released automatically. It is the developer's responsibility to free it. On the other hand, `FFIArrays` allocated in the Pharo heap can be moved by the garbage collector, so they should be pinned in memory before being safely used in FFI calls. Also, `FFIArrays` in the Pharo heap are managed by Pharo's garbage collector, and will be collected if no other Pharo objects reference it. Be careful, an `FFIArray` in the Pharo heap referenced from the C heap will still be garbage collected making the pointer in the C heap a dangling pointer.

Manipulating `FFIArray` instances

The elements in an `FFIArray` are accessed as any other Pharo array, using the `#at:` and `#at:put:` methods. Its size is accessed with the `#size` method, using 1-based indexes like in Pharo.

```
array at: 1 "for the first element".
array at: n "for the nth element".
```

If the array is allocated in the Pharo heap, array accesses will be bound checked and throw an exception in case of out-of-bounds access. Otherwise, if the ar-

ray is allocated in the C heap, array accesses may run into buffer over/underflows.

Reusable FFIArray types

From time to time, we need to create several array instances of the same type and size. Besides instantiating single arrays, FFIArray can define array types, using the newArrayType:size: method. An array type knows the types of its elements and its size and we can simply allocate it using the new or externalNew messages to allocate it in the Pharo heap or the C heap respectively.

```
char128Type := FFIArray newArrayType: #char size: 128.

"In Pharo heap"
newArrayOf128Chars := char128Type new.

"In C heap"
newArrayOf128Chars := char128Type externalNew.
```

FFIArrays created this way can be used as any other FFIArray. We will see in the next section how this array definition is useful to combine arrays inside structures.

4.3 Structures

A structure is data-type that joins together a group of variables, so-called fields. Each field of a structure has a name and a type. Structures are often used to group related values, so they can be manipulated together. For example, let's consider a structure that models a fraction, i.e., a number that has a numerator and a denominator. Both numerator and denominator can be defined as fields of type int. Such fraction structure data-type, and a function calculating a double precision floting point number from it, are defined in C as follows:

```
typedef struct
{
  int numerator;
  int denominator;
} fraction;

double fraction_to_double(fraction* a_fraction){
  return a_fraction -> numerator / (double)(a_fraction ->
    denominator);
}
```

Defining a structure with `FFIStructure`

Structures are declared in uFFI as subclasses of the `FFIStructure` class defining the same fields as defined in C. For example, defining our fraction structure is done as follows, defining a subclass of `FFIStructure`, a `fieldsDesc` class-side method returning the specification of the structure fields, and finally sending the `rebuildFieldAccessors` message to the structure class we created.

```
FFIStructure subclass: #FractionStructure
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

FractionStructure class >> fieldsDesc [
  ^ #(
    int numerator;
    int denominator;
    )
]

FractionStructure rebuildFieldAccessors.
```

Doing this will automatically generate some boilerplate code to manipulate the structure. You will see that the structure class gets redefined as follows, containing some auto-generated accessors.

```
FFIStructure subclass: #FractionStructure
  instanceVariableNames: ''
  classVariableNames: 'OFFSET_DENOMINATOR OFFSET_NUMERATOR'
  package: 'FFITutorial'

FractionStructure >> denominator [
  "This method was automatically generated"
  ^handle signedLongAt: OFFSET_DENOMINATOR
]

FractionStructure >> denominator: anObject [
  "This method was automatically generated"
  handle signedLongAt: OFFSET_DENOMINATOR put: anObject
]

FractionStructure >> numerator [
  "This method was automatically generated"
  ^handle signedLongAt: OFFSET_NUMERATOR
]

FractionStructure >> numerator: anObject [
  "This method was automatically generated"
  handle signedLongAt: OFFSET_NUMERATOR put: anObject
```

```
]
```

Once a structure type is defined, we can allocate structures from it using the `new` and `externalNew` messages, that will allocate it in the Pharo heap or the external C heap respectively.

```
"In Pharo heap"
aFraction := FractionStructure new.

"In C heap"
aFraction := FractionStructure externalNew.
```

We read or write in our structure using the auto-generated accessors.

```
aFraction numerator: 40.
aFraction denominator: 7.
```

And we can use it as an argument in a call-out by using its type.

```
FFITutorial >> fractionToDouble: aFraction [
  ^ self ffiCall: #(double fraction_to_double(FractionStructure*
    a_fraction))
]

FFITutorial new fractionToDouble: aFraction.
>>> 5.714285714285714
```

Structures by copy or by reference

Structures in C can be passed as argument both by copy and by reference. A structure passed by copy means that a copy of the entire structure has to be done and sent to the calling function. A structure passed by reference means that a pointer to the same structure is shared between the caller and callee. In C, we can distinguish between these two in two cases: 1) by the appearance of the pointer type modifier ***** in type declarations, and 2) by the need to explicitly send a pointer to the function expecting pointers.

```
int print_by_pointer(fraction* a_fraction_reference);
int print_by_copy(fraction a_fraction_copy);

...

fraction f;

//The function below requires a copy, so just send f, the compiler
    takes care
print_by_copy(f);

//The function below requires a pointer, so dereference f
print_by_pointer(&f);
```

In uFFI, such a difference is also reflected in FFI bindings by using or not a pointer type:

```
FFITutorial >> printByPointer: aFraction [
  ^ self ffiCall: #(int print_by_pointer(FractionStructure*
    a_fraction))
]

FFITutorial >> printByCopy: aFraction [
  ^ self ffiCall: #(int print_by_copy(FractionStructure a_fraction))
]
```

However, in contrast with C, does not require explicit pointers send to the functions by dereferencing structures. Just sending the structure object as argument will take care of the dereferencing if needed.

```
"In C heap"
aFraction := FractionStructure externalNew.

aFraction numerator: 40.
aFraction denominator: 7.

"Both below do work"
FFITutorial new printByPointer: aFraction.
FFITutorial new printByCopy: aFraction.
```

Structures embedding arrays

Arrays in C can appear embedded in structures defined as follows, where a structure contains an array of four ints.

```
struct {
  int some_array[4];
}
```

Constrastingly with a struct containing *a pointer* to an array, the structs created from the definition above will contain the entire array allocated within the struct. Choosing between a struct that embeds an array or one that references an array through a pointer is a responsibility of the author of the C library we are binding, and it is outside the scope of the booklet. However, different structures are defined differently in uFFI.

Defining the structure above in uFFI requires that we define an array type of size 4 for our `some_array` field. We can define such user defined type as a type alias in our class-side, import our type pool in our structure class and then use our type in our field definitions.

```
FFITutorialTypes class >> initialize [
  int4array := FFIArray newArrayType: #int size: 4.
]

FFIStructure subclass: #EmbeddingArrayStructure
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: 'FFITutorialTypes'
  package: 'FFITutorial'

EmbeddingArrayStructure class >> fieldsDesc [
  ^ #(
  int4array some_array;
  )
]

EmbeddingArrayStructure rebuildFieldAccessors.
```

Structure Alignment

Structures are usually organised in memory as a contiguous region containing all its fields in the order they were defined. However, compilers usually align structure fields to simplify the access to them, by adding some **padding**, i.e., hidden fields that occupy some space to force subsequent fields to move to the desired position.

For example, consider the structure with two fields `a` and `b` of types `char` and `int` respectively. Although the `char a` field only occupies 1 byte, the second field `b` starts in the fifth byte: it is aligned to 4 bytes. This means the compiled version of such a struct adds 3 bytes of padding between our two fields.

```
// We define
struct {
  char a;
  int b;
}

// The compiler defines
struct {
  char a;
  char padding1[3];
  int b;
}
```

uFFI handles paddings and alignments automatically respecting the C standard behavior. We can define the structure above using uFFI as:

```
FFIStructure subclass: #AlignmentExampleStructure
instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

AlignmentExampleStructure class >> fieldsDesc [
  ^ #(
    char a;
    int b;
    )
]

AlignmentExampleStructure rebuildFieldAccessors.
```

And then test that the fields are correctly aligned: `a` is the first byte, `b` is the fifth:

```
AlignmentExampleStructure classPool at: #OFFSET_A.
>>> 1
AlignmentExampleStructure classPool at: #OFFSET_B.
>>> 5
```

Packed Structures

From time to time we will find libraries that use **packed structures**. Packed structures are structures that are compiled without some or all of its padding. For example, some compilers will use the pragma `pack` to tweak the alignment of structures.

```
#pragma pack(push)  /* push current alignment to stack */
#pragma pack(1)     /* set alignment to 1 byte boundary */

struct {
  char a;
  int b;
}

#pragma pack(pop)   /* restore original alignment from stack */
```

Some other compilers will have directives to specify packing at the level of a field:

```
struct {
  char a;
  int b __attribute__((packed));
}
```

uFFI provides support for mapping **packed structures** through the `FFIPackedStructure` class, which is a subclass of `FFIStructure` that redefines how fields are aligned. `FFIPackedStructure` does avoid all paddings, creating a single packed structure where each field follows the next one. Consider

the example of `FFIPackedStructure` below, mapping our packed structure above.

```
FFIPackedStructure subclass: #PackedAlignmentExampleStructure
instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

PackedAlignmentExampleStructure class >> fieldsDesc [
  ^ #(
    char a;
    int b;
    )
]

PackedAlignmentExampleStructure rebuildFieldAccessors.
```

Differently from the non-packed structure of a couple of sections ago, this packed structure shows that both fields are contiguous: `a` is the first byte, `b` is the second:

```
PackedAlignmentExampleStructure classPool at: #OFFSET_A.
>>> 1
PackedAlignmentExampleStructure classPool at: #OFFSET_B.
>>> 2
```

4.4 Enumerations

Enumerations are data-types defining a finite set of named values. For example, let's consider we want to create a data-type to identify the different positions of players inside a football match: goalkeeper, defender, midfielder, forward. Such data-type can be defined in `C` as an enumeration as follows:

```
typedef enum {
  goalkeeper,
  defender,
  midfielder,
  forward
} position;
```

We can then use `position` as a type, and the values defined within it as valid values for `position`.

```
position myPosition = defender;
```

The values of C enumerations

To better understand how to map `C` enumerations using uFFI, we must before understand how `C` assigns value to each of the elements in the enumeration.

Internally, C assigns to each of the elements of the enumeration a sequencial numeric value starting from 0 (zero). In other words, `goalkeeper` has a value of 0, `defender` has a value of 1, and so on. C allows developers to specify the values they want too, using an assignment-like syntax.

```
typedef enum {
  goalkeeper = 42,
  defender,
  midfielder,
  forward
} position;
```

We can explicitly assign values to any of the elements of the enumeration. We may leave values without explicit values, in which case they will be automatically assigned a value following its previous value. And finally, many elements in the enumeration may have the same value. The example enumeration below shows these subtleties.

```
#include <assert.h>
#include <limits.h>

enum example {
    example0,            /* will have value 0 */
    example1,            /* will have value 1 */
    example2 = 3,        /* will have value 3 */
    example3 = 3,        /* will have value 3 */
    example4,            /* will have value 4 */
    example5 = INT_MAX,  /* will have value INT_MAX */
    /* Defining a new value after this one will cause an overflow
     error */
};
```

Defining an enumeration using `FFIEnumeration`

Enumerations are declared in uFFI as subclasses of the `FFIEnumeration` class defining the same elements as defined in C, and with their same values. For example, defining our example enumeration is done as follows, defining a subclass of `FFIEnumeration`, a `enumDecl` class-side method returning the specification of the enumeration elements, and finally sending the `initialize` message to the enumeration class we created.

```
FFIEnumeration subclass: #ExampleEnumeration
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

ExampleEnumeration class >> enumDecl [
  ^ #(
    example0 0
    example1 1
```

```
    example2 3
    example3 3
    example4 4
    example5 2147483647
    )
]

ExampleEnumeration initialize.
```

Doing this will automatically generate some boilerplate code to manipulate the enumeration. You will see that the enumeration class gets redefined as follows creating and initializing a class variable for each of its elements.

```
FFIEnumeration subclass: #ExampleEnumeration
  instanceVariableNames: ''
  classVariableNames: 'example0 example1 example2 example3 example4
    example5'
  package: 'FFITutorial'
```

To use the values of enumerations in our code, it is enough to import it as a pool dictionary, since uFFI enumerations are shared pools.

```
Object subclass: #FFITutorial
  ...
  poolDictionaries: 'ExampleEnumeration'
  ...
```

4.5 Unions

Unions are data-types defining a single value that can be interpreted with different internal representations. For example, the next piece of C code defines a type `float_or_int` that can be seen as a float or as an int.

```
typedef union {
  float as_float;
  int as_int;
} float_or_int;

float_or_int number;
number.as_float = 3.14f;

printf("Integer representation of PI: %d\n", number.as_int);
```

producing the next output:

```
Integer representation of PI: 1078523331
```

Defining an union using `FFIUnion`

Unions are declared in uFFI as subclasses of the `FFIUnion` class defining the same fields as defined in C, similarly as structures. For example, defining our `float_or_int` union is done as follows, defining a subclass of `FFIUnion`, a `fieldsDesc` class-side method returning the specification of the union fields, and finally sending the `rebuildFieldAccessors` message to the union class we created.

```
FFIUnion subclass: #FloatOrIntUnion
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

FloatOrIntUnion class >> fieldsDesc [
  ^ #(
    float as_float;
    int as_int;
    )
]

FloatOrIntUnion rebuildFieldAccessors.
```

Doing this will automatically generate some boilerplate code to manipulate the values inside the union. You will see that the union class gets redefined like structures did, containing some auto-generated accessors.

```
FFIStructure subclass: #FloatOrIntUnion
  instanceVariableNames: ''
  classVariableNames: 'OFFSET_DENOMINATOR OFFSET_NUMERATOR'
  package: 'FFITutorial'

FloatOrIntUnion >> as_float [
  "This method was automatically generated"
  ^handle floatAt: 1
]

FloatOrIntUnion >> as_float: anObject [
  "This method was automatically generated"
  handle floatAt: 1 put: anObject
]

FloatOrIntUnion >> as_int [
  "This method was automatically generated"
  ^handle signedLongAt: 1
]

FloatOrIntUnion >> as_int: anObject [
  "This method was automatically generated"
  handle signedLongAt: 1 put: anObject
```

```
]
```

Using the defined union type

Once a union type is defined, we can allocate unions from it using the `new` and `externalNew` messages, that will allocate it in the Pharo heap or the external C heap respectively.

```
"In Pharo heap"
aFloatOrInt := FloatOrIntUnion new.

"In C heap"
aFloatOrInt := FloatOrIntUnion externalNew.
```

We read or write in our union using the auto-generated accessors.

```
foi as_float: 3.14.
foi as_int.
>>> 1078523331
```

And we can use it as an argument in a call-out by using its type.

```
FFITutorial >> firstByte: float_or_union [
  ^ self ffiCall: #(char float_or_int_first_byte(FloatOrIntUnion*
    float_or_union))
]
```

4.6 Conclusion

In this chapter we have seen how complex C data-types can be mapped with uFFI. In contrast with basic types, which are automatically marshalled between Pharo and C, uFFI does not automatically marshall complex data-types. The reasoning behind this decision is that the memory layout of C complex data-types is entirely different than Pharo objects. Instead of automatically marshall Pharo objects into these complex data-types, uFFI does reify C them and allows developers to manipulate them through messages in normal Pharo code.

uFFI provides representations for arrays, structures, enumerations and unions. Moreover, these types can be combined, through the usage of type aliases.

In the next chapter we will study how we can define FFI bindings in an object-oriented fashion, using encapsulation, inheritance and delegation.

CHAPTER 5

Designing with uFFI and FFI Objects

In the previous chapters we have studied the uFFI mechanics necessary to bind an external C library. We have studied in particular how to call functions, how basic types are marshalled, and how we can manipulate more complex types. However, we have not yet discussed how those call-out bindings and types should be structured in a project.

This chapter presents several ways to organize library bindings. The first approach presented is the naïf-yet-simple *single library object* that organises the bindings as a façade. In addition, uFFI allows one to take a second approach exploiting the object-oriented nature of Pharo, in particular splitting the binding in different objects and encapsulating the data they manipulate. For this, we exploit the concepts of **external objects** and **opaque objects**.

5.1 First approach: single library façade object

The first approach for organizing external library bindings is to follow the Façade design pattern. In other words, there is a single object that concentrates all bindings and types of our library. This approach is useful to have an idea of our bindings with a naked eye, since all bindings are together.

The `FFILibrary` as a single point of access

uFFI proposes an idoneous place for such a pattern of usage: our library object, subclass of `FFILibrary`. We have seen in previous chapters that when defining a call-out, we need to specify the external library where the func-

tion is located, either at the definition of the call-out using the `ffiCall:library:` method:

```
FFITutorial >> abs: n [
  self ffiCall: #( int abs (int n) ) library: MyLibC
]
```

Or by omitting the library in the call-out and redefining the method `ffiLibrary` to specify the library for all call-outs in the class.

```
FFITutorial >> abs: n [
  self ffiCall: #( int abs (int n) )
]

FFITutorial >> ffiLibrary [
  ^ MyLibC
]
```

When we choose to use the Façade pattern for our bindings, uFFI provides a convenience behaviour: for all subclasses of `FFILibrary`, the method `ffiLibrary` returns `self`:

```
FFILibrary >> ffiLibrary [
  ^ self
]
```

This convenience behaviour allows binding developers to place call-out bindings inside the library object, and avoid declaring the `ffiLibrary` at all. By using this pattern, the library object is not only the object that knows how to find the external library and find the functions: it becomes also a reification of the external library with its functions too.

Redefining `MyLibC` as a façade

Following the pattern explained above, we can now turn our example library `MyLibC` from the first chapter into a Façade, as follows:

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

MyLibC >> unixLibraryName [
  ^ 'libc.so.6'
]

MyLibC >> macLibraryName [
  ^ 'libc.dylib'
]

MyLibC >> win32LibraryName [
```

```
  "While this is not a proper 'libc', MSVCRT has the functions we
    need here."
  ^ 'msvcrt.dll'
]

MyLibC >> ticksSinceStart [
  ^ self ffiCall: #( uint clock() )
]

MyLibC >> time [
  ^ self ffiCall: #( uint time( NULL ) )
]
```

Which can be used as any normal object now:

```
MyLibC new time
```

We leave it as an exercise for the reader to explore the differences between putting the call-out bindings as instance-side or class-side methods.

As the reader may see, this approach's is the strongest when we are binding a small library or a small subset of a large library. The main advantage is that the entire binding can be understood by taking a look at a single class. However, as soon as bindings grow in complexity, we need to re-structure and refactor our bindings into different objects, as we will see in the following sections.

5.2 Extracting behaviour into objects

The ffi library façade object may become a big god object when the library we are binding is big or complex. To cope with this inherent complexity, uFFI provides several mechanisms to extract FFI call-outs into objects. **External objects** have special marshalling rules which combined with `self` arguments allow us to design nice object-oriented APIs. Also, uFFI provides support for **Opaque objects**, which are external objects whose internal implementation is not exposed by the ffi library.

External objects marshalling rules

External objects are uFFI objects representing objects from the external library. We have already seen several external objects such as structures, unions and enumerations, which require creating subclasses of `FFIStructure`, `FFIUnion` or `FFIEnumeration` respectively. uFFI external objects are normal Pharo objects wrapping an external address, hold in a *handle* instance variable. This *handle* instance variable is defined in `ExternalObject` the common superclass of all external objects.

```
Object subclass: #ExternalObject
  instanceVariableNames: 'handle'
  classVariableNames: ''
  package: 'FFI-Kernel'
```

In the previous examples we have also seen that we can use external objects in our callouts by specifying their class name. Using external objects in callouts is possible because uFFI defines special marshalling rules for such objects.

- When a uFFI external object is sent as argument, uFFI will use its *handle* in the callout, thus using the actual memory pointer.
- When a uFFI external object is expected as return value, uFFI expects that the returned value is a pointer and it instantiates an external object of the specified type setting that pointer as *handle*.

An example of external object marshalling

To study how marshalling of external objects work, we will re-introduce the fraction example from the last chapter. Consider the fraction structure both in C:

```
typedef struct
{
  int numerator;
  int denominator;
} fraction;

double fraction_to_double(fraction* a_fraction){
  return a_fraction -> numerator / (double)(a_fraction ->
    denominator);
}
```

And its uFFI binding:

```
FFIStructure subclass: #FractionStructure
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

FractionStructure class >> fieldsDesc [
  ^ #(
    int numerator;
    int denominator;
    )
]
```

Marshalling external object arguments

As an example let's consider the `fraction_to_double` from before and its usage below:

```
FFITutorial >> fractionToDouble: aFraction [
  ^ self ffiCall: #(double fraction_to_double(FractionStructure*
    a_fraction))
]

FractionStructure rebuildFieldAccessors.
aFraction := FractionStructure externalNew.
aFraction numerator: 40.
aFraction denominator: 7.
FFITutorial new fractionToDouble: aFraction.
>>> 5.714285714285714
```

In the example above we see that the call-out binding makes use of our defined type `FractionStructure`. However, when using this binding, we send as argument a fraction object, which is a normal Pharo object. Using external object types as function arguments is actually a uFFI syntax sugar for the following (not so nice) binding. This binding makes use of pointers with `void*` argument types and breaks the encapsulation of our structure to access its internal handle, coupling itself with the internals of uFFI.

```
FFITutorial >> fractionToDouble: aFraction [
  ^ self ffiCall: #(double fraction_to_double(void* a_fraction))
]

...
FFITutorial new fractionToDouble: aFraction handle.
>>> 5.714285714285714
```

Marshalling external object return values

External object types can be used as return values of functions too. Consider the following C function that creates a fraction struct and its uFFI binding. Using `FractionStructure` as return type of our binding, we tell uFFI to take the pointer returned from the function and to create a `FractionStructure` with that pointer as handle.

```
fraction* make_fraction(int numerator, int denominator){
  fraction* f = (fraction*)malloc(sizeof(fraction));
  f -> numerator = numerator;
  f -> denominator = denominator;
  return f;
}
```

```
FFITutorial >> newFractionWithNumerator: numerator denominator:
    denominator [
  ^ self ffiCall: #(FractionStructure* make_fraction(int numerator,
    int denominator))
]

aFraction := FFITutorial new newFractionWithNumerator: 40
    denominator: 7.
FFITutorial new fractionToDouble: aFraction.
>>> 5.714285714285714
```

As with arguments, external object return type is actually a uFFI syntax sugar for the following (again not so nice) binding. This binding makes use of a pointer with `void*` return type and manually initializes structure from it, coupling itself with the internals of uFFI.

```
FFITutorial >> newFractionWithNumerator: numerator denominator:
    denominator [
  ^ self ffiCall: #(void* make_fraction(int numerator, int
    denominator))
]

aFraction := FractionStructure fromHandle: (FFITutorial new
    newFractionWithNumerator: 40 denominator: 7).
FFITutorial new fractionToDouble: aFraction handle.
>>> 5.714285714285714
```

Using external objects as `self` argument

To make our bindings more object-oriented, the next step is to move the behaviour manipulating our objects closer to our objects. In other words, define our call-outs in the classes they manipulate, e.g., ask a fraction to transform itself into double.

```
aFraction asDouble.
>>> 5.714285714285714
```

A naïve, yet working, implementation of such binding requires to move our already-existing methods to the `FractionStructure` class:

```
FractionStructure >> asDouble [
  ^ self fractionToDouble: self
]

FractionStructure >> fractionToDouble: aFraction [
  ^ self ffiCall: #(double fraction_to_double(FractionStructure*
    a_fraction))
]
```

However, uFFI allows us to enhance our bindings even further, combining external objects with `self` arguments in our call-outs. Indeed, our methods

`asDouble` and `fractionToDouble:` can be merged in a single one using `self` as a literal argument of the function.

```
FractionStructure >> asDouble [
  ^ self ffiCall: #(double fraction_to_double(FractionStructure
    *self))
]
```

5.3 Opaque Objects

Many libraries hide their internal representation using opaque data-types. An opaque data-type is data-type whose internal representation is not exposed to us. We can see it as a structure which fields are not visible to us, the point being that it may not be a structure. Libraries using such data-types allow one to only create values of such types and manipulate them through their functions, which renders an API similar to encapsulated objects. uFFI provides support for opaque objects through the `FFIOpaqueObject` class.

Defining an opaque type

Consider an external function which publishes its public API through a header file with function definitions. This header file defines a type `fraction` although we do not know how it is internally defined.

```
typedef struct str_fraction fraction;
fraction mk_fraction(int numerator, int denominator);
float fraction_to_float(fraction f);
```

The simplest way to map such definitions is through type aliases:

```
FFITutorial class >> initialize [
  fraction := #FFIOpaqueObject.
]

FFITutorial class >> makeFractionFromNumerator: n denominator: d [
  self ffiCall: #(fraction mk_fraction(int n, int d)).
]

FFITutorial >> fractionToFloat: aFraction [
  self ffiCall: #(float fraction_to_float (fraction aFraction))
]
```

Opaque types are External Objects

To make the example above more object-oriented, opaque data-types can be easily defined as external objects with the class `FFIOpaqueObject`. An `FFIOpaqueObject` is an external object that assumes nothing about its internal representation: it's just a pointer to some external data. Moreover,

we can then define bindings inside that class, and use `self` as argument to simplify our bindings.

```
FFIOpaqueObject subclass: #OpaqueFraction
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'FFITutorial'

OpaqueFraction class >> makeFractionFromNumerator: n denominator: d [
  self ffiCall: #(OpaqueFraction mk_fraction(int n, int d)).
]

OpaqueFraction >> toFloat [
  self ffiCall: #(float fraction_to_float (self))
]
```

5.4 Conclusion

In this chapter we have seen two different strategies for mapping external libraries: façades and external objects. Façades are the simplest approach, and pretty straightforward for simple or small libraries: they are a god-like objects containing all the call-out definitions of our bindings. External objects allow us to distribute our bindings in a more object-oriented fashion, putting the behaviour closer to the data, and exploiting encapsulation. Complex data-types, as defined in the previous chapters (unions, structures, enumerations), are external objects in this sense. Moreover, we saw that we can define **complex data-types** from opaque objects and type aliases. These two strategies are not incompatible between them, so a same library mapping can mix-and-match and extract functions into external objects as they need it.

CHAPTER 6

uFFI and memory management

Using uFFI requires developers to be concious about how memory is managed, because differently than C, Pharo is a garbage-collected language. On the one hand, most C libraries will require users to make manual bookeeping of the memory they use by explicitly allocating or de-allocating memory. On the other hand, Pharo will automatically reclaim unused Pharo objects, or move them in the memory if so is required.

Binding developers need to be extra careful when design their bindings to manage these differences. Failing to do so will produce many funny effects such as memory leaks that are hard to detect and incorrect memory accesses.

In this chapter we re-visit how external objects are allocated (and de-allocated) in both the Pharo and the C memory. We see the case of Pharo objects sent by reference to C libraries, and we introduce the concept of **pinned objects**: objects that are not be moved by the garbage collector, yet are still garbage collected. Finally, we introduce uFFI auto-released objects: Pharo external objects that will automatically release their C counterpart when they are garbage collected.

6.1 Pharo vs C memory management: the basics

Memory in Pharo and in C is managed in fundamentally different ways. Pharo has automatic memory management where a garbage collector tracks used-objects, moves them around and periodically reclaims unused objects. C requires developers to manually manage their memory. This section introduces the difference between these two models, and finally introduces the subtleties of uFFI external objects which allow both of them.

Memory in C

C programs organize their usage of memory in three different ways: static, automatic and dynamic memory. Static memory is memory allocated when the process starts and not released until the process finishes. How much memory to allocate is known before the execution, typically calculated at compile time. For example, `static` variable declarations in a C program tell the compiler how much memory to pre-allocate for them.

```
static int pre_allocated_variable = 5;
```

Automatic memory is the memory allocated and released without explicit developer intervention. For example, space for the local variables of functions is automatically allocated when functions are called, and released when functions return. Automatic memory is generally managed by **SD strange** the `stack`, i.e., extra space is allocated in the stack on a function call, and automatically released on function return because the extra space is popped from the stack to come back to the caller function. How much extra memory has to be allocated in the stack is generally calculated at compile time.

```
void some_function(){
  // Automatic variable allocated in the stack
  int t = 42;
}
```

Finally, dynamic memory is the memory that cannot be statically calculated, so programs explicitly allocate and reclaim it. Dynamic memory is manipulated through system libraries, for example with the functions `malloc` and `free`. This kind of memory is said to be stored in the **heap**, since the memory allocated by the system is usually organized with a heap data-structure. Memory dynamically allocated needs to be manually released, otherwise provoking potential memory leaks.

```
  //Allocate 17 bytes and grab a pointer to that memory region
  int* pointer = (int*)malloc(17);
  //Free that memory
  free(pointer);
```

Objects in the Pharo Heap

Pharo programs feature automatic memory management: all objects need to be explicitly allocated, and are automatically reclaimed by a garbage collector when they are not used anymore. Objects are allocated in Pharo by sending the `new` and `new:` messages to a class. Although there are several kind of classes in Pharo, for the purpose of this booklet we will concentrate on the two main kind of classes: fixed-size and variable-size classes.

Fixed-size classes are classes with a fixed number of instance variables, instantiated with the message `new`. When the VM is instructed to instantiate

one of these classes, it calculates the amount of memory required for the instance by looking at the class' instance variables. This is the case of most of the classes we create ourselves. For example, class `Point` is a fixed-size class with two instance variables `x` and `y`.

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'

Point new
```

Variable-size classes are classes whose instances have variable size. For example, the `Array` class allows instances with 0 or many slots. These classes are instantiated through the `new: aSize` message, specifying the number of required slots at instantiation-time.

```
ArrayedCollection variableSubclass: #Array
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Collections-Sequenceable-Base'

Array new: 20.
```

In contrast with C-managed memory, once instantiated, the life-cycle of a Pharo object is automatically managed by the virtual machine. For the purposes of this uFFI booklet, it is important to know two main properties of Pharo's garbage collector: 1. the storage of an object that is not used anymore will be automatically reclaimed; 2. the position of an object may change during execution to avoid memory fragmentation. Although these two properties are nice from a Pharo perspective, they require special attention for a uFFI developer, as she has a foot on the Pharo world and a foot in the C world. We will see in the following section how these properties affect programming with uFFI, and how Pharo and uFFI provide support to minimize the impact of these issues through auto-release and pinning.

6.2 uFFI external objects in the C Heap

External objects such as structures, arrays or unions, support a special kind of instantiation message: `externalNew`. This message allocates external objects in the C heap, similar to what the `malloc` function does.

```
myStructure := MyStructure externalNew.
```

Using external allocation of external objects means that we need to do manual deallocation too, using the message `free`.

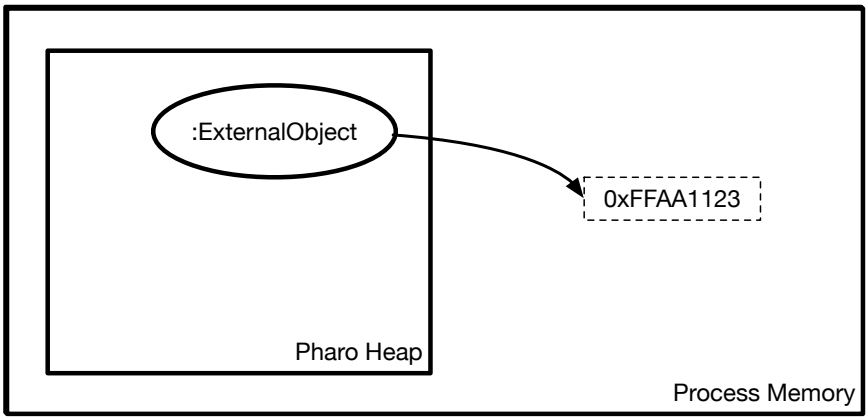

Figure 6-1 An External Address references an address outside the Pharo memory

```
myStructure := MyStructure externalNew.
" ... use my structure ... "
myStructure free.
```

External allocation is useful when we want to have full control on how and when memory is allocated and deallocated. Moreover, since the allocated memory exists outside of the control of Pharo's garbage collector, external allocation avoids problems such as moving objects, explained later in this chapter.

Memory leaks in the C Heap

Allocating an external object using the message `externalNew` allocates the required memory on the C heap and returns to our Pharo program an `ExternalAddress` to that external memory. This `ExternalAddress` is the only reference to the the external memory.

In this setting, a memory leak can happen if our `ExternalAddress` object is garbage collected: the memory occupied by the `ExternalAddress` object is reclaimed, but the memory in the C heap remains allocated since there was no call to `free`.

Of course, an avid reader would ask herself *why not freeing the memory of an external address as soon as it is garbage collected?*. However, such automatic release cannot be done blindly for all `ExternalAddresses`. On the one hand, during the program execution an alias to the external memory can be created with a new `ExternalAddress` object, leading to two `ExternalAddresses` with the same value. In this situation, the first `free` will succeed, while the second one will cause a program failure. On the other hand, not all `Exter-`

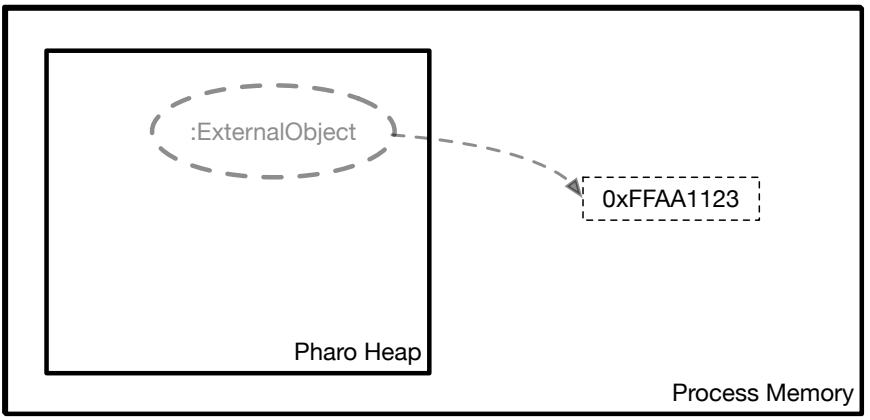

Figure 6-2 A garbage collected External Address creates a memory leak

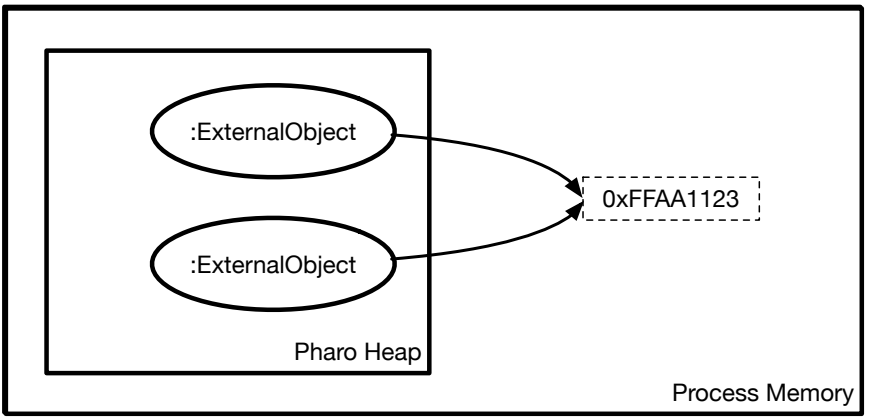

Figure 6-3 Two External Address referencing the same address are an alias: freeing one makes the second invalid

`nalAddress` objects contain address to externally allocated objects. Some `ExternalAddresses` may have offsets, arbitrary pointers or other kind of reference that we should not `free`.

The uFFI auto-release mechanism

For those external objects that we can automatically release on garbage collection, uFFI supports an *auto-release* feature which does what was described above: the memory of an external object is freed upon garbage collection. Using this feature requires a user to register an external object for auto-releasing by sending it the `autoRelease` message. By default, auto released

external objects will just call `free` on the reference they manage.

```
myStructure := MyStructure externalNew.
myStructure autoRelease.
" ... use my structure ... "
" ... dereference it so it will be collected ..."
myStructure := nil.
```

uFFI also supports extending the auto-release mechanism to implement our own. The first extension point is the class method `finalizeResourceData:` of external objects. User defined external objects can re-define the method `finalizeResourceData:` on the class side to control how its instances are deallocated. The default implementation looks like the following:

```
FFIExternalReference >> finalizeResourceData: handle [
    handle isNull ifTrue: [ ^ self ].
    handle free.
    handle beNull
]
```

Indeed, upon garbage collection `finalizeResourceData:` does not receive the (already garbage collected) external object but the handle it contained. Overriding this method allows users to, for example: - call library-specific free functions. For example, libraries such as SDL or libgit have their own free functions that correctly free their internal data structures, - do aditional Pharo-side cleaning. For example, unregistering the handle from some internal registry, and - do logging.

```
MyExternalStructure >> finalizeResourceData: handle [
   handle isNull ifTrue: [ ^ self ].

   "Logging the handle in the transcript for information"
   ('Freed ', handle asString) traceCr.

   handle free.
   handle beNull
]
```

uFFI provides in addition a second extension point for auto-release: `resourceData`. The method `resourceData` allows one to specify what data to store and send as argument on `finalizeResourceData:`. By default this method returns the handle of the external object.

```
FFIExternalReference >> resourceData [
  ^ self getHandle
]
```

However, we can modify both methods to have richer information at the time the resource data is finalized. For example, the SDL library bindings use as resource data an Array containing both the handle of its windows and also the window ID. When the external object is garbage collected, the method

`#finalizeResourceData:` receives the stored array and can act on it, as in the following example.

```
SDL_Window class >> resourceData [
  ^ {self getHandle. self windowID }
]

SDL_Window class >> finalizeResourceData: aTuple [
  | handle windowId |

  handle := aTuple first.
  handle isNull ifTrue: [ ^ self ].

  windowId := aTuple second.
  OSSDL2Driver current unregisterWindowWithId: windowId.
  self destroyWindow: handle.
  handle beNull
]
```

6.3 uFFI external objects in the Pharo Heap

We have seen in previous chapters that the different kind of external objects such as structures, arrays or unions, can be instantiated as normal objects using the `new` message. This makes external objects to be allocated in the Pharo heap.

```
myStructure := MyStructure new.
```

Allocating in the Pharo heap has a main advantage: we do not need to manually track the life-cycle of the object and use functions like C's `free()` to manually release it. Instead, the object and the storage it occupies will be released automatically by the garbage collector as soon as our Pharo program does not use it anymore, just like any other Pharo object.

```
myStructure := MyStructure new.
" ... use my structure ... "
" nil it and PLUF, eventually the object will be discarded "
myStructure := nil.
```

However, as objects in the Pharo heap and subject to the control of Pharo's garbage collector, the garbage collector can then wrongly decide to collect objects that, seemingly unused from Pharo, are used from a C library, or decide to move it, leaving a dangling pointer in the C library. This situation leads to memory corruptions and funny bugs.

The problem of garbage collection and C dangling pointers

As soon as we send an external object as the argument of a C function, the C function has the freedom to manipulate that external object as it pleases. In

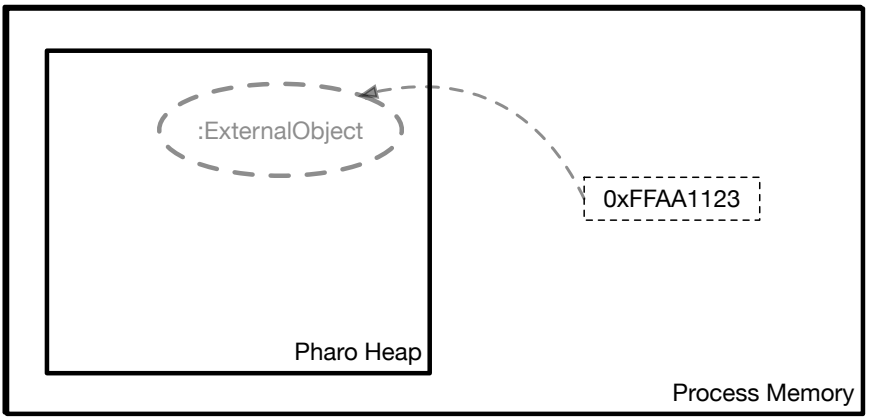

Figure 6-4 A dangling pointer is created when an external address points to a garbage collected object

particular, this can present some troubles when we send the object by reference using a pointer type, as illustrated in the following function binding:

```
FFITutorial >> myFunction: aStructure [
  ^ self ffiCall: #(void myFunction(MyStructure* aStructure))
]
```

Sending references to Pharo objects to external libraries introduces the problem of dangling references. A Pharo-allocated external object sent by reference sends the *address* of that object to the function. Such address is an unmanaged object address: it is supposed to reference an object, but since it is now on the C world, the garbage collector cannot update it in case the object is moved, and it cannot know if that address is in use. In case the garbage collector decides to move that object or collect it, the C library then finds itself using an address to a wrong object: a dangling pointer.

If we are lucky enough, dangling pointers will crash right away and we will realize the cause of it: they will try to access an object that is not there anymore and produce an error. However, it may happen that the dangling pointer references seemingly valid data. In that case, the execution of the program will continue for some more time, probably reading and writing wrong values, thus corrupting the memory. Such are the worst cases to debug, because the cause of the bug is far away from the symptoms we see. Moreover, there are no guarantees that concurrently the garbage collector and the C library access and modify the address at the same time. We will see in a future chapter that multi-threaded FFI weakens such guarantees even more.

To avoid such dangling pointers, no magic is available. Dangling pointers caused by garbage reclamation need to be avoided by users making sure their objects are not collected while in use. Dangling pointers caused by

moving objects can be solved by one of Pharo's runtime features that we will study in the next section: pinned objects.

Pinned Objects

Objects in use will not be garbage collected but may be moved in memory. To cope with this problem of moving objects, the Pharo runtime supports *pinned* objects. Pinned objects are objects that can be reclaimed but not moved by the garbage collector, avoiding the problem of *moving objects*. To pin an object in memory, we can use the message `pinInMemory`.

```
myStructure := MyStructure new.
myStructure pinInMemory.
" ... use safely my structure ... "
" nil it and PLUF, eventually the object will be discarded "
myStructure := nil.
```

If eventually we decide to unpin the object, we can do so by using `unpinInMemory`.

```
myStructure unpinInMemory.
```

For more fine grained control, external object also support the messages `isPinnedInMemory` and `setPinnedInMemory: aBoolean`. The former returns a boolean specifying wheter the object is pinned or not. The latter allows changing the pinned property with a boolean.

> **Note** Remember that pinning objects in memory do not prevent the garbage collector to reclaim those objects. Any pinned yet unused object will be garbage collected and may create memory corruptions too.

6.4 Conclusion

In this chapter we have studied the differences between memory allocated in Pharo and in C. We have see the problems that may arise by exchanging pointers between them, specially when we allocate external objects: memory leaks and dangling pointers.

Memory leaks are caused when we use external allocation and we do not correctly free this external memory on garbage collection. uFFI proposes an auto-release mechanism that can be extended in two (composable) ways to be able to free external memory when an external object is garbage collected.

Dangling pointers are caused when we send a pharo-allocated object reference to a C program, since the garbage collector can move or collect the object, and the reference living in the C side does not get updated. uFFI supports pinned objects to support with moved objects. However, it is the user's responsibility to avoid his objects from being collected if they are in usage.