Nim in Action

by Dominik Picheta

Chapter 8

brief contents

8 Interfacing with other languages

This chapter covers

- Getting to know Nim's foreign function interface
- Distinguishing between static and dynamic linking
- Creating a wrapper for an external C library
- Using the JavaScript backend
- Wrapping JavaScript APIs

For many years, computer programmers have been writing software libraries in various programming languages. Many of these libraries have been in development for a very long time, accumulating features and maturing over the years. These libraries are not typically written in Nim; instead, they've been written in older programming languages such as C and C++.

When writing software, you might have required an external C library to perform a task. A good example of this is the OpenSSL library, which implements the SSL and TLS protocols. It's primarily used for securely transferring sensitive data over the internet, such as when navigating to a website using the HTTPS protocol.

Many of the HTTP client modules in the standard libraries of various programming languages, including Nim's, use the C library to transfer encrypted data to and from HTTP servers securely. It's easy to forget that this library is used, because it's usually invoked behind the scenes, reducing the amount of work the programmer needs to do.

The Nim standard library takes care of a lot of things for you, including interfacing with other languages, as is the case with the OpenSSL library. But there will be times when you'll need to interface with a library yourself.

This chapter will prepare you for those times. First, you'll learn how to call procedures implemented in the C programming language, passing data to those procedures and receiving data back from them. Then, you'll learn how to wrap an external library called SDL, and you'll use your wrapper to create a simple SDL application that draws on the screen. (A wrapper is a thin layer of code that acts as a bridge between Nim code and a library written in another programming language, such as C.) Last, you'll work with the JavaScript backend, wrapping the Canvas API and drawing shapes on the screen with it.

Nim makes the job of calling procedures implemented in the C programming language particularly easy. That's because Nim primarily compiles to C. Nim's other compilation backends, including C++, Objective-C, and JavaScript, make using libraries written in those languages easy as well.

8.1 Nim's foreign function interface

Nim's foreign function interface (FFI) is the mechanism by which Nim can call procedures written in another programming language. Most languages offer such a mechanism, but they don't all use the same terminology. For example, Java refers to its FFI as the Java Native Interface, whereas Common Language Runtime languages such as C# refer to it as P/Invoke.

In many cases, the FFI is used to employ services defined and implemented in a lower-level language. This lower-level language is typically C or C++, because many important OS services are defined using those languages. Nim's standard library makes extensive use of the FFI to take advantage of OS services; this is done to perform tasks such as reading files or communicating over a network.

In recent years, the web has become a platform of its own. Web browsers that retrieve and present web pages implement the JavaScript programming language, allowing complex and dynamic web applications to be run inside the browser easily. In order to run Nim applications in a web browser and make use of the services provided by the browser, like the DOM or WebGL, Nim source code can be compiled to JavaScript. Accessing those services and the plethora of JavaScript libraries is also done via the FFI. Figure 8.1 shows an overview of Nim's FFI.

It's important to note that the FFI allows you to interface with C, C++, and Objective-C libraries in the same application, but you can't interface with both C and JavaScript libraries at the same time. This is because C++ and Objective-C are both backward compatible with C, whereas JavaScript is a completely different language.

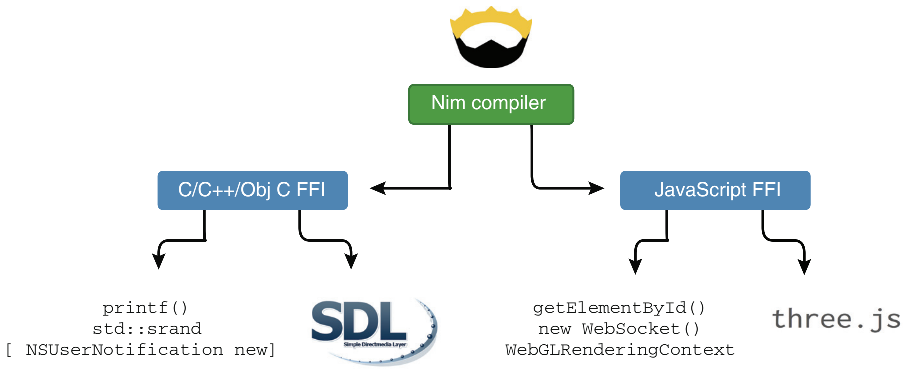


Figure 8.1 Using the Nim FFI, you can take advantage of services in other languages. Nim offers two versions of the FFI: one for C, C++, and Objective-C; and a second one for JavaScript. Both can't be used in the same application.

8.1.1 Static vs. dynamic linking

Before looking at the FFI in more detail, let's look at the two different ways that C, C++, and Objective-C libraries can be linked to your Nim applications.

When using an external library, your application must have a way to locate it. The library can either be embedded in your application's binary or it can reside somewhere on the user's computer. The former refers to a *statically linked library*, whereas the latter refers to a *dynamically linked library*.

Dynamic and static linking are both supported, but dynamic linking is favored by Nim. Each approach has its advantages and disadvantages, but dynamic linking is favored for several reasons:

- Libraries can be updated to fix bugs and security flaws without updating the applications that use the libraries.
- A development version of the linked library doesn't need to be installed in order to compile applications that use it.
- A single dynamic library can be shared between multiple applications.

The biggest advantage of static linking is that it avoids dependency problems. The libraries are all contained in a single executable file, which simplifies the distribution and installation of the application. Of course, this can also be seen as a disadvantage, because these executables can become very big.

Dynamically linked libraries are instead loaded when the application first starts. The application searches special paths for the required libraries, and if they can't be found, the application fails to start. Figure 8.2 shows how libraries are loaded in statically and dynamically linked applications.

It's important to be aware of the dynamically linked libraries that your application depends on, because without those libraries, it won't run.

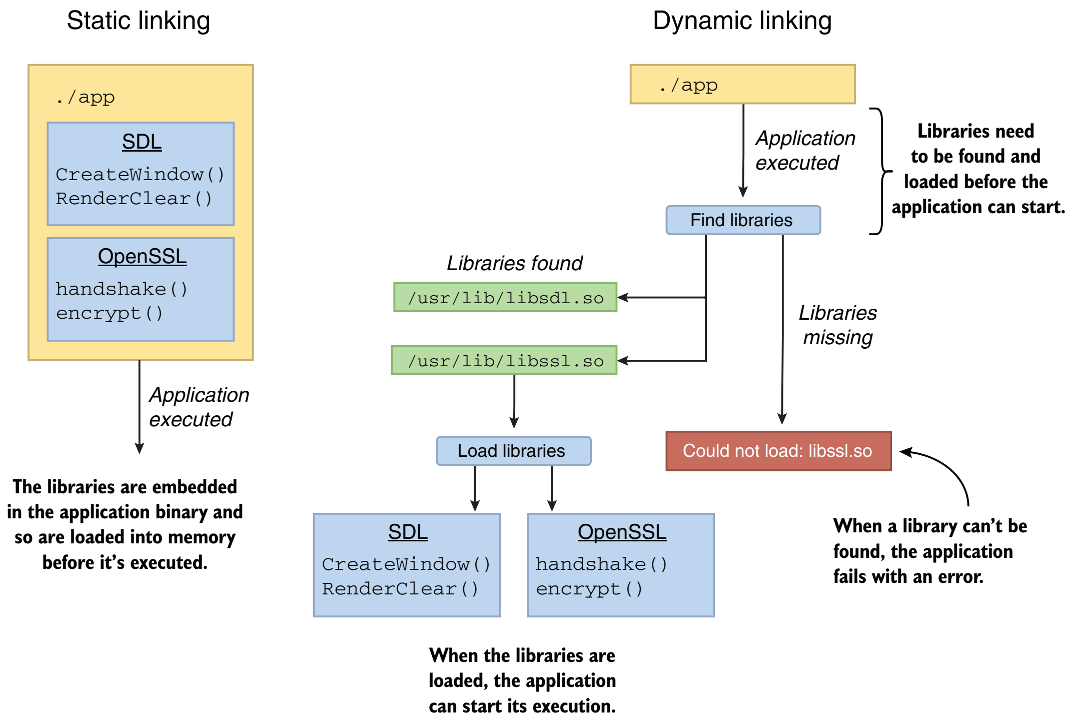

Static linking
./app
SDL
CreateWindow()
RenderClear()
OpenSSL
handshake()
encrypt()
Application executed
The libraries are embedded in the application binary and so are loaded into memory before it's executed.
Dynamic linking
./app
Application executed
Libraries need to be found and loaded before the application can start.
Find libraries
Libraries found
/usr/lib/libsdl.so
/usr/lib/libssl.so
Libraries missing
Load libraries
Could not load: libssl.so
SDL
CreateWindow()
RenderClear()
OpenSSL
handshake()
encrypt()
When a library can't be found, the application fails with an error.
When the libraries are loaded, the application can start its execution.

Figure 8.2 Static vs. dynamic linking

With these differences in mind, let's look at the process of creating wrappers in Nim.

8.1.2 *Wrapping C procedures*

In this section, we'll wrap a widely used and fairly simple C procedure: `printf`. In C, the `printf` procedure is declared as follows:

```
int printf(const char *format, ...);
```

What you see here is the procedure prototype of `printf`. A *prototype* specifies the procedure's name and type signature but omits its implementation. When wrapping procedures, the implementation isn't important; all that matters is the procedure prototype. If you're not familiar with this procedure, you'll find out what it does later in this section.

In order to wrap C procedures, you must have a good understanding of these procedure prototypes. Let's look at what the previous procedure prototype tells us about `printf`. Going from left to right, the first word specifies the procedure's return type, in this case an `int`. The second specifies the procedure name, which is `printf`. What follows is the list of parameters the procedure takes, in this case a `format` parameter of type `const char *` and a variable number of arguments signified by the ellipsis.

Table 8.1 summarizes the information defined by the `printf` prototype.

Table 8.1 Summary of the `printf` prototype

Return type	Name	First parameter type	First parameter name	Second parameter
`int`	`printf`	`const char *`	`format`	Variable number of arguments

This prototype has two special features:

- The `const char *` type represents a pointer to an immutable character.
- The function takes a variable number of arguments.

In many cases, the `const char *` type represents a `string`, as it does here. In C, there's no `string` type; instead, a pointer that points to the start of an array of characters is used.

When wrapping a procedure, you need to look at each type and find a Nim equivalent. The `printf` prototype has two argument types: `int` and `const char *`. Nim defines an equivalent type for both, `cint` and `cstring`, respectively. The `c` in those types doesn't represent the C programming language but instead stands for *compatible*; the `cstring` type is therefore a *compatible string* type. This is because C isn't the only language supported by Nim's FFI. The `cstring` type is used as a native JavaScript string as well.

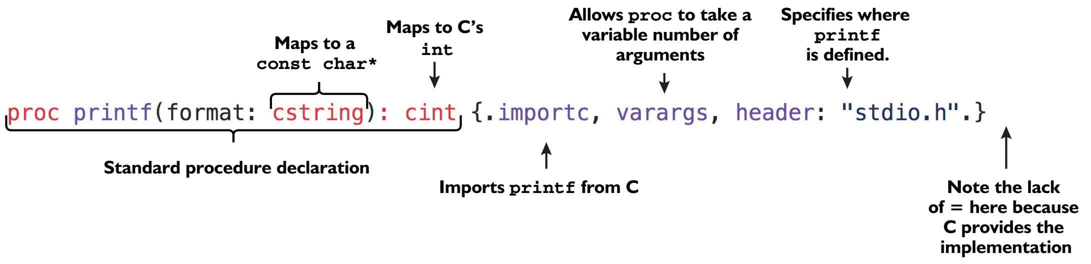
These compatible types are defined in the implicitly imported `system` module, where you'll find a lot of other similar types. Here are some examples:

- `cstring`
- `cint`, `cuint`
- `pointer`
- `clong`, `clonglong`, `culong`, `culonglong`
- `cchar`, `cschar`, `cuchar`
- `cshort`, `cushort`
- `cint`
- `csize`
- `cfloat`
- `cdouble`, `clongdouble`
- `cstringArray`

Let's put all this together and create the wrapper procedure. Figure 8.3 shows a wrapped `printf` procedure.

The following code shows how the procedure can be invoked:

```
proc printf(format: cstring): cint {.importc, varargs, header: "stdio.h".}
discard printf("My name is %s and I am %d years old!\n", "Ben", 30)
```



Maps to a const char*
Maps to C's int
Allows proc to take a variable number of arguments
Specifies where printf is defined.
proc printf(format: cstring): cint {.importc, varargs, header: "stdio.h".}
Standard procedure declaration
Imports printf from C
Note the lack of = here because C provides the implementation

Figure 8.3 `printf` wrapped in Nim

Save the preceding code as ffi.nim. Then compile and run it with `nim c -r ffi.nim`. You should see the following output:

```
My name is Ben and I am 30 years old!
```

The `printf` procedure takes a string constant, `format`, that provides a description of the output. It specifies the relative location of the arguments to `printf` in the `format` string, as well as the type of output that this procedure should produce. The parameters that follow specify what each format specifier in the `format` string should be replaced with. The procedure then returns a count of the printed characters.

One thing you might immediately notice is the `discard` keyword. Nim requires return values that aren't used to be explicitly discarded with the `discard` keyword. This is useful when you're working with procedures that return error codes or other important pieces of information, where ignoring their values may lead to issues. In the case of `printf`, the value can be safely discarded implicitly. The `{.discardable.}` pragma can be used for this purpose:

```
proc printf(format: cstring): cint {.importc, varargs, header: "stdio.h",
                                    discardable.}

printf("My name is %s and I am %d years old!\n", "Ben", 30)
```

What really makes this procedure work is the `importc` and `header` pragmas. The `header` pragma specifies the header file that contains the imported procedure. The `importc` pragma asks the Nim compiler to import the `printf` procedure from C. The name that's imported is taken from the procedure name, but it can be changed by specifying a different name as an argument to the `importc` pragma, like so:

```
proc displayFormatted(format: cstring): cint {.importc: "printf", varargs,
                                              header: "stdio.h", discardable.}

displayFormatted("My name is %s and I am %d years old!\n", "Ben", 30)
```

That's pretty much all there is to it. The `printf` procedure now wraps the `printf` procedure defined in the C standard library. You can even export it and use it from other modules.

8.1.3 Type compatibility

You may wonder why the `cstring` and `cint` types need to be used in the `printf` procedure. Why can't you use `string` and `int`? Let's try it and see what happens.

Modify your ffi.nim file so that the `printf` procedure returns an `int` type and takes a `string` type as the first argument. Then, recompile and run the program.

The program will likely show no output. This underlines the unfortunate danger that comes with using the FFI. In this case, the procedure call does nothing, or at least it appears that way. In other cases, your program may crash. The compiler trusts you to specify the types correctly, because it has no way of inferring them.

Because the `cstring` type was changed to the `string` type, your program now passes a Nim string object to the C `printf` procedure. C expects to receive a `const char*` type, and it always assumes that it receives one. Receiving the wrong type can lead to all sorts of issues, one of the major ones being memory corruption.

Nim's string type isn't as simple as C's, but it is similar. A Nim string is an object that contains two fields: the length of the string and a pointer to an array of `char`s. This is why a Nim string can be easily converted to a `const char*`. In fact, because this conversion is so easy, it's done implicitly for you, which is why, even when you pass a `string` to `printf`, which expects a `cstring`, the example compiles.

> **CONVERSION FROM CSTRING TO STRING** A conversion in the other direction, from a `cstring` to a `string`, is not implicit because it has some overhead. That's why you must do it explicitly using a type conversion or the `$` operator.

As for the `cint` type, it's very similar to the `int` type. As you'll see in the Nim documentation, it's actually just an alias for `int32`: http://nim-lang.org/docs/system.html#cint. The difference between the `int` type and the `int32` type is that the former's bit width depends on the current architecture, whereas the bit width of the latter type is always 32 bits.

The `system` module defines many more compatibility types, many of which are inspired by C. But there will come a time when you need to import types defined in C as well. The next section will show you how that can be done.

8.1.4 Wrapping C types

The vast majority of the work involved in interfacing with C libraries involves wrapping procedures. Second to that is wrapping types, which we'll look at now.

In the previous section, I showed you how to wrap the `printf` procedure. In this section, you'll see how to wrap the `time` and `localtime` procedures, which allow you to retrieve the current system time in seconds and to convert that time into calendar time, respectively. These procedures return two custom types that need to be wrapped first.

Let's start by looking at the `time` procedure, which returns the number of seconds since the UNIX epoch (Thursday, 1 January 1970). You can look up its prototype online. For example, C++ Reference (http://en.cppreference.com/w/c/chrono/time) specifies that its prototype looks like this:

```
time_t time( time_t *arg );
```

Further research into the type of time_t indicates that it's a *signed integer*.[1] That's all you need to know in order to declare this procedure in Nim. The following listing shows this declaration.

Listing 8.1 Wrapping time

The CTime type is the wrapped version of time_t, defined as a simple alias for a 64-bit signed integer.

The time C procedure is defined in the <time.h> header file. To import it, the header pragma is necessary.

```
type
  CTime = int64

proc time(arg: ptr CTime): CTime {.importc, header: "<time.h>".}
```

In this case, you wrap the time_t type yourself. The procedure declaration has an interesting new characteristic. It uses the ptr keyword to emulate the time_t * type, which is a pointer to a time_t type.

To convert the result of time into the current hour and minute, you'll need to wrap the localtime procedure and call it. Again, the specification of the prototype is available online. The C++ Reference (http://en.cppreference.com/w/c/chrono/localtime) specifies that the prototype looks like this:

```
struct tm *localtime( const time_t *time );
```

The localtime procedure takes a pointer to a time_t value and returns a pointer to a struct tm value. A struct in Nim is equivalent to an object. Unfortunately, there's no way to tell from the return type alone whether the struct that the localtime returns has been allocated on the stack or on the heap.

Whenever a C procedure returns a pointer to a data structure, it's important to investigate whether that pointer needs to be manually deallocated by your code. The documentation for this procedure states that the return value is a "pointer to a static internal tm object." This means that the object has a *static* storage duration and so doesn't need to be deallocated manually. Every good library will state the storage duration of an object in its documentation.

When wrapping code, you'll undoubtedly run into a procedure that returns an object with a *dynamic* storage duration. In that case, the procedure will allocate a new object every time it's called, and it's your job to deallocate it when you no longer need it.

DEALLOCATING C OBJECTS The way in which objects created by a C library can be deallocated depends entirely on the C library. A free function will usually be offered for this purpose, and all you'll need to do is wrap it and call it.

The struct tm type is much more complex than the time_t type. The documentation available in the C++ Reference (http://en.cppreference.com/w/c/chrono/tm) shows

[1] The type of time_t is described in this Stack Overflow answer: http://stackoverflow.com/a/471287/492186.

that it contains nine integer fields. The definition of this type in C would look something like this:

```
struct tm {
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
   int tm_wday;
   int tm_yday;
   int tm_isdst;
};
```

Wrapping this type is fairly simple, although a bit mundane. Fortunately, you don't have to wrap the full type unless you need to access all the fields. For now, let's just wrap the `tm_min` and `tm_hour` fields. The following listing shows how you can wrap the `tm` type together with the two fields.

Listing 8.2 Wrapping `struct tm`

```
type
  TM {.importc: "struct tm", header: "<time.h>".} = object
    tm_min: cint
    tm_hour: cint
```

The struct keyword can't be omitted in the argument to the pragma.

The two fields are defined as they would be for any Nim data type. The cint type is used because it's compatible with C.

You can then wrap the `localtime` procedure and use it together with the `time` procedure as follows.

Listing 8.3 A complete `time` and `localtime` wrapper

```
type
  CTime = int64

proc time(arg: ptr CTime): CTime {.importc, header: "<time.h>".}

type
  TM {.importc: "struct tm", header: "<time.h>".} = object
    tm_min: cint
    tm_hour: cint

proc localtime(time: ptr CTime): ptr TM {.importc, header: "<time.h>".}

var seconds = time(nil)
let tm = localtime(addr seconds)
echo(tm.tm_hour, ":", tm.tm_min)
```

Passes the address of the seconds variable to the localtime procedure

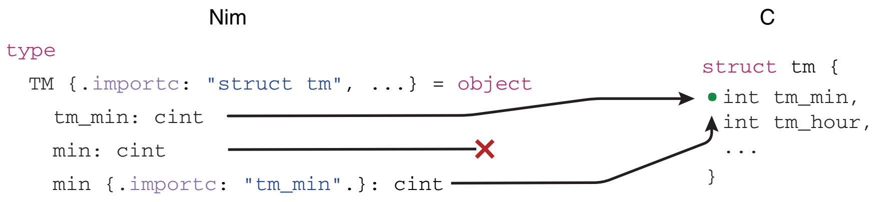
Assigns the result of the time call to a new seconds variable. The time procedure can also optionally store the return value in the specified argument; nil is passed here, as it's not needed.

Displays the current time

The localtime procedure takes a "time_t *" and returns a "struct tm *", both of which are pointers. That is why the ptr keyword is used.

Save this code as ffi2.nim, and then compile and run it. You should see the current time displayed on your screen after execution, such as `18:57`.

The main takeaway from the example in listing 8.3 is that wrapping a type essentially involves copying its structure into a Nim type definition. It's important to remember that the field names have to match those of the C type. You can specify the name of each field in an `importc` pragma if you wish to rename them. Figure 8.4 demonstrates this.



```
type
  TM {.importc: "struct tm", ...} = object
    tm_min: cint
    min: cint
    min {.importc: "tm_min".}: cint
```

```
struct tm {
  int tm_min,
  int tm_hour,
  ...
}
```

Figure 8.4 The mapping between fields in a wrapped type and a C struct

Another interesting aspect of wrapping the `localtime` procedure is the need to pass a pointer to it. You need to account for this in your wrapper. The `addr` keyword returns a pointer to the value specified, and that value must be mutable, which is why the return value of `time` is assigned to a new `seconds` variable in listing 8.3. Writing `localtime(addr time(nil))` wouldn't work because the return value isn't stored anywhere permanent yet.

You should now have a pretty good idea of how C types can be wrapped in Nim. It's time to wrap something a little more ambitious: an external library.

8.2 Wrapping an external C library

So far, I've shown you how to wrap some very simple procedures that are part of the C standard library. Most of these procedures have already been wrapped to some extent by the Nim standard library and are exposed via modules such as `times`.

Wrapping an external library is slightly different. In this section you'll learn about these differences as you wrap a small bit of the SDL library.

Simple DirectMedia Layer, or SDL, is a cross-platform multimedia library. It's one of the most widely used libraries for writing computer games and other multimedia applications. SDL manages video, audio, input devices, and much more. Some practical things that you can use it for are drawing 2D graphics on the screen or playing sound effects.

I'll show you how to draw 2D graphics. By the end of this section, you'll produce an application that displays the window shown in figure 8.5.

SDL WRAPPER The wrapper shown here will be very basic. You'll find a full SDL wrapper that's already been created by the Nim community here: https://github.com/nim-lang/sdl2.

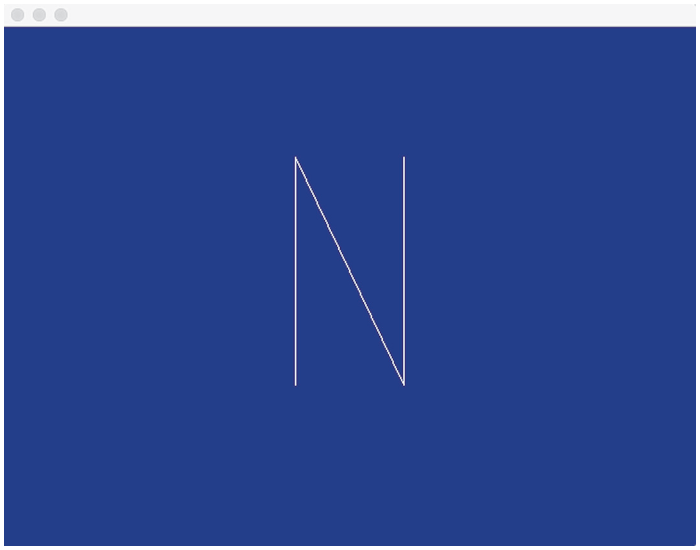


Figure 8.5 The application you'll produce in this section

8.2.1 Downloading the library

Before you begin writing the wrapper for the SDL library, you should download it. For this chapter's example, you'll only need SDL's runtime binaries, which you can download here: www.libsdl.org/download-2.0.php#source.

8.2.2 Creating a wrapper for the SDL library

A library, or package, wrapper consists of one or more modules that contain wrapped procedures and type definitions. The wrapper modules typically mirror the contents of C header files, which contain multiple declarations of procedure prototypes and types. But they may also mirror other things, such as the contents of JavaScript API reference documentation.

For large libraries like SDL, these header files are very large, containing thousands of procedure prototypes and hundreds of types. The good news is that you don't need to wrap it all completely in order to use the library. A couple of procedures and types will do. This means you can wrap libraries on demand instead of spending days wrapping the full library, including procedures that you're never going to use. You can just wrap the procedures that you need.

AUTOMATIC WRAPPING An alternative means of wrapping libraries is to use a tool such as c2nim. This tool takes a C or C++ header file as input and converts it into a wrapper. For more information about c2nim, take a look at its documentation: http://nim-lang.org/docs/c2nim.html.

As in the previous section, you can go online to look up the definition of the procedure prototypes that you're wrapping. Be sure to consult the project's official documentation and ensure that it has been written for the version of the library that you're using. Alternatively, you can look up the desired procedure or type inside the library's header files.

First, though, you need to figure out what needs to be wrapped. The easiest way to figure that out is to look for examples in C, showing how the library in question can be used to develop a program that performs your desired actions. In this section, your objective is to create an application that shows a window of a specified color with the letter N drawn in the middle, as shown in figure 8.5.

The SDL library can do a lot more than this, but in the interest of showing you how to wrap it, we'll focus on this simple example.

With that in mind, let's start. The wrapper itself will be a single module called `sdl`. Before moving on to the next section, create this module by creating a new file called sdl.nim.

8.2.3 Dynamic linking

Earlier in this chapter, I explained the differences between static and dynamic linking. The procedures you wrapped in the previous section are part of the C standard library, and as such, the linking process was automatically chosen for you. The process by which the C standard library is linked depends on your OS and C compiler.

When it comes to linking with external C libraries, dynamic linking is recommended. This process involves some trivial initial setup that we'll look at now.

Whenever you instruct the Nim compiler to dynamically link with a C library, you must supply it with the filename of that library. The filenames depend entirely on the library and the OS that the library has been built for. Table 8.2 shows the filenames of the SDL libraries for Windows, Linux, and Mac OS.

Table 8.2 The filenames of the SDL library

Windows	Linux	Mac OS
SDL2.dll	libSDL2.so	libSDL2.dylib

These files are called *shared library* files because in many cases, especially on UNIX-like OSs, they're shared among multiple applications.

The SDL wrapper needs to know these filenames, so let's define them in the `sdl` module you just created. The following listing shows how to define these for each OS. Add this code to your `sdl` module.

Listing 8.4 Defining the shared library filename conditionally

```
when defined(Windows):
  const libName* = "SDL2.dll"
elif defined(Linux):
  const libName* = "libSDL2.so"
elif defined(MacOsX):
  const libName* = "libSDL2.dylib"
```

This code is fairly simple. Only one constant, `libName`, is defined. Its name remains the same, but its value changes depending on the OS. This allows the wrapper to work on the three major OSs.

That's all the setup that's required. Strictly speaking, it's not absolutely necessary to create these constants, but they will enable you to easily change these filenames at a later time.

Now, recall the previous section, where I showed you the `header` and `importc` pragmas. These were used to import C procedures from a specific header in the C standard library. In order to instruct the compiler to dynamically link a procedure, you need to use a new pragma called `dynlib` to import C procedures from a shared library:

```
proc init*(flags: uint32): cint {.importc: "SDL_Init", dynlib: libName.}
```

The `dynlib` pragma takes one argument: the filename of the shared library where the imported procedure is defined. Every time your application starts, it will load a shared library for each unique filename specified by this pragma. If it can't find the shared library, or the wrapped procedure doesn't exist in the shared library, the application will display an error and terminate.

The `dynlib` pragma also supports a simple versioning scheme. For example, if you'd like to load either `libSDL2-2.0.1.so` or `libSDL2.so`, you can specify `"libSDL2(|-2.0.1).so"` as the argument to `dynlib`. More information about the `dynlib` pragma is available in the Nim manual: http://nim-lang.org/docs/manual.html#foreign-function-interface-dynlib-pragma-for-import.

Now, you're ready to start wrapping.

8.2.4 Wrapping the types

Before you can successfully wrap the required procedures, you first need to define four types. Thankfully, wrapping their internals isn't necessary. The types will simply act as stubs to identify some objects. The following listing shows how to define these types.

Listing 8.5 Wrapping the four necessary types

```
type
  SdlWindow = object
  SdlWindowPtr* = ptr SdlWindow
  SdlRenderer = object
  SdlRendererPtr* = ptr SdlRenderer
```

Defines an object stub. This object likely contains fields, but you don't need to access them in your application, so you can omit their definitions.

Many of the procedures in the SDL library work on pointers to objects, so it's convenient to give this type a name and export it instead of writing "ptr TheType" everywhere.

The type definitions are fairly simple. The `SdlWindow` type will represent a single on-screen SDL window, and the `SdlRenderer` will represent an object used for rendering onto the SDL window.

The pointer types are defined for convenience. They're exported because the SDL procedures that you'll wrap soon return them.

Let's look at these procedures now.

8.2.5 *Wrapping the procedures*

Only a handful of procedures need to be wrapped in order to show a colored window on the screen using SDL. The following listing shows the C prototypes that define those procedures.

Listing 8.6 The SDL C prototypes that will be wrapped in this section

```
int SDL_Init(Uint32 flags)                              <-- Initializes the SDL library

int SDL_CreateWindowAndRenderer(int            width,
                                int            height,
                                Uint32         window_flags,
                                SDL_Window**   window,
                                SDL_Renderer** renderer)
                    Creates an SDL window and rendering context associated with that window

int SDL_PollEvent(SDL_Event* event)                     <-- Checks for input events

int SDL_SetRenderDrawColor(SDL_Renderer* renderer,
                           Uint8         r,
                           Uint8         g,
                           Uint8         b,
                           Uint8         a)
                    Sets the current draw color on the specified renderer

void SDL_RenderPresent(SDL_Renderer* renderer)
                    Updates the screen with any rendering that was performed

int SDL_RenderClear(SDL_Renderer* renderer)
                    Clears the specified renderer with the drawing color

int SDL_RenderDrawLines(SDL_Renderer*     renderer,
                        const SDL_Point* points,
                        int               count)
                    Draws a series of connected lines
```

You've already seen how to wrap the `SDL_Init` procedure:

```
proc init*(flags: uint32): cint {.importc: "SDL_Init", dynlib: libName.}
```

The wrapper for this procedure is fairly straightforward. The `Uint32` and `int` types in the prototype map to the `uint32` and `cint` Nim types, respectively. Notice how the procedure was renamed to `init`; this was done because the `SDL_` prefixes are redundant in Nim.

Now consider the rest of the procedures. Each wrapped procedure will need to specify the same `dynlib` pragma, but you can remove this repetition with another pragma called the `push` pragma. The `push` pragma allows you to apply a specified pragma to the procedures defined below it, until a corresponding `pop` pragma is used.

The following listing shows how the rest of the procedures can be wrapped with the help of the push pragma.

Listing 8.7 Wrapping the procedures in the sdl module

The pointer type in Nim is equivalent to a void *, which is a pointer of any type.

This ensures that each proc gets the dynlib pragma.

The var keyword is used in place of a ptr. In Nim, these end up generating equivalent C code.

```
{.push dynlib: libName.}
proc init*(flags: uint32): cint {.importc: "SDL_Init".}

proc createWindowAndRenderer*(width, height: cint, window_flags: cuint,
    window: var SdlWindowPtr, renderer: var SdlRendererPtr): cint
    {.importc: "SDL_CreateWindowAndRenderer".}

proc pollEvent*(event: pointer): cint {.importc: "SDL_PollEvent".}

proc setDrawColor*(renderer: SdlRendererPtr, r, g, b, a: uint8): cint
    {.importc: "SDL_SetRenderDrawColor", discardable.}

proc present*(renderer: SdlRendererPtr) {.importc: "SDL_RenderPresent".}

proc clear*(renderer: SdlRendererPtr) {.importc: "SDL_RenderClear".}

proc drawLines*(renderer: SdlRendererPtr, points: ptr tuple[x, y: cint],
    count: cint): cint {.importc: "SDL_RenderDrawLines", discardable.}
{.pop.}
```

The discardable pragma is used here to implicitly discard the return value.

This stops the propagation of the dynlib pragma.

The points parameter is a pointer to the beginning of an array of tuples.

Most of the code here is fairly standard. The createWindowAndRenderer procedure's arguments include one pointer to a pointer to an SdlWindow and another pointer to a pointer to an SdlRenderer, written as SdlWindow** and SdlRenderer**, respectively. Pointers to SdlWindow and SdlRenderer were already defined in the previous subsection under the names SdlWindowPtr and SdlRendererPtr, respectively, so you can define the types of those arguments as ptr SdlWindowPtr and ptr SdlRendererPtr. This will work well, but using var in place of ptr is also appropriate in this case.

You may recall var T being used in chapter 6, where it stored a result in a variable that was passed as a parameter to a procedure. The exact same thing is being done by the createWindowAndRenderer procedure. Nim implements these var parameters using pointers, so defining that argument's type using var is perfectly valid. The advantage of doing so is that you no longer need to use addr, and Nim also prevents you from passing nil for that argument.

For the pollEvent procedure, the argument type was defined as pointer. This type is equivalent to a void* type in C, essentially a pointer to any type. This was done because it avoids the need to wrap the SdlEvent type. You may run into C libraries that declare procedures accepting a void* type, in which case you can use the pointer type. In practice, however, it's better to use a `ptr T` type for improved type safety. But you can only do so if you know that the procedure you're wrapping will only ever accept a specific pointer type.

Lastly, the `drawLines` procedure is the most complicated, as it accepts an array of points to draw as lines. In C, an array of elements is represented by a pointer to the first element in the array and the number of variables in that array. In the case of the `drawLines` procedure, each element in the `points` array is an `SDL_Point` type, and it's defined as a simple C struct containing two integers that represent the `x` and `y` coordinates of the point. In Nim, this simple struct can be represented using a tuple.

Add the contents of listing 8.7 to your `sdl` module. It's time to use it to write the application.

8.2.6 Using the SDL wrapper

You can now use the wrapper you've just written. First, create an sdl_test.nim file beside your wrapper, and then import the wrapper by writing `import sdl` at the top of the file.

Before the library can be used, you'll have to initialize it using the `init` procedure. The `init` procedure expects to receive a `flags` argument that specifies which SDL subsystems should be initialized. For the purposes of this application, you only need to initialize the video subsystem. To do this, you'll need to define a constant for the `SDL_INIT_VIDEO` flag, like this:

```
const INIT_VIDEO* = 0x00000020
```

The value of this constant needs to be defined in the Nim source file because it's not available in the shared library. C header files typically define such constants using a `#define` that isn't compiled into any shared libraries.

Add this constant into your `sdl` module. Then, you'll finally be ready to use the `sdl` wrapper to implement a simple application. The following listing shows the code needed to do so.

Listing 8.8 An SDL application implemented using the `sdl` wrapper

```
import os
import sdl

if sdl.init(INIT_VIDEO) == -1:                 # Initializes the SDL video subsystem
  quit("Couldn't initialise SDL")               # Quits with an error if the initialization fails

var window: SdlWindowPtr
var renderer: SdlRendererPtr
if createWindowAndRenderer(640, 480, 0, window, renderer) == -1:   # Creates a window and renderer to draw things on
  quit("Couldn't create a window or renderer")  # Quits with an error if the creation of the window or renderer fails
  discard pollEvent(nil)
  renderer.setDrawColor 29, 64, 153, 255
  renderer.clear

  renderer.present
  sleep(5000)
```

This is where you'd handle any pending input events. For this application, it's only called so that the window initializes properly.

Sets the drawing color to the specified red, green, blue, and alpha values

Clears the screen with the specified drawing color

Shows the pixels drawn on the renderer

Waits for 5 seconds before terminating the application

Compile and run the sdl_test.nim file. You should see a window with a blue background, as shown in figure 8.6 (to see color versions of the figures, please refer to the electronic version of this book).

A blank SDL window is a great achievement, but it isn't a very exciting one. Let's use the drawLines procedure to draw the letter *N* in the middle of the screen. The following code shows how this can be done:

Figure 8.6 The result of running listing 8.8

```
renderer.setDrawColor 255, 255, 255, 255
var points = [
  (260'i32, 320'i32),
  (260'i32, 110'i32),
  (360'i32, 320'i32),
  (360'i32, 110'i32)
]
renderer.drawLines(addr points[0], points.len.cint)
```

Changes the draw color to white

Defines an array of points that define the coordinates to draw an N. Each coordinate must be an int32 because that's what a cint is.

Draws the lines defined by the points array

Add this code just below the `renderer.clear` statement in the sdl_test.nim file. Then, compile and run the file. You should see a window with a blue background and the letter *N*, as shown in figure 8.7.

In the preceding code, the `drawLines` call is the important one. The address of the first element in the `points` array is passed to this procedure together with the length of the `points` array. The `drawLines` procedure then has all the information it needs to read all the points in the array. It's important to note that this call isn't memory safe; if the points count is too high, the `drawLines` procedure will attempt to read memory

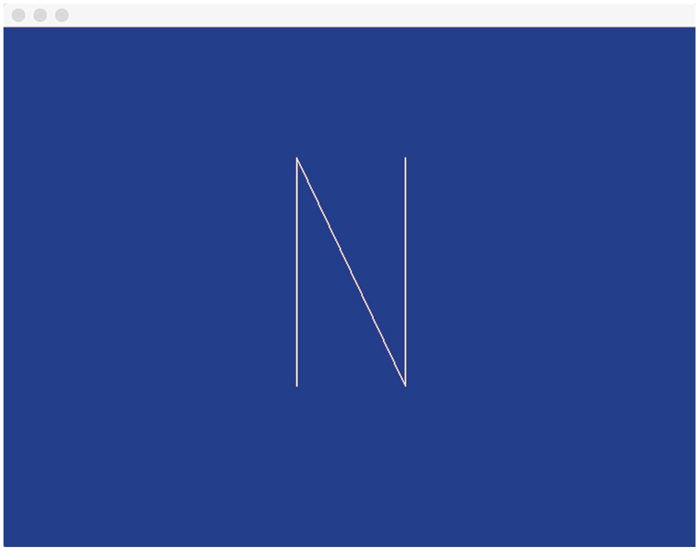


Figure 8.7 The final sdl_test application with the letter N drawn

that's adjacent to the array. This is known as a *buffer overread* and can result in serious issues because there's no way of knowing what the adjacent memory contains.[2]

That's how you wrap an external library using Nim. Of course, there's plenty of room for improvement. Ideally, a module that provides a higher-level API should always be written on top of a wrapper; that way, a much more intuitive interface can be used for writing applications. Currently, the biggest improvement that could be made to the `sdl` module is to add exceptions. Both `init` and `createWindowAndRenderer` should raise an exception when an error occurs, instead of requiring the user to check the return value manually.

The last two sections have given you an overview of the C FFI. Nim also supports interfacing with other C-like languages, including C++ and Objective-C. Those two backends are beyond the scope of this book, but the concepts you've learned so far should give you a good starting point. For further information about these backends, take a look at the Nim manual: http://nim-lang.org/docs/manual.html#implementation-specific-pragmas-importcpp-pragma.

Next, we'll look at how to write JavaScript wrappers.

8.3 The JavaScript backend

JavaScript is increasingly becoming known as the "assembly language of the web" because of the many new languages that target it. Languages that can be translated to JavaScript are desirable for various reasons. For example, they make it possible to

[2] See the Wikipedia article for an explanation of buffer overreads: https://en.wikipedia.org/wiki/Buffer_over-read.

share code between client scripts that run in a web browser and applications that run on a server, reducing the need for code duplication.

As an example, consider a chat application. The server manages connections and messages from multiple clients, and a client script allows users to connect to the server and send messages to it from their web browser. These messages must be understood by all the clients and the server, so it's beneficial for the code that parses those messages to be shared between the server and the client. If both the client and the server are written in Nim, sharing this code is trivial. Figure 8.8 shows how such a chat application could take advantage of Nim's JavaScript backend.

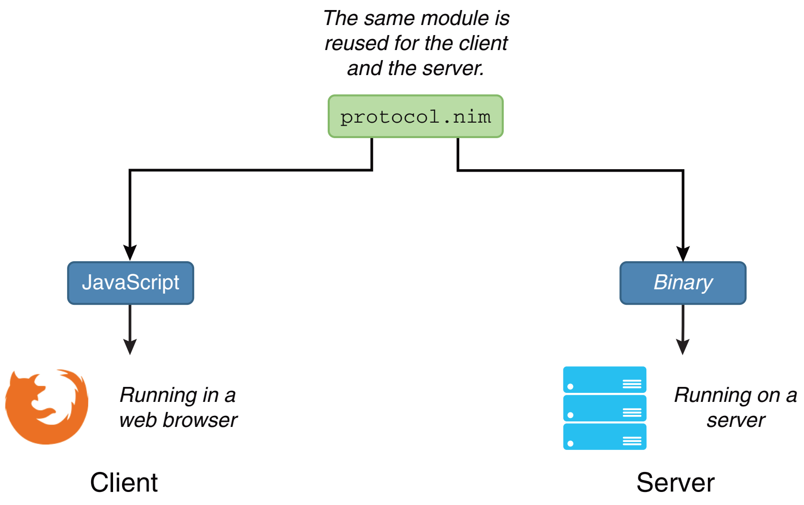


Figure 8.8 How the same code is shared between two platforms

Of course, when writing JavaScript applications, you'll eventually need to interface with the APIs exposed by the web browser as well as libraries that abstract those APIs. The process of wrapping JavaScript procedures and types is similar to what was described in the previous sections for the C backend, but there are some differences that are worth an explanation.

This section will show you how to wrap the JavaScript procedures required to achieve the same result as in the previous section with the SDL library: filling the drawable surface with a blue color and drawing a list of lines to form the letter N.

8.3.1 *Wrapping the canvas element*

The canvas element is part of HTML5, and it allows rendering of 2D shapes and bitmap images on an HTML web page. All major web browsers support it and expose it via a JavaScript API.

Let's look at an example of its usage. Assuming that an HTML page contains a `<canvas>` element with an ID of `canvas`, and its size is 600 x 600, the code in the following listing will fill the canvas with the color blue and draw the letter N in the middle of it.

Listing 8.9 Using the Canvas API in JavaScript

```
var canvas = document.getElementById("canvas");
canvas.width = 600;
canvas.height = 600;
var ctx = canvas.getContext("2d");

ctx.fillStyle = "#1d4099";
ctx.fillRect(0, 0, 600, 600);
ctx.strokeStyle = "#ffffff";
ctx.moveTo(250, 320);
ctx.lineTo(250, 110);
ctx.lineTo(350, 320);
ctx.lineTo(350, 110);
ctx.stroke();
```

The code is fairly self-explanatory. It starts by retrieving the canvas element from the DOM by ID. The canvas size is set, and a 2D drawing context is created. Lastly, the screen is filled with a blue color, the letter *N* is traced using the `moveTo` and `lineTo` procedures, and the letter is drawn using the `stroke` procedure. Wrapping the procedures used in this example shouldn't take too much effort, so let's begin.

Create a new file called canvas.nim. This file will contain the procedure wrappers needed to use the Canvas API. The `getElementById` procedure is already wrapped by Nim; it's a part of the DOM, so it's available via the `dom` module.

Unlike in C, in JavaScript there's no such thing as a header file. The easiest way to find out how a JavaScript procedure is defined is to look at the documentation. The following list contains the documentation for the types and procedures that will be wrapped in this section:

- `CanvasRenderingContext2D` type—https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D
- `canvas.getContext(contextType, contextAttributes);` procedure—http://mng.bz/6kIp
- `void ctx.fillRect(x, y, width, height);` procedure—http://mng.bz/xN3Y
- `void ctx.moveTo(x, y);` procedure—http://mng.bz/A9Bk
- `void ctx.lineTo(x, y);` procedure—http://mng.bz/t355
- `void ctx.stroke();` procedure—http://mng.bz/nv6C

Because JavaScript is a dynamically typed programming language, procedure definitions don't contain information about each argument's type. You must look at the documentation, which more often than not tells you enough to figure out the underlying type. The following listing shows how the `CanvasRenderingContext2D` type and the five procedures should be wrapped. Save the listing as canvas.nim.

Listing 8.10 Wrapping the Canvas API

```
import dom

type
  CanvasRenderingContext* = ref object
    fillStyle* {.importc.}: cstring
    strokeStyle* {.importc.}: cstring

{.push importcpp.}

proc getContext*(canvasElement: Element,
    contextType: cstring): CanvasRenderingContext

proc fillRect*(context: CanvasRenderingContext, x, y, width, height: int)

proc moveTo*(context: CanvasRenderingContext, x, y: int)

proc lineTo*(context: CanvasRenderingContext, x, y: int)

proc stroke*(context: CanvasRenderingContext)
```

The dom module exports the Element type used in the getContext proc.

All JavaScript objects have ref semantics; hence, the ref object definition.

Each field must be explicitly imported using importc.

Each procedure is given the importcpp pragma.

The contextAttributes argument is intentionally omitted here. It's an optional argument with a default value.

This code is fairly short and to the point. You should be familiar with everything except the `importcpp` pragma. The name of this pragma is borrowed from the C++ backend. It instructs the compiler to generate JavaScript code that calls the specified procedure as if it were a member of the first argument's object. Figure 8.9 demonstrates the difference between `importc` and `importcpp` for the JavaScript backend.

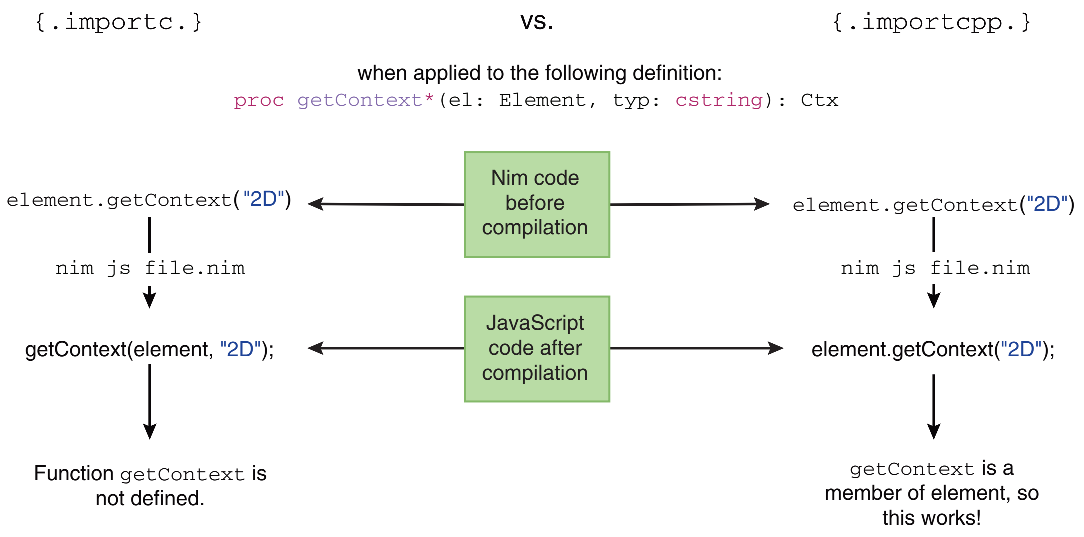

{.importc.}
vs.
{.importcpp.}
when applied to the following definition:
proc getContext*(el: Element, typ: cstring): Ctx
Nim code before compilation
JavaScript code after compilation
element.getContext("2D")
nim js file.nim
getContext(element, "2D");
Function getContext is not defined.
element.getContext("2D")
nim js file.nim
element.getContext("2D");
getContext is a member of element, so this works!

Figure 8.9 The differences in JavaScript code produced with the `importc` and `importcpp` pragmas

There aren't many other surprises, but one interesting aspect to note is that when you're wrapping a data type in JavaScript, the wrapped type should be declared as a `ref object`. JavaScript objects have reference semantics and so should be wrapped as such.

That's all there is to it! Time to put this wrapper to use.

8.3.2 Using the Canvas wrapper

Now that the wrapper is complete, you can write a little script that will make use of it, together with a small HTML page to execute it.

Save the following listing as index.html beside your canvas.nim file.

Listing 8.11 The index.html file

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>Nim in Action - Chapter 8</title>
    <script type="text/javascript" src="canvas_test.js"></script>
    <style type="text/css">
      canvas { border: 1px solid black; }
    </style>
  </head>
  <body onload="onLoad();" style="margin: 0; overflow: hidden;">
    <canvas id="canvas"></canvas>
  </body>
</html>
```

The HTML is pretty bare bones. It's got some small style adjustments to make the canvas full screen, and it defines an `onLoad` procedure to be called when the `<body>` tag's `onLoad` event fires.

Save the next listing as canvas_test.nim beside your canvas.nim file.

Listing 8.12 The canvas_test.nim file

```
import canvas, dom

proc onLoad() {.exportc.} =
  var canvas = document.getElementById("canvas").EmbedElement
  canvas.width = window.innerWidth
  canvas.height = window.innerHeight
  var ctx = canvas.getContext("2d")

  ctx.fillStyle = "#1d4099"
  ctx.fillRect(0, 0, window.innerWidth, window.innerHeight)
```

Note how similar the code is to JavaScript. This code listing defines an `onLoad` procedure that's then exported, which allows the browser to use it as an event callback. The `exportc` procedure is used to do this. It simply ensures that the generated JavaScript code contains an `onLoad` procedure. This pragma also works for the other backends.

You may wonder what the purpose of the .EmbedElement type conversion is. The getElementById procedure returns an object of type Element, but this object doesn't have width or height properties, so it must be converted to a more concrete type. In this case, it's converted to the EmbedElement type, which allows the two width and height assignments.

Compile this canvas_test module by running nim js -o:canvas_test.js canvas_test.nim. You can then test it by opening the index.html file in your favorite browser. You should see something resembling figure 8.10.

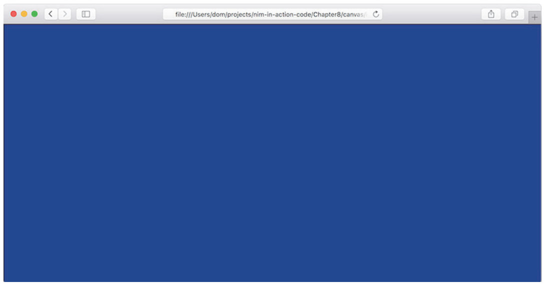

file:///Users/dom/projects/nim-in-action-code/Chapter8/canvas...

Figure 8.10 The canvas_test.nim script showing a blue screen in the web browser

For now, this is just a blue screen. Let's extend it to draw the letter *N*. Add the following code at the bottom of the onLoad procedure:

```
ctx.strokeStyle = "#ffffff"
let letterWidth = 100
let letterLeftPos = (window.innerWidth div 2) - (letterWidth div 2)
ctx.moveTo(letterLeftPos, 320)
ctx.lineTo(letterLeftPos, 110)
ctx.lineTo(letterLeftPos + letterWidth, 320)
ctx.lineTo(letterLeftPos + letterWidth, 110)
ctx.stroke()
```

Sets the stroke color to white

Creates a local letterWidth variable to store the desired letter width

Calculates the top-left position where the letter should be placed

Begins tracing the lines of the letter

Draws the letter

In this case, the code calculates where the letter should be placed so that it's in the middle of the screen. This is necessary because the canvas size depends on the size of the web browser window. In the SDL example, the SDL window was always the same size, so this calculation wasn't needed.

Recompile the canvas_test.nim file by running the same command again, and then refresh your browser. You should see something resembling figure 8.11.

That's all there is to it. You should now have a good basic understanding of how to wrap JavaScript and how to make use of Nim's JavaScript backend.

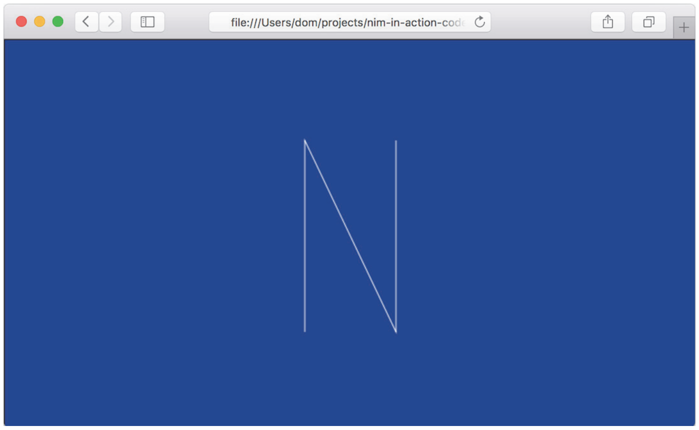


Figure 8.11 The canvas_test.nim script showing a blue screen with the letter N in the web browser

8.4 Summary

- The Nim foreign function interface supports interfacing with C, C++, Objective-C, and JavaScript.
- C libraries can be either statically or dynamically linked with Nim applications.
- C header files declare procedure prototypes and types that provide all the information necessary to wrap them.
- The `importc` pragma is used to wrap a foreign procedure, including C and JavaScript procedures.
- The `discardable` pragma can be used to override the need to explicitly discard values.
- The `cstring` type should be used to wrap procedures that accept a string argument.
- Using an external C library is best done via dynamic linking.
- The `dynlib` pragma is used to import a procedure from a shared library.
- The `importcpp` pragma is used to wrap C++ procedures and also member procedures in JavaScript.