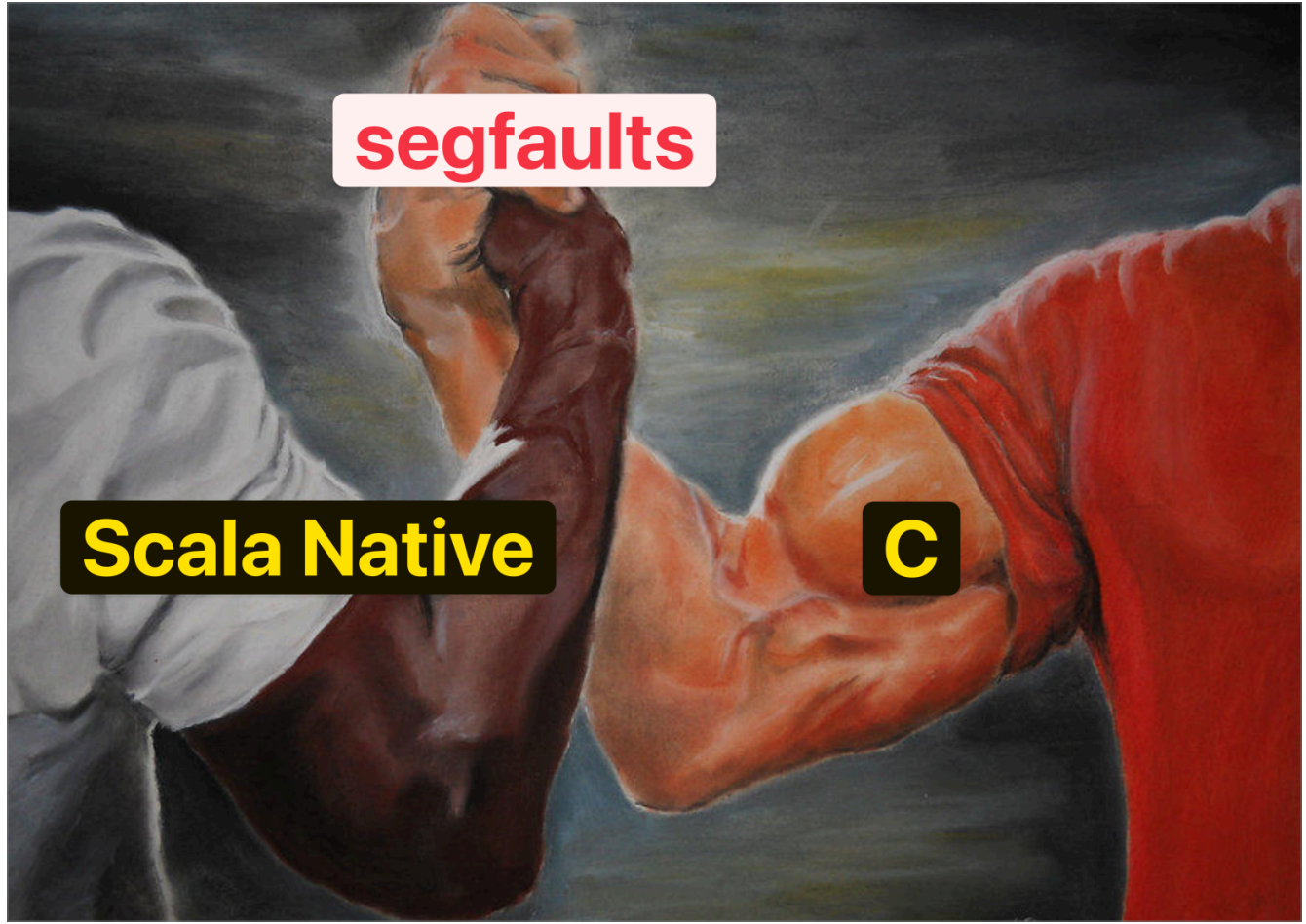
Scala Native and C – tight as brothers

Anton Sviridov, Scalar 2026

PDF


segfaults
Scala Native
C

Acknowledgements

- **Scala Native** – optimising ahead-of-time compiler for Scala, producing native binaries and libraries, with C interop capabilities
- Most of the things here you see here are solved by sn-bindgen
- I am not a C expert or even user – interop and parsing is where my interests lie
- The river of pain runs deep, the examples are not exhaustive

Advent of Scala Native

- In ideal world, we'd have clean room implementations for *everything*
  - And in some stacks we do!
- In the meantime, we can take hardened C implementations and build fundamental JVM-like shims:
  - `openssl` -> `java.security.*`
  - `curl` -> `java.net.http.HttpClient`
  - `libpq/libmysql/sqlite3` -> `java.sql.*`


Idiomatic Scala libraries
Established JVM interfaces
Fundamental C libraries

Advent of Scala Native

We know it's possible

- `openssl` -> `java.security.*`

  https://github.com/lolgab/scala-native-crypto

- `curl` -> `java.net.http.HttpClient`

  https://github.com/scala-native/scala-native/issues/4104#issuecomment-3702281878

- `libpq/libmysql/sqlite3` -> `java.sql.*`

  https://github.com/lolgab/scala-native-jdbc

C libraries in Scala Native are inescapable – but to work with them effectively, we must learn about C ABI


ABI
Your program
Library function

ABI

Set of rules governing how separately compiled units can interact with each other at binary level:

- How to call functions and where to put arguments, returns values
- How to layout data structures in memory, down to a byte
- Unfortunately, ABI for C libraries is expressed solely as C code
  - In the same language that has had 50+ years of syntax changes and deprecations that never went away

ABI is not source code based – by the time interaction happens, most structural and lexical information is erased

Let’s express some ABI contracts in C and see what happens at binary time

Sidebar: it’s C all the way down

C being the lingua franca of inter-system communication is a sad state of affairs

https://faultlore.com/blah/c-isnt-a-language/

Demonstration

Two compilation modules, `b` referring to `a`

a.c, `clang a.c -c` – produces `a.o`

```c
#include <stdio.h>

float sum(float a, float b)
{
    printf("a: %f, b: %f\n", a, b);
    return a + b;
}
```

b.c, `clang b.c a.o` – produces executable `a.out`

```c
#include <stdio.h>

extern float sum(float a, float b);

int main()
{
    printf("%f\n", sum(1.0, 2.0f));
    return 0;
}
```

Demonstration

- So far so good, running `a.out` produces `3.000000` as expected.
- But what if we change the extern signature of `sum` to take one `double` instead of `float`?

b.c, `clang b_bad.c a.o` – produces executable `a.out`

```c
#include <stdio.h>

extern float sum(double a /* 😱 */, float b);

int main()
{
    printf("%f\n", sum(1.0, 2.0f));
    return 0;
}
```

This produces `2.000000` instead of `3.000000`!

Can you guess why?

Demonstration

Answer (on ARM64): when `sum` is being called, the caller puts arguments into registers:

```
fmov   d0, #1.00000000
fmov   s1, #2.00000000
```

- `d0` is a 64-bit register "view" of `x0` register, which means the first 32 significant bits of `x0` are 0!
- `s1` is a 32-bit register "view" of `x1` register, setting the first 32 significant bits correctly.

Demonstration

But when `sum` runs, it reads from only the significant portion of the register:

```
0x1000004f0 <+68>: fadd   s0, s0, s1
```

Which in the case of `s0` will be all zeroes!

```
(lldb) reg read s0
      s0 = 0
(lldb) reg read s1
      s1 = 2
(lldb) reg read d0
      d0 = 1
```

The lesson here is that ABI is real, and it WILL find you, and it WILL hurt you

C libraries and interfaces

- A lot (most) of fundamental services and libraries provide **only C programmatic interface**
  - Postgres, MySQL, OpenSSL, Curl, Git, sqlite, SDL, Linux kernel, etc.
- The interface itself is expressed in C language itself
  - Structs, unions, functions, pointers, macro definitions (!), primitive types, etc.
- To know how to use those interfaces safely, it becomes important to learn the quirks of C language

Scala Native interop: primitive types and functions

- `scala-cli run b.scala --native --native-linking $(pwd)/a.o`

```
//> using platform native
import scalanative.unsafe.*

@extern def sum(a: Float, b: Float): Float = extern

@main def hello =
  println(sum(1.0f, 2.0f))
```

Produces 3.00000 as expected.

Things are simple when only primitive types are involved.

We can’t surive on numbers alone, we must evolve.

Scala Native interop: structs

```c
#include <stdio.h>

typedef struct
{
    float a;
    float b;
} Sumbo;

float sum(Sumbo *s)
{
    printf("a: %f, b: %f\n", s->a, s->b);
    return s->a + s->b;
}
```

Scala Native interop: structs

Our interface will look like this:

```
//> using platform native

import scalanative.unsafe.*

type Sumbo = CStruct2[Float, Float]

@extern def sum(a: Ptr[Sumbo]): Float = extern

@main def hello =
  val struct = stackalloc[Sumbo]()
  struct._1 = 1.0f
  struct._2 = 2.0f
  println(sum(struct))
```

Opaque types for structs

```scala
//> using platform native
import scalanative.unsafe.*

object types:
  opaque type Sumbo = CStruct2[Float, Float]
  object Sumbo:
    given Tag[Sumbo] = summon[Tag[CStruct2[Float, Float]]]
    extension (s: Sumbo)
      inline def a = s._1
      inline def a_=(other: Float) = (s._1 = other)
      inline def b = s._2
      inline def b_=(other: Float) = (s._2 = other)

    extension (s: Ptr[Sumbo])
      inline def a = s.at1
      inline def a_=(value: Float) = (s._1 = value)
      inline def b = s.at2
      inline def b_=(value: Float) = (s._2 = value)

import types.*
```

Opaque types for structs

```
1   //> using platform native
2   import scalanative.unsafe.*
3
4   object types:
5     opaque type Sumbo = CStruct2[Float, Float]
6     object Sumbo:
7       given Tag[Sumbo] = summon[Tag[CStruct2[Float, Float]]]
8       extension (s: Sumbo)
9         inline def a = s._1
10        inline def a_=(other: Float) = (s._1 = other)
11        inline def b = s._2
12        inline def b_=(other: Float) = (s._2 = other)
13
14      extension (s: Ptr[Sumbo])
15        inline def a = s.at1
16        inline def a_=(value: Float) = (s._1 = value)
17        inline def b = s.at2
18        inline def b_=(value: Float) = (s._2 = value)
19
20  import types.*
```

Opaque types for structs

```scala
//> using platform native
import scalanative.unsafe.*

object types:
  opaque type Sumbo = CStruct2[Float, Float]
  object Sumbo:
    given Tag[Sumbo] = summon[Tag[CStruct2[Float, Float]]]
    extension (s: Sumbo)
      inline def a = s._1
      inline def a_=(other: Float) = (s._1 = other)
      inline def b = s._2
      inline def b_=(other: Float) = (s._2 = other)

    extension (s: Ptr[Sumbo])
      inline def a = s.at1
      inline def a_=(value: Float) = (s._1 = value)
      inline def b = s.at2
      inline def b_=(value: Float) = (s._2 = value)

import types.*
```

Opaque types for structs

```scala
//> using platform native
import scalanative.unsafe.*

object types:
  opaque type Sumbo = CStruct2[Float, Float]
  object Sumbo:
    given Tag[Sumbo] = summon[Tag[CStruct2[Float, Float]]]
    extension (s: Sumbo)
      inline def a = s._1
      inline def a_=(other: Float) = (s._1 = other)
      inline def b = s._2
      inline def b_=(other: Float) = (s._2 = other)

    extension (s: Ptr[Sumbo])
      inline def a = s.at1
      inline def a_=(value: Float) = (s._1 = value)
      inline def b = s.at2
      inline def b_=(value: Float) = (s._2 = value)

import types.*
```

Opaque types for structs

```scala
//> using platform native
import scalanative.unsafe.*

object types:
  opaque type Sumbo = CStruct2[Float, Float]
  object Sumbo:
    given Tag[Sumbo] = summon[Tag[CStruct2[Float, Float]]]
    extension (s: Sumbo)
      inline def a = s._1
      inline def a_=(other: Float) = (s._1 = other)
      inline def b = s._2
      inline def b_=(other: Float) = (s._2 = other)

    extension (s: Ptr[Sumbo])
      inline def a = s.at1
      inline def a_=(value: Float) = (s._1 = value)
      inline def b = s.at2
      inline def b_=(value: Float) = (s._2 = value)

import types.*
```

Functions

When it comes to simple functions, translation is obvious:

```
1   void test_my_c(float a, int, long c, Sumbo* p);
```

translates to

```
1   import scala.scalanative.unsafe.*
2
3   @extern def test_my_c(a: Float, b: Int, c: Long, p: Ptr[Sumbo]): Unit = extern
```

But some C libraries require passing structs by value.

How can we do that?

Functions and glue code

Thankfully, Scala Native toolchain allows embedding C/C++/Assembly code directly alongside your Scala code.

C function taking pointer to struct

```c
typedef struct {
    int a;
} Sumbo;

void bad_function(Sumbo s);

void __sn_wrapper(Sumbo* ptr) {
    bad_function(*ptr);
}
```

Different wrappers for usability

```scala
@extern
private def __sn_wrapper(ptr: Ptr[Sumbo]): Unit = extern

def bad_function(s: Sumbo)(using Zone): Unit =
  val ptr = alloc[Sumbo](1)
  !ptr = s
  __sn_wrapper(ptr)

def bad_function(ptr: Ptr[Sumbo]): Unit =
  __sn_wrapper(ptr)
```

Recursive definitions

This is completely valid in C:

```c
typedef struct {
  int value;
  struct Tree* left;
  struct Tree* right;
} Tree;
```

Bot not in Scala:

```scala
// cyclic reference error
opaque type Tree = CStruct3[Int, Ptr[Tree], Ptr[Tree]]
```

Recursive definitions

Solution: erase `Ptr[Tree]` to `Ptr[Byte]` and recover it in extension methods.

```
opaque type Tree = CStruct3[CInt, Ptr[Byte], Ptr[Byte]]

object Tree:
   given _tag: Tag[Tree] = Tag.materializeCStruct3Tag[CInt, Ptr[Byte], Ptr[Byte]]

   extension (struct: Tree)
      def value : CInt = struct._1
      def value_=(value: CInt): Unit =
        !struct.at1 = value

      def left : Ptr[Tree] = struct._2.asInstanceOf[Ptr[Tree]]
      def left_=(value: Ptr[Tree]): Unit =
        !struct.at2 = value.asInstanceOf[Ptr[Byte]]

      def right : Ptr[Tree] = struct._3.asInstanceOf[Ptr[Tree]]
      def right_=(value: Ptr[Tree]): Unit =
        !struct.at3 = value.asInstanceOf[Ptr[Byte]]
```

Unions

Scala Native has no representation for unions:

```
union {
  char hello[7];
  double x;
  int bla;
}
```

Used ubiqutously in various libraries

```
struct Item {
  int dataKind;

  union {
    int intValue;
    double doubleValue;
    const char* string;
  } data;
};
```

Unions in Scala Native

```
opaque type Un = CArray[Byte, Nat._8]
object Un:
  // ...
  extension (struct: Un)
     def hello : CArray[CChar, Nat._7] = !struct.at(0).asInstanceOf[Ptr[CArray[CChar, Nat._
     def hello_=(value: CArray[CChar, Nat._7]): Unit =
      !struct.at(0).asInstanceOf[Ptr[CArray[CChar, Nat._7]]] = value

     def x : Double = !struct.at(0).asInstanceOf[Ptr[Double]]
     def x_=(value: Double): Unit =
      !struct.at(0).asInstanceOf[Ptr[Double]] = value

     def bla : CInt = !struct.at(0).asInstanceOf[Ptr[CInt]]
     def bla_=(value: CInt): Unit =
      !struct.at(0).asInstanceOf[Ptr[CInt]] = value
```

Unions in memory

```c
typedef union {
  char hello[7]; // 7 bytes
  double x; // 8 bytes
  int bla; // 4 bytes
} Un; // full size — 8 bytes
```

```c
Un u;
u = (Un){.hello = "hello"};
print_bytes(&u, sizeof(Un));

u.x = 25.0;
print_bytes(&u, sizeof(Un));

u.bla = 11;
print_bytes(&u, sizeof(Un));

u.x = 25.0;
print_bytes(&u, sizeof(Un));
```

```
h  e  l  l  o
68 65 6c 6c 6f 00 00 00
25.0
00 00 00 00 00 00 39 40
11
0b 00 00 00 00 00 39 40
25.0
00 00 00 00 00 00 39 40
```

Largest built-in `CStruct` in Scala Native only goes up to 22.

22 fields should be enough for everyone.
Right?
Right?..

`AVCodecContext` in **ffmpeg** has...

151 fields

Solution?

Erase to `CArray`, calculate field offsets manually, recover type information later.

Large structs (>22 members)

Manually calculating offsets

```
opaque type AVCodecContext = CArray[CChar, /* ... */]
object AVCodecContext:
  private val offsets =
    val res = Array.ofDim[Int](151)

    res(0) = sizeof[Ptr[AVClass]]
    res(1) = res(0) + sizeof[CInt]
    res(2) = res(1) + sizeof[AVMediaType]
    // ...
    res(150) = res(149) + sizeof[Ptr[Ptr[AVFrameSideData]]]

  extension (struct: AVCodecContext)
    inline def av_class: Ptr[AVClass] =
      !struct.at(offsets(0)).asInstanceOf[Ptr[Ptr[AVClass]]]
    inline def log_level_offset: CInt =
      !struct.at(offsets(1)).asInstanceOf[Ptr[CInt]]
```

This must be fine, right?


YOU SEGFAULT
ALIGNMENT WILL CONTINUE UNTIL OFFSETS IMPROVE

The CPU in modern computer hardware performs reads and writes to memory most efficiently when the data is naturally aligned, which generally means that the data's memory address is a multiple of the data size

https://en.wikipedia.org/wiki/Data_structure_alignment

Enums

C doesn't have enums, it has sparkling numbers

```
typedef enum {
    A, B, C, D
} Letter;

void my_func(Letter l ) {
    printf("%d\n", l);
}

int main() {
    my_func(A);
    my_func(B);
    my_func(C);
    my_func(0×12312312);
    my_func(-5);
    my_func(A - B * D);
}
```

All of those work and compile without warnings even under `-Wall`.

Some C libraries define unions for documentation purposes and then not use them in actual functions or structs.

C enums in Scala Native

We can define them whichever way we want

```
opaque type Letter = CUnsignedInt

object Letter extends _BindgenEnumCUnsignedInt[Letter]:
  given _tag: Tag[Letter] = Tag.UInt
  inline def define(inline a: Long): Letter = a.toUInt
  val A = define(0)
  val B = define(1)
  val C = define(2)
  val D = define(3)
  def getName(value: Letter): Option[String] =
    value match
      case `A` ⇒ Some("A")
      case `B` ⇒ Some("B")
      case `C` ⇒ Some("C")
      case `D` ⇒ Some("D")
      case _ ⇒ _root_.scala.None
  extension (a: Letter)
    inline def &(b: Letter): Letter = a & b
    inline def |(b: Letter): Letter = a | b
    inline def is(b: Letter): Boolean = (a & b) == b
```

C macros

Not like the warm and cozy macros we have at home

Just yeet tokens at the parser and see what happens:

```c
#include <stdio.h>

#define HELLO + y - z; printf("z = %d\n", z);

int main() {
    int y = 25;
    int z = y HELLO;
    int t = y + y - t;
    printf("t = %d\n", t);
}
```

- Can you read this?
- Can you guess what it prints?

C macros

We will probably never support even a fraction of what C macros can do

- Constants are doable
- This is not:

```
#define luai_openlibs(L)  \
  {  luaL_openlibs(L); \
     luaL_requiref(L, "T", luaB_opentests, 1); \
     lua_pop(L, 1); }
```

Solution: glue code again!

Shipping glue code for this has the benefit of involving C compiler in verification stage

The river of pain

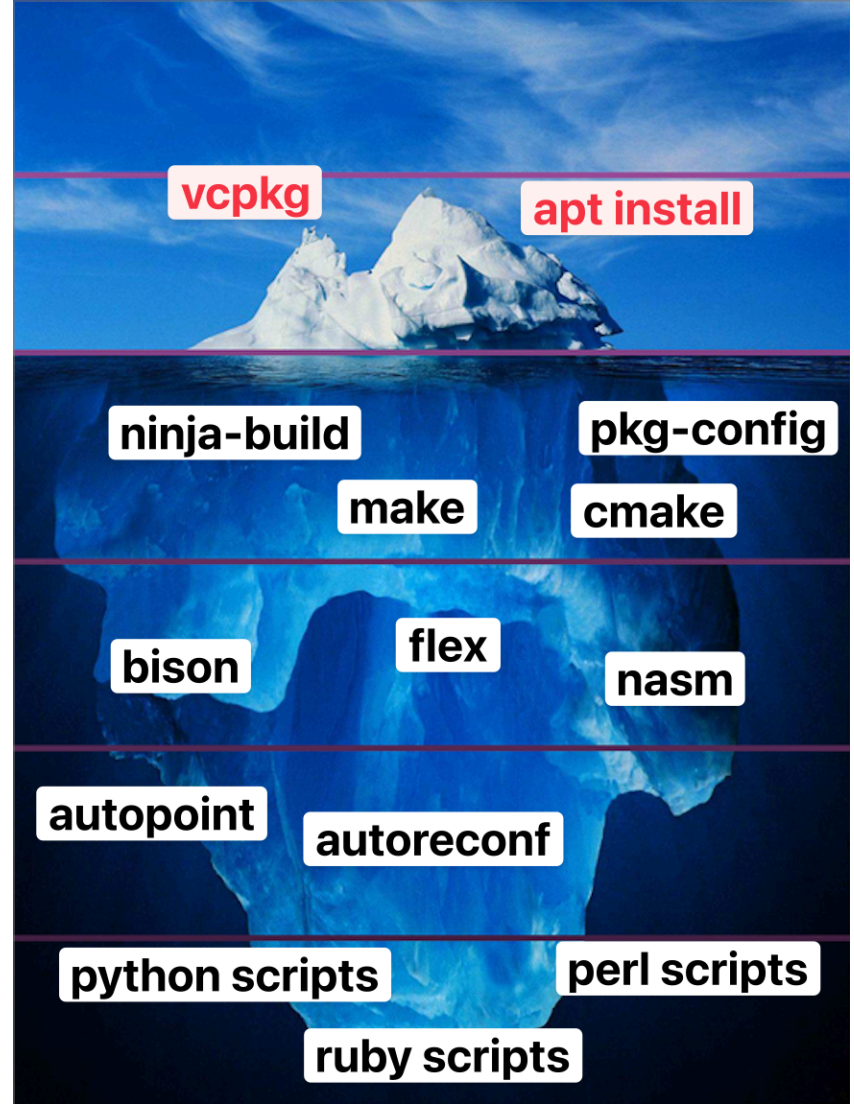
It runs deep

- Compiler-specific attributes
- Struct packing
- Bitfields
- Flexible Array Members
- Platform differences
  - Some structs have the same name and purpose but different fields
  - Proposal: https://github.com/keynmol/cross-platform-struct-encoding-scala-native
- Static linking with `libclang`
- Nested anonymous unions/structs
- C `inline`
- **Verifying that our encoding is correct to the byte**

The river of pain

We didn't even talk about dependency management!

- My current direction is using Vcpkg via https://github.com/indoorvivants/sn-vcpkg
- Everything still sucks



Side by side (opening a window with SDL3)

Scala

```
SDL_Init(SDL_INIT_VIDEO)
val window = SDL_CreateWindow(
  c"Hello SDL",
  1024,
  768,
  0×0000000000000002.toUInt
)
assert(window ≠ null)
var done = false
val event = stackalloc[SDL_Event]()
while !done do
  while SDL_PollEvent(event) do
    if (!event).`type` == SDL_EventType.SDL_EVENT_QUIT.uint
    then done = true

SDL_DestroyWindow(window)
SDL_Quit()
```

C

```
SDL_Window *window;
bool done = false;
SDL_Init(SDL_INIT_VIDEO);
window = SDL_CreateWindow(
    "An SDL3 window",
    1024,
    768,
    SDL_WINDOW_OPENGL
);
assert(window ≠ null)
while (!done) {
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_EVENT_QUIT) {
            done = true;
        }
    }
}
SDL_DestroyWindow(window);
SDL_Quit();
```

Parsing markdown with cmark

```
val str = c"Hello **Scalar**!"
val node = cmark_parse_document(str, libc.string.strlen(str), 0)

def iterateNodes(node: Ptr[cmark_node]): Unit =
  def go(nd: Ptr[cmark_node]): Unit =
    val iter = cmark_iter_new(nd)
    var ev_type: cmark_event_type = cmark_event_type.CMARK_EVENT_NONE

    inline def stop =
      ev_type = cmark_iter_next(iter)
      ev_type == cmark_event_type.CMARK_EVENT_DONE

    while !stop do
      val cur = cmark_iter_get_node(iter)
      println(
        s"Node: ${fromCString(cmark_node_get_type_string(cur))}, ev_type: ${ev_type}, address: ${cur}"
      )

    end while

  end go

  go(node)
end iterateNodes
```

Iterating branches with libgit2

```
git_libgit2_init()
val ref = alloc[Ptr[git_repository]](1)
val res = git_repository_open(
  ref,
  c"./"
)

stdio.printf(c"Repo path: %s\n", git_repository_path(!ref))

val it = alloc[Ptr[git_branch_iterator]](1)
git_branch_iterator_new(it, !ref, git_branch_t.GIT_BRANCH_REMOTE)

val gitref = alloc[Ptr[git_reference]](1)
val branch_type = alloc[git_branch_t](1)
val str = alloc[Ptr[CChar]](100)

while git_branch_next(gitref, branch_type, !it) == 0 do
  git_branch_name(str, !gitref)

  stdio.printf(c"Branch: %s\n", !str)

git_libgit2_shutdown()
```

Sending Redis commands with hiredis

```
val c = redisConnect(c"127.0.0.1", 6379)

Zone:
  if c == null then throw new Exception("Failed to allocated redis context")
  else if (!c).err ≠ 0 then
    throw new Exception(fromCString((!c).errstr.at(0)))
  else
    // The cast is actually a quirk of Redis API - it returns `void *`
    val reply = redisCommand(c, c"SET foo bar").asInstanceOf[Ptr[redisReply]]

    assert(fromCString((!reply).str) == "OK")

    println((!reply).`type`)

redisFree(c)
```

Sending queries to Postgres with libpq

```
val conninfo =
  c"postgresql://postgres:mysecretpassword@localhost:5432/postgres"

given conn: Ptr[PGconn] = PQconnectdb(conninfo)

def exit_nicely(using conn: Ptr[PGconn]) =
  PQfinish(conn)
  sys.exit(1)

def execute(statement: CString, expected: ExecStatusType)(using
    conn: Ptr[PGconn]
): Ptr[PGresult] =
  var res = PQexec(conn, statement)
  if PQresultStatus(res) ≠ expected then
    stdio.printf(c"Failed to execute %s: %s\n", PQerrorMessage(conn))
    PQclear(res)
    exit_nicely
  res
end execute
```

Want to continue the party?

https://sn-bindgen.indoorvivants.com/

Binding generator for Scala 3 Native, CLI and SBT plugin

```
$ bindgen --header hello.h --package hello_scalar --macros SCALAR_*
```

Template: https://github.com/indoorvivants/sn-bindgen-template

Want to continue the party?

https://sn-bindgen-web.indoorvivants.com/

100% Scala Native backend (https, cats-effect, fs2, skunk), 100% Scala.js frontend, 100% Pulumi Scala 3 deployment to Kubernetes

Want to continue the party?

https://github.com/indoorvivants/sn-bindgen-examples

Ready made bindings and examples of various C libraries

Curl, DuckDB, ffmpeg, JNI, lua, mysql, openssl, postgres, redis, rocksdb, SDL2/3, sqlite3, tree sitter, cmark

Serves as community build for continuous sn-bindgen testing

Conclusion

- Scala Native has decent tools for basic C interop
- A lot of the gaps can be fixed in userland through compile-time constructs
- C libraries are not *as* scary as I remember them

Farewell

Scala Native needs ambition, resolve, and focus from the entire Scala community

Where is our direct style pure HTTP stack?
Where is our pure TLS implementation?
We gotta be strong, and we gotta be fast. And we gotta be fresh from the fight