Extracted from:

Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM

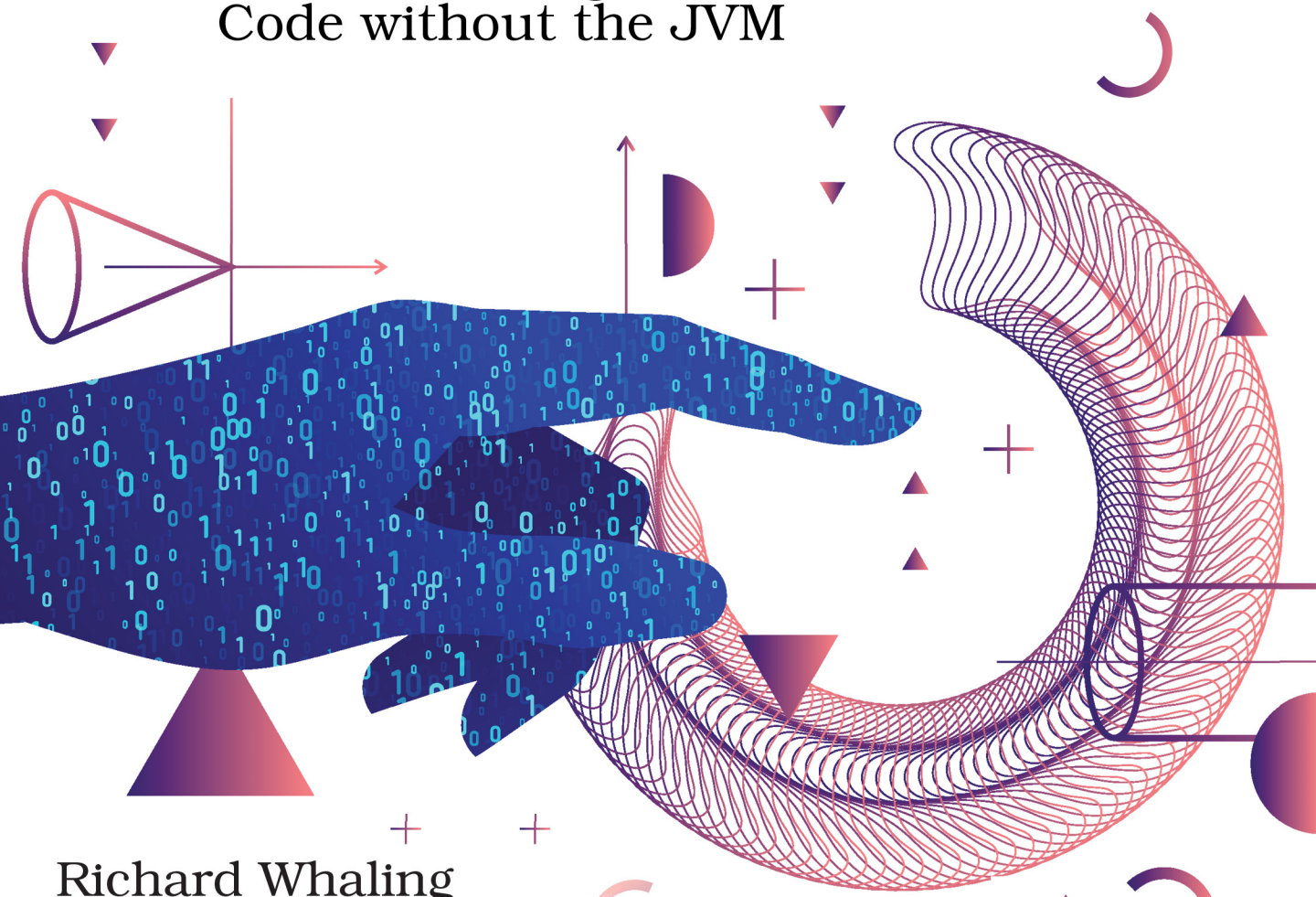
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM



Richard Whaling
Foreword by Denys Shabalin, Author of Scala Native
Edited by Katharine Dvorak

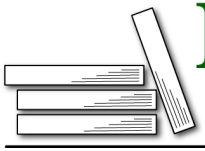
Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM

Richard Whaling

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-622-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2020

The Stack and the Heap

Memory addresses in a process are *virtualized* and *segmented*. A piece of hardware called a *memory management unit* (MMU) translates between the addresses that our program can see and the much more complex hardware devices, all while ensuring isolation as multiple programs run simultaneously. It does this by assigning chunks of memory, called *pages*, to specific positions in the address space of a single program. For example, if there are two programs running at the same time on my computer, they can both have access to totally different objects with the memory address 0x12345678, because the OS has *mapped* different pages of RAM to that address.

A running program with an isolated address space is called a *process*. (You'll learn more about processes in Chapter 4, Managing Processes: A Deconstructed Shell, on page ?.) It's also possible to create concurrent programs where multiple execution contexts occur simultaneously and share a single address space: such execution contexts are called *threads*, and although they're common in JVM programming, we'll make little use of them in this book.

What does this mean for us? As programmers, we see the impact of the OS's memory management practices in the properties of different kinds of memory, which live in different ranges of the address space. Typically, we'll encounter memory addresses in the following three segments:

- The *stack* segment, which holds short-lived local variables that live as long as the function that calls them.
- The *text* segment, which is a read-only segment containing the instructions for the program itself.
- The *data* segment or *heap* segment, which holds everything else.

You'll learn more about how to use the text segment later in this chapter. For now, we need to figure out how to allocate memory on the heap and what constraints we'll need to keep in mind to use it correctly. The most important rule to remember with heap memory is this: once heap memory is allocated, we're *responsible for it* in every sense. (You'll learn more about exactly what that means shortly.)

Heap memory management is one of the fundamental problems of systems programming. You could spend years studying and refining various techniques, and this chapter will only scratch the surface. Errors in memory allocation can also be extremely hard to test, isolate, and debug. For this reason, I strongly advocate the approach of using manual allocation *only where neces-*

sary, and instead relying on higher-level techniques wherever performance constraints aren't extreme.

All that being said, manual memory management allows for a level of cleverness and optimization in program design unmatched by any higher-level language, and a well-designed allocation scheme can be both elegant and highly performant.

malloc

The fundamental function for allocating heap memory is malloc, which has a straightforward signature:

```
def malloc(size:Int):Ptr[Byte]
```

malloc's simple signature hides a great deal of subtlety. First of all, you may notice that unlike stackalloc, malloc doesn't take a type parameter; instead it just takes the number of bytes requested and returns a pointer to an unused region of memory of the requested size. In theory, malloc can fail, but that's essentially impossible on a modern system—your operating system will generally terminate the process before malloc itself returns an error.

Since malloc returns a Ptr[Byte], it's pretty straightforward to apply it to handling the sort of string and buffer data we used in the previous chapter. But what if we want to allocate space for a struct or an array of structs?

To create space for structured data, we can use two techniques in tandem: we can use Scala Native's built-in sizeof[T] function to compute the size, in bytes, of the data type we need, and then multiply that size by the number of elements we wish to store. However, we might request space for ten integers like this:

```
val 10_ints = malloc(10 * sizeof[Int])
```

But the type of 10_ints is still going to be Ptr[Byte], not the Ptr[Int] that we want!

The solution to the problem is a *cast*, an instruction to the compiler to reinterpret the type of a pointer. A *cast* is not the same as a *conversion*—no transformation is applied to the contents of a variable when we cast it. The most common use of a cast is to convert pointer types to and from Ptr[Byte], not only for use with malloc, but also for quasi-generic functions such as malloc (and qsort) in which Ptr[Byte] effectively serves as a catch-all reference type, analogous to Scala's AnyRef.

To cast our 10_ints to our desired type of Ptr[Int], we would do this:

```
val ngram_array = malloc(10 * sizeof[Int]).cast[Ptr[Int]]
```

Whew! But before we can safely use malloc to store our data, we'll need to look at two more functions that are essential for working with heap memory: free and realloc.

free

When we allocate memory with malloc, that block of memory will stay around for as long as we need it; however, we have no garbage collector to rely on with malloc, so if we know we're done with a pointer, we need to call free() to give it back. Failing to call free() in a long-lived program will result in a *memory leak*, in which your program continues to allocate new memory without ever reclaiming unused space until your computer's RAM is totally exhausted.

free works like this:

```
def free(p:Ptr[Byte]):Unit
```

But, there are a few catches that aren't evident in the signature. First of all, the pointer passed to free *must* be a valid pointer created by malloc *that has not been freed previously*. Calling free twice on the same pointer is a severe error, as is calling free on a pointer on the stack, a function pointer, or anything else not managed by malloc.

malloc and free: Not System Calls

malloc is another good example of a major C standard library function that isn't a system call. malloc, internally, keeps track of the total size of the heap, plus the size and location of every allocated chunk of memory within it. When we ask for more memory, malloc will either give us an available chunk of memory within the heap or it will *grow* the heap using the more obscure system call, sbrk, to change the size of the heap segment itself.

Likewise, free never makes a system call, but it can still be slow—maintaining malloc's map of allocated memory is expensive in a data-intensive program.

The best way to handle malloc and free is to structure your program so that every pointer created by malloc has a clear lifecycle, such that you can be assured that free will be called in an orderly way for every call to malloc. If it isn't possible to do this, it may be worth either redesigning your program to support this pattern, or else rely on a garbage collector instead.

One edge case can simplify things: if you have a short-lived program, there's relatively little benefit to calling free exhaustively *immediately before your program terminates*. All memory, stack or heap, will be released when your

process exits, and it can sometimes be useful to write a short-lived program that simply mallocs until completion.

One more trick we can do: realloc will allow us to *resize* a block of memory returned by malloc. Sort of.

realloc

realloc has the following signature:

```
def realloc(in:Ptr[Byte]):Ptr[Byte]
```

realloc takes a pointer as its sole argument and on success, returns a pointer to a region of memory of the desired size. It may or may not resize the region in-place, depending on whether there is enough space for it in adjacent areas of memory. If realloc returns a pointer to a new address, realloc will internally copy your data from the old space to the new, larger area and free the old pointer. However, this can be dangerous if you have *any* outstanding pointers to the old region somewhere in your program because they have now been invalidated in a nondeterminstic way.

In other words, realloc is powerful, but dangerous, and it can also be expensive. When we use it, we'll want to use it as little as possible.

As you may have observed, all of these heap memory management functions have quirks, caveats, and dangers associated with them. Wherever possible, we'll try to isolate this code, and provide a clean abstraction over it, rather than let our malloc operations and such spread all over our codebase.

Zones

The stack and the heap have been with us for about as long as we've had modern computers; but, Scala Native also provides a newer technique for semiautomatic memory management, in the form of the Zone object. A Zone is an object that can allocate memory and track its usage for us; in other words, it automatically cleans up memory like the stack does, but it can work in the much larger heap segment. It also isn't bound to the scope of a single function —instead, a Zone object can be passed as an implicit argument, which allows many nested functions to share a single Zone, following this sort of pattern:

```
def handler(s:CString):CString = Zone { implicit z =>
  val stringSize = strlen(s) + 1
  val transformed_1 = native.alloc[Byte](stringSize)
  inner_handler(transformed)
}

def inner_handler(s:CString)(implicit z: Zone):CString = {
```

```
  // the inner handler can allocate as much as it wants here,
  // and it will all get cleaned up when the outer handler returns.
}
```

Zones are a great alternative to the stack for short-lived objects, and we'll continue to explore ways to use them, especially in the second half of the book.

But for now, let's shift gears and try to plan out *how* to store our data in the big memory chunks that malloc and its friends can give us.