Working with C

Jan 14, 2023
2023 2023 Jan c hashlink haxe

Haxe has a special keyword called `extern`, which is used to indicate to the Haxe compiler that this class will be implemented in the target somehow. I've never really understood it because, as far as I can tell, the semantics of using it depend entirely on the target. Case in point: it doesn't seem to work at all for Hashlink. At least in my testing, any use of `extern` would throw up the following vague error from the Haxe compiler:

> Invalid_argument("index out of bounds")

Instead, there is a dedicated macro for Hashlink called `@:hlNative`, which you use to tell the Haxe compiler that a class or method exists in the VM's **natives table**. This is strictly a Hashlink concept, which is nicely detailed in this article by the guy who built everything I'm talking about here.

Using it is pretty straightforward:

```
@:hlNative("nose")
class Nose {
    public static function sneeze():String {
        return null;
    }
}
```

The name in parenthesis will be used when generating the corresponding call in C. It also seems that doing anything other than returning null from the function will cause the compiler to treat it as a regular Haxe function, rather than an external C one. This is convenient in case we want parts of our class to be implemented in Haxe, which greatly reduces the amount of C code you need to maintain.

Here's an example of using that class:

```
class Main {
    public static function main(){
        trace(Nose.sneeze());
    }
}
```

Compiling that with HL/C will generate the following C code:

```
// Generated by HLC 4.2.5 (HL v4)
#define HLC_BOOT
```

```c
#include <hlc.h>
#include <_std/Main.h>
#include <hl/types/ArrayDyn.h>
#include <haxe/Log.h>
extern haxe__$Log g$_haxe_Log;
#include <hl/natives.h>
extern hl_type t$vrt_329ffa8;
extern String s$Main_hx;
extern hl_type t$String;
extern hl_type t$_i32;
extern String s$Main;
extern String s$main;
extern hl_type t$vrt_eaa6a3b;

void Main_main() {
    String r3, r5;
    vvirtual *r4, *r7;
    haxe__$Log r2;
    vclosure *r1;
    int r6;
    r2 = (haxe__$Log)g$_haxe_Log;
    r1 = r2->trace;
    if( r1 == NULL ) hl_null_access();
    r3 = nose_sneeze(); // <---------- This is ours!
    r4 = hl_alloc_virtual(&t$vrt_329ffa8);
    r5 = (String)s$Main_hx;
    if( hl_vfields(r4)[1] ) *(String*)(hl_vfields(r4)[1]) = (String)r5; else
    r6 = 3;
    if( hl_vfields(r4)[2] ) *(int*)(hl_vfields(r4)[2]) = (int)r6; else hl_dy
    r5 = (String)s$Main;
    if( hl_vfields(r4)[0] ) *(String*)(hl_vfields(r4)[0]) = (String)r5; else
    r5 = (String)s$main;
    if( hl_vfields(r4)[3] ) *(String*)(hl_vfields(r4)[3]) = (String)r5; else
    r7 = hl_to_virtual(&t$vrt_eaa6a3b,(vdynamic*)r4);
    r1->hasValue ? ((void (*)(vdynamic*,vdynamic*,vvirtual*))r1->fun)((vdyna
    return;
}
```

Note the call to `nose_sneeze();`. Remember that this is C11, not C++, so we don't have classes or anything like that. Thus the final symbol name is a combination of the Haxe class name and method name, and it is up to you to ensure you provide an implementation following the correct naming convention.

In addition to the references at call sites (like above), the only other presence of `nose_sneeze` in the generated C code is in 2 places: the `<out>/hl/natives.h` header, which simply declares all functions in the native table:

```c
//...
HL_API String nose_sneeze(void);
//...
```

...and in the `<out>/hl/functions.c` file as an entry in the global array of function pointers:

```c
void *hl_functions_ptrs[] = {
    //...
    nose_sneeze,
    //...
};
```

I don't know if HL/C ever uses those function pointers in generated code, but it might be used internally by some of HL's API functions (e.g. reflection). It's also probably used by the VM when executing byte code.

One interesting observation here is that we could potentially play around with `hl_functions_ptrs` before loading our bytecode in order to intercept/manipulate functions. Idk how well this would work in practice, but even if it does, doing that will lead to deviation from HL/C, since HL/C doesn't seem to be using the table in the same way. So if we're interested in supporting both HL and HL/C in whatever we're building, it's probably not a good idea to mess with that.

Here's an example implementation of our `nose_sneeze` function in C++:

```cpp
//nose.cpp
#include <hl.h>
#include <_std/String.h>
extern hl_type t$String;

String make_string(const char *str){
    hl_buffer *b = hl_alloc_buffer();
    String ret = (String)hl_alloc_obj(&t$String);
    int len;
    hl_buffer_cstr(b, str);
    ret->bytes = (vbyte*) hl_buffer_content(b, &len);
    ret->length = len;
    return ret;
}

extern "C" String nose_sneeze(void) {
    return make_string("ah-choo!");
}
```

The `make_string` function uses the HL APIs to allocate a Haxe `String` object and set its value from a standard C string. Remember that HL has a garbage collector, so as long as we allocate using the HL allocation APIs (like `hl_alloc_obj` above), we should be safe.

One bit of ugliness is the `extern hl_type t$String;`. This is required because `hl_alloc_obj` expects an `hl_type` pointer, yet this is only defined in the generated file `<out>/hl/types.c`. Since it's not declared in any header, we need to reference it using an extern declaration. Is there a nicer way to do this out of the box? I don't know, but it seems like those symbol names are stable, so they won't unexpectedly change between builds.

Looking at `types.c`, we can see that classes all follow the pattern `t$<classname>`, however there also seem to be some duplicates in the form of `t$<classname>`. For example, the String class has the following definitions:

```c
hl_type t$String = { HOBJ };
hl_type t$String = { HOBJ };
```

Reading through some of the generated code, it looks like the `t$<classname>` versions are meant to represent the class type. So if we update our example Main.hx from above to this:

```
class Main {
    public static function main(){
        trace(Type.getClass((Nose.sneeze())));
        trace(Nose.sneeze());
    }
}
```

Then here's the output when we use `t$String` in our `make_string` function:

> Main.hx:3: $String Main.hx:4: ah-choo!

and here is is when we use `t$String`:

> Main.hx:3: null Main.hx:4: $String

Obviously this wasn't designed to work this way, but I think it's clear what those two symbols represent.

(Index) How to embed HL/C programs