Chapter 1

Calling Foreign Code with NativeBoost

Interacting with the external world is mandatory in modern systems. Even more, being able to call external libraries is often a key to business success.

To interface or bridge two systems, one could use network communications. This is a simple and fast to develop solution. However, this is not a solution to interact with external libraries such as OpenGL or Cairo. In such cases, one approach is to define a dedicated plugin for the Virtual Machine (VM). However, dealing with VM internals can be complex and does not offer important advantages over using a Foreign Function Interface (FFI).

Suppose you want to use the Cairo[1] library. The solution is to make a *binding* of each C library function to a Pharo method, and then to call these methods from Pharo code. But we should answer several questions: how does a binding call a C function? How do you write such bindings? How do we pass heap allocated objects? The libraries and mechanisms in charge of doing this kind of work are usually called FFI, for Foreign Function Interface.

In this chapter we will start by showing you how to use NativeBoost, a new foreign function library. NativeBoost is based on NativeBoost and previous FFI libraries. We will also expose some practical examples and how to solve common problems faced when communicating with other languages.

1.1 Getting Started

Right now to run NativeBoost, you will need:

[1] Cairo is a C library to compute 2D ...

- a VM that has the NativeBoost plugin. One can be found on the Pharo's continuous integration server[2].
- a Pharo image with the necessary packages. Execute the following code to load them:

Script 1.1: *Installing NativeBoost*

```
Gofer it
   squeaksource: 'NativeBoost';
   package: 'ConfigurationOfNativeBoost';
   load.
(Smalltalk at: #ConfigurationOfNativeBoost) load.
```

You can now check your installation: open the Test Runner from the World menu, select the NativeBoost-Tests package and run all tests.

1.2 Calling a Simple External Function

Suppose you want to know the amount of time the image has been running by calling the underlying OS function named clock. This function is part of the standard C library (`libc`). Its C declaration is:[3]

Script 1.2: *The C declaration of clock()*

```
uint clock (void)
```

To call clock from the image, you should write a binding: a normal Smalltalk method annotated with the <primitive:module:> pragma. This pragma specifies that a native call should be performed using the NativeBoost plugin. The external function call is described using the nbCall:module: message[4]. Here is the full definition of the binding of this clock function.

Script 1.3: *Defining our first binding*

```
CExamples class>>ticksSinceStart
 <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

 ^ self nbCall: #( uint clock () ) module: NativeBoost CLibrary
```

In the above example, we define a method named ticksSinceStart on the class side of the class named CExamples. NativeBoost imposes no constraint

[2] https://ci.lille.inria.fr/pharo/view/NativeBoost/

[3] According to man its return type is clock_t instead of uint, but we deliberately made it uint for the sake of simplicity. We will see how to deal with typedefs in the following sections.

[4] Note that it exists other messages to define callouts. They will be discussed later.

on the class where the method should be added and programmers can choose to add their bindings either on the instance- or the class-side.

To know the number of native clock ticks since the Pharo Virtual Machine started, execute the method and print its result:

Script 1.4: *Invoking a C binding*

```
CExamples ticksSinceStart
```

Analysis of an FFI Callout

Now that the call worked, let us look at the definition again:

```
CExamples class>>ticksSinceStart
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ^ self nbCall: #( int clock () ) module: NativeBoost CLibrary
```

Callout. What we have just done is usually named an FFI *callout*. `#nbCall:module:` is a method provided by the NativeBoost package and defined in the `Object` class (so you can use it anywhere). It is responsible for generating an FFI callout. Calling a C function requires to marshall all arguments from Smalltalk to C, push them to the C stack, perform a call to the external function, and finally convert the return value from C to Smalltalk. Note, that this message `#nbCall:module:` should only be sent by methods that define a callout, because it does some context handling behind the scenes.

`#nbCall:module:` handles all of this using the description of the function you provided. Indeed, it has two arguments: the *signature of the function* that will be invoked and the *module or library* where to look for it. In this example we look for `clock()` function in the standard C library.

The signature is described by an array: here `#(int clock ())`.

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

Basically, if you strip down outer #(), what is inside is a C function prototype, which is very close to normal C syntax. This is intentionally done so, that in most cases you can actually copy-and-paste a complete C function declaration, taken from header file or from documentation, and it is ready for use.

The second argument, #module:, should be a module name or its handle, where to look up the given function. NativeBoost provides a convenience method named CLibrary to obtain a handle of the standard C library. In other cases (suppose you want to use some standalone library), you must specify either a library name, or a full path to it, or an handle to the already loaded library. You must use NativeBoost methods to load an external library (and hence obtain handle on it) before making any calls to its functions. For example:

```
"Loads the module named 'nameOfModule' "
NativeBoost forCurrentPlatform loadModule: 'nameOfModule'
```

Note that module names are platform dependent, so for example to load the OpenGL library you may want to use 'OpenGL32' on windows and 'libGL.so' or even '/usr/lib/libGL.so' on linux.

About marshaling. In the clock callout example above, the return type is int and not SmallInteger. This is the same as with function arguments. They are all described in terms of C language. In general you don't have to worry too much about this, because the NativeBoost library knows how to map standard C values to Smalltalk objects and viceversa. So, when this message is executed, the return value will be converted into a SmallInteger. You can also define new mappings from C types to Smalltalk objects. This is useful when wrapping libraries that define new C types as we will show later.

This conversion process for types from different languages is called *marshalling*. We will see more examples of automatic conversions[5] in this Chapter.

Presentation Conventions

There are always three things to take into account when doing callouts: a C function, a specification of the binding between C and Pharo, and the location of the binding declaration at the Smalltalk level.

1. First, of course, is the signature of the C function we will call, this definition is the definition of the function from a C point of view. Through this book we describe it in a C header box as follows:

   C header.

   ```
   int clock (void)
   ```

[5] By automatic, we mean that they are already defined in the NativeBoost library.

2. Second, how do we communicate between the C and Smalltalk worlds. That's the purpose of the *binding* declaration. It includes the *Primitive pragma* which instructs the VM to actually "do the thing", when you run the corresponding Smalltalk method. Also, note, that there is no hidden complexity behind this primitive. It does only one thing: runs a generated machine code, while the code generation logic is at the image-level and ready for study and modification.

   Pragma declaration.

```
<primitive: #primitiveNativeCall module: #NativeBoostPlugin>
```

3. Third, we have to see how we organize the C calls from the Smalltalk side. Often we create a class that groups callout definitions. Such a class can then be used as an entry point to perform callouts.

   Callout grouping.

   CExamples ticksSinceStart

1.3 Passing arguments to a function

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

Passing a method parameter

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

C header.

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

Smalltalk binding.

```
CExamples class>>abs: anInteger
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ self nbCall: #( int abs (int anInteger) ) module: NativeBoost CLibrary
```

Compared to the previous example, we changed the name of the function and added an argument. When functions have arguments you have to specify two things for each of them: their type and the object that you want to be sent to the C function. That is, in the arguments array, we put the type of the argument (int in this example), and after that, the *name of the Smalltalk variable* we pass as argument.

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

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

Now you can try printing this:

CExamples abs: -20.

About arguments

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

Passing literals

Imagine that we want to have a wrapper that always calls the abs function with the number -45. Then we directly define it as follows:

```
CExamples class>>absMinusFortyFive
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ NBFFICallout nbCall: #( int abs (−45) ) module: NativeBoost CLibrary
```

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

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

```
CExamples class>>floor: aFloat
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ NBFFICallout nbCall: #( double floor(double aFloat) ) module: NativeBoost CLibrary
```

and then you can use this method as follows: `self mymethod: 4.5`

Luc ►*Examples with more complex literals: arrays, ...*◄

Passing variables

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

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

```
Object subclass: #MyClass
   ...
   classVariables: 'MyConstant'
..
```

Then don't forget to initialize it properly:

```
MyClass class>>initialize
   MyConstant := −45.
```

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

```
MyClass class>>absMinusFortyFive
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>
   ^ NBFFICallout nbCall: #( int abs (MyConstant) ) module: NativeBoost CLibrary
```

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

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

```
SmallInteger>>absoluteValue
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ NBFFICallout nbCall: #( int abs (int self) ) module: NativeBoost CLibrary
```

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

Passing strings

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

C header.

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

Smalltalk binding.

```
CExamples class>>stringLength: aString
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ^ NBFFICallout nbCall: #( int strlen (String aString) ) module: NativeBoost CLibrary
```

Callout grouping.

CExamples stringLength: 'awesome!'

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

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

- The first difference is the const keyword of the argument. For those not used to C, that's only a modifier keyword that the compiler takes into account to make some static validations at compile time. It has no value when describing the signature for calling a function at runtime.
- The second difference, an important one, is the specification of the argument. It is declared as String aString instead of char * aString. With String aString, NativeBoost will automatically do the arguments conversion from Smalltalk strings to C strings (null terminated). Therefore it is important to use String and not char *.

  In the example, the string passed will be put in an external C char array and a null termination character will be added to it. Also, this array will be automatically released after the call ends. This automatic memory management is very useful but we can also control it as we will see later.

Using (String aString) is equivalent to (someString copyWith: (Character value:0) as in CExamples stringLength: (someString copyWith: (Character value:0). Conversely, NativeBoost will take the C result value of calling the C function and convert it to a proper Smalltalk Integer in this particular case.

Passing two strings

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

C header.

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

Smalltalk binding.

```
CExamples class>>stringCompare: aString with: anotherString
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ^ NBFFICallout nbCall: #( int strcmp (String aString, String anotherString) )
    module: NativeBoost CLibrary
```

Callout grouping.

CExamples stringCompare: 'awesome!' with: 'awesome'

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

1.4 Getting return value from a function

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

We already saw that implicitly through multiple examples since the beginning of the chapter. For example in the abs example, the result is converted from an int to a SmallInteger. In the floor example, the result is converted from a double to a Float. Luc *►verify return types. verify auto-conversions between SmallInteger <-> LargeInteger. Example: abs(-16rFFFFFFFF) where -16rFFFFFFFF is a LargeNegativeInteger whereas we defined the parameter as an int ◄*

Luc *►find an example: a simple function that returns a pointer type◄*

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

Luc ►*how to deal with the result void*?*◄

1.5 About Heap Memory

Passing a Smalltalk object by reference

So far we only used objects that were taken from the Smalltalk heap and pushed into the C stack (doing the corresponding marshaling). That means that until now we only passed copies of the arguments, or that we passed them *by-value*. With the next example we will show how NativeBoost supports to pass objects *by-reference*, so that they can be modified by the external world. Stéf ►*here*◄ The next example adds a bit more complexity: we are going to send a ByteArray to the memset function. memset fills an array with one value (literally man defines it as: "fill a byte string with a byte value").

C header.

```
void * memset ( void * ptr, int value, int num );
```

Sets the first num bytes of the block of memory pointed by ptr to the specified value

Smalltalk binding.

```
CExamples class>>memorySet: aByteArray with: anInteger for: aByteCount
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ^ NBFFICallout nbCall: #( void* memset (char* aByteArray, int anInteger, int
    aByteCount) ) module: NativeBoost CLibrary
```

Callout grouping.

```
array := ByteArray new: 10.
CExamples memorySet: array with: 100 for: 5.
array
```

As you should notice, this time NativeBoost takes care of finding out the memory address where aByteArray lives and pushes a pointer to it as the first argument of memset. After executing the example you can see that the byte array elements were actually changed.

Then, there is an important difference between passing arrays and passing strings. The expert user would notice that string functions in C language

expects to receive a pointer to contiguous memory bytes ending with a null character to delimit its length. NativeBoost is responsible for copying the string to a bigger buffer and inserting this special end character. This means that Smalltalk String objects are passed by-copy, while on the other hand byte arrays are passed by reference!

Be careful with ownership-type pointers

You also may notice in this last example that we are sending a regular Smalltalk object handle to a native library. Indeed, the ByteArray objet lives in the Smalltalk's memory. NativeBoost allows you to pass an objet handle as a pointer argument to a C function. We call ownership-type pointers these kind of pointers whose values are addresses in the Smalltalk memory.

But there are some kind of C functions where it is dangerous or impossible to do that. In such cases, no error or warning will happen, you will probably just get fatal consequences (memory corruption/segmentation fault followed by imminent crash). That is why it is important to understand these potential problems.

A first problem comes from the Garbage Collector (GC). It is responsible to auto cleaned the Smalltalk's memory and potentially moves objects to other places. So, if we pass an ownership-type pointer to an external function and hen the GC runs. During GC compaction phase any object in object memory can be relocated into different memory region. Finally, when the external library will try to access memory (read/write) using the ownership-type pointer, it will do it at previous object location, that results in fatal consequences. This problem must be addressed by a FFI that supports asynchronous calls because the GC may run during callouts. We will talk about asynchronous callouts later in this chapter.

Another problem is when a function which stores (remembers) the ownership-type pointer value somewhere for later use. Since usually we cannot tell if an external library will store such a pointer, we cannot tell it to patch the pointer value each time the referenced object is relocated by the GC in the object memory. To know if a function takes an ownership on some pointer, which is passed as argument to it, refer to function documentation or check its implementation (if it is available).

To sum up, this method of passing object references (pointers to the Smalltalk's heap) to external functions is easy and simple because it is automatically handled by NativeBoost. Nevertheless, it is dangerous and we have to be very careful when dealing with C pointer arguments to the Smalltalk's memory.

If you're not sure, the rule is to avoid passing pointers to your data like byte arrays, residing in object memory to external functions. The alterna-

tive solution is to always use pointers to externally allocated memory as presented in the next section.

External heap management

NativeBoost provides some external heap management functionalities, including:

- allocating/deallocating external memory (allocate: and free:)
- copying data between two memory locations (memCopy:to:size:)

Luc ►*find a real example and show the binding of* externalFunction: *with the types.*◄

Let's see a general example.

```
"Allocate memory in the external heap"
mem := NativeBoost allocate: 10.

"Allocate memory a ByteArray in the Smalltalk's heap"
bytes := #[1 2 3 4 5 6 7 8 9 10] copy.

"Copy data from Smalltalk heap to external heap"
NativeBoost memCopy: bytes to: mem  size: 10.

"Call external function passing data in the external heap"
self externalFunction: mem.

"Recopy data from the external heap to the Smalltalk's one"
NativeBoost memCopy: mem to: bytes size: 10.

"Free external heap memory"
NativeBoost free: mem.
```

Note, that external memory must be freed explicitly (using free:) since there is no garbage collection to clean it up. So, if you forget to free the external memory, this forms a resource leak, and eventually, if you do this repeatedly, you will no longer be able to allocate new memory, because there is no free memory left.

1.6 Using C structures

external object (automatically has a handle) CairoSurface inherits from NBExternalObject (which manages the opaque handle to the CairoSurface C structure then

```
create
  <>
  call: CairoSurface  fct
```

NBExternalObjectType is responsible for the generation of the code

1.7 Callbacks

1.8 Loading code from other libraries

1.9 Asynchronous FFI

1.10 Todo

- default options **Luc** ►*what does it mean?*◄
- calling conventions: cdecl...

1.11 Behind the scenes: NativeBoost naked!

1.12 NativeBoost Executive Summary

Here we say everything again but in a concise way. **Javier** ►*Ha!*◄

1.13 Wrapping Cairo library

Cairo is a drawing library written in C. Here we present the basis of writing a wrapper for it, which should be complete enough to write any wrapper for any C library.

Small example

Whenever you want to start writing a wrapper, you should do it this way: find the smallest piece of code you can find that uses that library, and wrap everything you need to make it work from Smalltalk. Then you can go on adding more and more to the wrapped functionality.

In this example we have a minimal program that uses Cairo library, taken from Cairo'sFAQhttp://cairographics.org/FAQ/ It creates a Cairo surface, puts some text on it and renders everything to a PNG.

```
#include <cairo.h>

int
main (int argc, char *argv[])
{
	cairo_surface_t *surface =
	  cairo_image_surface_create (CAIRO_FORMAT_ARGB32, 240, 80);
	cairo_t *cr =
	  cairo_create (surface);

	cairo_select_font_face (cr, "serif", CAIRO_FONT_SLANT_NORMAL,
	 CAIRO_FONT_WEIGHT_BOLD);
	cairo_set_font_size (cr, 32.0);
	cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
	cairo_move_to (cr, 10.0, 50.0);
	cairo_show_text (cr, "Hello, world");

	cairo_destroy (cr);
	cairo_surface_write_to_png (surface, "hello.png");
	cairo_surface_destroy (surface);
	return 0;
}
```

The way we will work then is to wrap everything we find in our way from the first to the last line. In order to wrap a library, you should get its reference documentation, so that you know exactly which are its structures, and functions. We start by creating some classes for all Cairo structures that are going to be used here and there in the wrapper.

We create a class for all the library's constants (we will fill them later):

Class 1.5:

```
SharedPool subclass: #NBCairoConstants
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo-Core'
```

and a class for all the library's typedefs and enums (we will also fill them later):

Class 1.6:

```
SharedPool subclass: #NBCairoTypes
  instanceVariableNames: ''
```

```
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo-Core'
```

Notice that both of them are Shared Pools. We now create a base class for structures used in the wrapper, putting the shared pools, so that we can use all of them in the derived classes.

Class 1.7:

```
NBExternalObject subclass: #NBCairoObject
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: 'NBCairoConstants NBCairoTypes'
  category: 'NBCairo-Core'
```

Finally, to wrap the first line of main function we will create a class to represent a Cairo surface.

Class 1.8:

```
NBCairoObject subclass: #NBCairoSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo-Core'
```

With most well written libraries, the contents of the structures they declare is private to their internals. They are opaque, in the sense that you may not know nor care what bits they have inside, because you only change them through calls to the library. For that kind of structures, NativeBoost has the NBExternalObject. Our wrapping classes inherit from NBExternalObject, which is made for this situation, where we don't need to fill the fields of the structs.

Now, NBCairoSurface is described as an abstract structure, there are different concrete ones, like the image surface, so we add it as a subclass.

Class 1.9:

```
NBCairoSurface subclass: #NBCairoImageSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo-Core'
```

Then we can finally create a factory method, to wrap the `cairo_image_surface_create` function. We go to the class side and we enter:

```
NBCairoImageSurface class>>create: aFormat width: aWidth height: aHeight

   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ self nbCall: #(NBCairoImageSurface cairo_image_surface_create (int aFormat,
                                          int aWidth,
                                          int aHeight) )
```

and of course, we shouldn't forget to wrap the code that destroys the instance on the C side, but this time on the instance side of the NBCairoSurface class:

```
NBCairoSurface>>destroy
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ self nbCall: #(void cairo_surface_destroy (NBCairoSurface self) )
```

In the creation method, we declare the function as returning a NBCairoImageSurface, but actually it is returning a cairo_surface_t pointer. The same happens on the destroy method which is described as an NBCairoSurface, but actually is a cairo_surface_t pointer. In both cases, NativeBoost takes care of the conversion.

These methods are almost working. To finish, we add some helper functions to the NBCairoObject class, the one which was going to contain some useful configuration:

```
NBCairoObject class>>nbLibraryNameOrHandle
   ^NativeBoost forCurrentPlatform loadModule: '/usr/lib/libcairo.so.2'

NBCairoObject>>nbLibraryNameOrHandle
   ^self class nbLibraryNameOrHandle
```

Now to be able to test a bit, we have to add some more things. We add cairo_status_t (which is an enum) and cairo_surface_t to the NBCairoTypes.

Class 1.10:

```
SharedPool subclass: #NBCairoTypes
   instanceVariableNames: ''
   classVariableNames: 'cairo_status_t cairo_surface_t '
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

```
NBCairoTypes class>>initialize
  "self initialize"

  cairo_status_t := #int.
  cairo_surface_t  := #NBCairoSurface.
```

and we doit:

NBCairoTypes initialize

so that cairo_status_t and cairo_surface_t get initialized. Then, we add a method to the NBCairoSurface class:

```
NBCairoSurface>>status
 <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

 ^ self nbCall: #(cairo_status_t cairo_surface_status (cairo_surface_t self) )
```

Note what we just did: we added two types mappings, cairo_status_t that maps to int, and cairo_surface_t that maps to NBCairoSurface. That way we can now use this mappings in the function specifications, getting closer to copy-pasting the functions definitions into the binding.

We are now ready to test just a bit. Take a second and save the image now. Remember that if you wrongly specify the parameters of an FFI call, you could crash the image.

To test that everything is working do print this in a workspace:

surface := NBCairoImageSurface new. surface status

If everything is fine, status should be 0. Then don't forget to destroy the surface:

surface destroy

To make the example complete, let's add the final pieces needed.

Here is how the original example will look after translating it to use the wrapper:

```
CairoExamples>>minimalProgram
 "
 self new minimalProgram
 "

 | surface context fileName |
 surface := NBCairoImageSurface create: CAIRO_FORMAT_ARGB32 width: 240 height:
       80.
 context := surface createContext.
 context selectFont: 'serif' slant: CAIRO_FONT_SLANT_NORMAL weight:
      CAIRO_FONT_WEIGHT_BOLD;
    setFontSize: 32.0;
    setSourceR: 0.0 G: 0.0 B: 1.0;
    moveToX: 10.0 Y: 50.0;
    showText: 'Hello, world';
    destroy.
```

```
fileName := (FileDirectory default / 'Hello.png') fullName.
surface writeToPng: fileName.
surface destroy.

Form openImageInWindow: fileName
```

but before it works we have to add some constants and types. CAIRO_FORMAT_ARGB32 is a constant defined on cairo_format_t, and we are using constants for font slants and font weights too. For each enum we add a type on NBCairoTypes and its constants to NBCairoConstants:

```
SharedPool subclass: #NBCairoConstants
   instanceVariableNames: ''
   classVariableNames: 'CAIRO_FONT_SLANT_ITALIC
      CAIRO_FONT_SLANT_NORMAL CAIRO_FONT_SLANT_OBLIQUE
      CAIRO_FONT_WEIGHT_BOLD CAIRO_FONT_WEIGHT_NORMAL
      CAIRO_FORMAT_A1 CAIRO_FORMAT_A8 CAIRO_FORMAT_ARGB32
      CAIRO_FORMAT_INVALID CAIRO_FORMAT_RGB16_565
      CAIRO_FORMAT_RGB24'
   poolDictionaries: ''
   category: 'NBCairo-Core'

SharedPool subclass: #NBCairoTypes
   instanceVariableNames: ''
   classVariableNames: 'cairo_font_slant_t cairo_font_weight_t cairo_status_t
      cairo_surface_t cairo_t'
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

and then we add the code to initialize these variables:

```
NBCairoTypes class>>initialize
  "self initialize"

  cairo_status_t := cairo_font_slant_t := cairo_font_weight_t := #int
  cairo_t := #NBCairoContext.
  cairo_surface_t  := #NBCairoSurface.


NBCairoConstants class>>initialize
  "self initialize"

  "enum cairo_format_t"
  CAIRO_FORMAT_INVALID :=  -1.
  CAIRO_FORMAT_ARGB32 :=  0.
  CAIRO_FORMAT_RGB24 := 1.
  CAIRO_FORMAT_A8 := 2.
  CAIRO_FORMAT_A1 := 3.
```

```
CAIRO_FORMAT_RGB16_565 := 4.

"enum cairo_font_slant_t"
CAIRO_FONT_SLANT_NORMAL := 0.
CAIRO_FONT_SLANT_ITALIC := 1.
CAIRO_FONT_SLANT_OBLIQUE := 2.

"enum _cairo_font_weight_t"
CAIRO_FONT_WEIGHT_NORMAL := 0.
CAIRO_FONT_WEIGHT_BOLD := 1.
```

don't forget to doit:

NBCairoConstants initialize

and

NBCairoTypes initialize

As a last step, you need to wrap the remaining functions. We leave here a last example and the rest of them you can take the code from the NBCairo repository on squeaksource.

Class 1.11:

```
NBCairoSurface subclass: #NBCairoImageSurface
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

```
CairoContext>>selectFont: aFamily slant: slant weight: aWeight
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ self nbCall: #(void cairo_select_font_face (cairo_t self,
                                                 String aFamily,
                                                 cairo_font_slant_t slant,
                                                 cairo_font_weight_t aWeight) )
```

Now as a general principle it is better not to manipulate all the variables in your program. Why first because the call out will be slowler and second because it is better to encapsulate all these constants. **Javier** ►*Explain more!*◄

So the solution is to do something like @@@to be fixed@@

```
AthensCairoSurface>>primWidth: aWidth height: aHeight
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ^ self nbCall: #(AthensCairoSurface cairo_image_surface_create(
            CAIRO_FORMAT_ARGB32,
```

```
        int aWidth,
        int aHeight) )
```