9

Memory and Pointers

This chapter covers

* Memory failures
* A practical look at memory management
* The Pointer type
* The OwnedPointer type
* The ArcPointer type
* The UnsafePointer type
* Structs with pointers
* Collections using pointers

The topics of memory management and pointers were skimmed over throughout the preceding chapters (for example § 1.2.1, § 3.2.6 and § 7.2). We talked about stack and heap memory, the address of a variable and pointers to variables, the ownership model and the lifetime of variables, how Mojo uses keywords like read, mut, owned and the transfer (^)operator to determine what can be done with the variable, the alignment of struct instances in memory, and finally about the \_\_init\_\_, \_\_copyinit\_\_, \_\_moveinit\_\_ and \_\_del\_\_ methods in the lifecycle of an object.

In this chapter, we’ll finally deal with the topic of memory and pointers in depth and provide you with some additional concepts. But above all, we’ll learn concrete methods to work with memory and pointers at a system level, showing that Mojo works indeed as a low-level system language. We’ll discuss the different kinds of pointers, going from safe to unsafe territory, and showing numerous examples of using pointers in code.

*Undefined* (or *unsafe*) behavior (abbreviated to UB) can only occur in rare situations where the outcome of the compiler cannot be predicted.

We’ll explore all kinds of pointers in this chapter. Only one of these, the UnsafePointer, is unsafe and can exhibit UB, but it is also the most performant to use.

The goals of getting the highest performance and ultimate code safety oppose each other when we work at system level. In particular, the *memory* module of the stdlib deals with such unsafe types and code. But unlike C/C++, Mojo emphasizes memory safety. We’ll point out the unsafe behavior and tell you how to deal with it.

Many collection types discussed in the previous chapter have an underlying pointer that points to the data the collection is holding in memory. We’ll see examples of their use.

To follow along, use Magic to create a project called memory\_and\_pointers and cd into it. All code examples used in this chapter can be found in the code repo at <https://github.com/Ivo-Balbaert/Mojo_in_Action/tree/main/9_Memory_and_Pointers>.

We’ll start with a discussion of how code can fail to adequately work with memory. Then, after arguing why pointers are important, we’ll continue with a discussion of UnsafePointer, because we’ve met it before in code and we need it in almost every example in this chapter. As the name suggests, it is also the pointer type we should be most careful with.

* 1. Memory failures

Stack memory is automatically managed. Mojo also performs automatic memory management for the heap, freeing up memory that is no longer needed as soon as possible. With a few exceptions, most notably the UnsafePointer, it does so for all stdlib data types. You are responsible for cleaning up data from the structs you defined, especially when they contain pointers. Be sure to define an appropriate destructor \_\_del\_\_.

Without getting you into panic mode, let’s discuss what can go wrong with memory and pointers.

* Locations or regions in memory that don’t have been initialized with values (yet) are called *uninitialized memory*. They mostly contain random data from previous memory usage. After a fresh memory allocation, that memory is uninitialized. To initialize it, value(s) must be stored in it. Accessing uninitialized memory is unsafe and can cause UB. Dereferencing a pointer that points to uninitialized memory is UB.

If memory is not freed consistently (for example because a specific \_\_del\_\_ destructor is not executed) it is wasted. If this happens a lot, memory gets exhausted: the program occupies more memory than it needs and eventually runs out of memory and crashes. This situation is called a *memory leak.* This is mostly due to pointers, when the memory space of their corresponding data is not freed.

* A pointer that doesn’t contain a memory address for attached data or that points to an invalid memory location is a *null pointer*, it is *nullable.* This is effectively an invalid pointer: dereferencing such a pointer most likely crashes the program it contains.
* Even after freeing a pointer’s memory, but the pointer itself is not (yet) destroyed, we have a *dangling pointer*: the address still points to its previous location, but the memory is no longer allocated to this pointer. Dereferencing a pointer in this state (a *use-after-free*) results in UB.
* Problems can also occur when a pointer is freed more than once (a *double free*). This can result in corrupted memory management data structures and lead to all kinds of other unexpected behavior, which could include arbitrary code execution and program crashes.
* The language needs to address problems of *concurrent mutation*. If the *data* our pointer points to can be changed by some other function or in another thread, it gets *corrupted*. Mojo ensures through its lifecycle model that data can *only be mutated by the variable that owns the data*.

S*afe pointers* are defined to never get you into trouble: they exhibit no memory errors. They can’t access uninitialized memory, and they can’t be a null pointer.

In your career, or even ordinary life, you surely have encountered many computer problems caused by one or more of these failures.

In some programming languages (mostly so-called higher-level languages) pointers even don’t exist. That automatically gives them a lot so safety. But, in these languages, you’re not able to write low-level system code and the goal of highest performance is beyond reach. Let’s briefly discuss why you would need pointers.

We’ll see examples of executing code in these unsafe states in § 9.3.

* 1. The need for pointers

To do low-level things and get the best performance, we need direct access to memory locations, just like in C and other low-level languages. Mojo gives you the power to do whatever you want with pointers.

Pointers offer the following advantages:

* *Memory Efficiency*: Pointers allow programs to use memory more efficiently. Instead of copying and storing entire data structures, a program can use pointers to reference these structures. This is particularly useful when dealing with large data structures.
* Dynamic Memory Allocation: Pointers enable dynamic memory allocation. This means that memory for variables can be allocated and deallocated during runtime, which provides flexibility and control over the memory usage of your program.
* *Data Structures and Algorithms*: Pointers are essential for creating complex data structures like trees and linked lists. They also enable efficient implementation of various algorithms.
* *Function Arguments*: Pointers can be used to pass arguments by reference to a function. This means that the function can modify the original data, not just a copy of it. In Mojo however, the default is that arguments are passed as borrowed, they are immutable references (see § 6.1.3).  Use the keyword owned when passing an argument, enabling you to transfer ownership of a pointer to another function.

Mojo has 4 types of pointers: Pointer, OwnedPointer, ArcPointer and UnsafePointer. These types are all *generic:* they can point to items of any type, be it on the stack or on the heap. Pointers are usually kept in the stack, pointing to the data in the heap. If needed, data can be copied from the heap to the stack, using the pointer to fetch it.

As you know by now, pointers don’t store a value directly, they only store the value’s address in memory (a reference). The pointer has to be dereferenced with the [] operator to get the value. The value pointed to by a pointer is called the *pointee*. This is used in the naming of the methods of UnsafePointer, like init\_pointee\_copy. As a reminder, the principle of a pointer is shown in Figure 9.1:

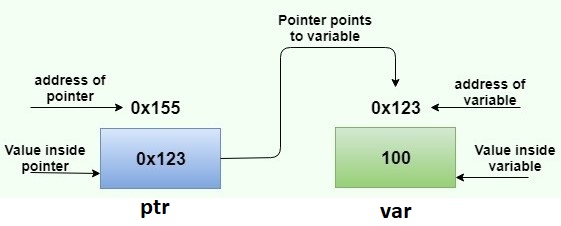


Figure 9.1 – Pointer ptr contains the address 0x123 of variable var. var contains the value 100.

A pointer ptr pointing to the memory address associated with a variable var**,** ptr contains the memory address 0x123 of the variable var. The address of ptr itself is irrelevant, unless you have a case of a pointer pointing to a pointer. However, such complexity is not encouraged: more indirections in code decrease performance.

Some pointer types, like the UnsafePointer which we are about to discover, can allocate memory to store their pointees. Other pointers can only point to pre-existing values. Also Memory allocation must either be done *explicitly*, for example, for an UnsafePointer. Or it can be done *implicitly*: it is performed automatically when initializing a pointer with a value.

To those developers coming from Python: don't be afraid of pointers. When you use them with care, they will serve you well. Let’s get our feet wet!

* 1. Unsafe Pointers

Let’s kick in the door with the pointer type that is unsafe by definition and, when not treated carefully, can get you into most of the problems discussed in § 9.2.2: the UnsafePointer, which is similar to the raw pointer used in C/C++.

We worked with UnsafePointer before (in § 3.2.6) to get and compare the addresses of variables. It points to one or more consecutive memory locations which can be uninitialized.

Why can this pointer result in UB?

* It has methods for allocating memory, initializing, accessing, changing and destroying stored values (deallocating memory), giving you a lot of control, but they are all unsafe: the developer takes full responsibility, guaranteeing that memory gets allocated and deciding when it can be freed correctly and as soon as possible. In other words: Mojo doesn’t keep track of the lifetime the pointer points to, you have to do it!
* An UnsafePointer can be a nullable pointer, not pointing to any data. And if it points to data, these can be uninitialized.
* Did we mention that this pointer does no bounds checking? You have index access to locations that do not belong to your pointer’s data. So, you can get into all sorts of memory bugs that exist in C and C++.

That’s why you should only use this pointer when none of the other pointer types works for the job at hand. But you will probably use it in the following cases:

* When *performance* is crucial, for example when using array- or matrix-like structures such as List or Tensor. UnsafePointer can let you pre-allocate a big chunk of memory, and initialize it with values only when they are stored in the collections.
* When interacting with low-level interfaces to the operating system or hardware.
* When you interact with external libraries in other languages such as C++ or Python, passing data buffers to and from the external library.

Here is the start of its definition (we leave out with … details which are less important now):

**@register\_passable**(trivial)

struct UnsafePointer[type: **AnyType**,… ]

From it we see that an unsafe pointer is register passable (rather evident: an address is an unsigned integer), and the pointee can be of any type. UnsafePointer is *generic*, it can point to a String, a struct instance, a List, a Tensor, or all kinds of collections.

Because you have to carefully manage the lifetime of the data pointed to by your UnsafePointer, it is very important to understand its lifecycle (see unsafe\_pointers.mojo).

9.3.1 The states of an UnsafePointer

Let’s first see how you can set up an UnsafePointer ptr:

from memory import UnsafePointer  # A

def print\_address(ptr: UnsafePointer): #A2

    print(Int(ptr))

fn main() raises:

    var ptr: UnsafePointer[Int]  # B

    ptr = UnsafePointer[Int]()   # C

print\_address(ptr)  # => 0

    ptr = UnsafePointer[Int].**alloc**(7)  # D

#A Import UnsafePointer  from module memory

#A2 A function to print out the addres ptr points to

#B Declare ptr as an UnsafePointer to Int data

#C Making a null pointer: don’t do this!

#D Allocates memory for 7 Int’s

We will use our function print\_address from before to print out the address (which is an integer) the pointer contains. In the first line, ptr is only *declared*, it is not initialized, so no memory is allocated yet. Adding () calls a constructor: the pointer itself is allocated, but without data: this is a *null pointer*. You better not do this, unless there is a good reason to:

A black and white image of a square with letters

Description automatically generated

Figure 9.2 – A null pointer ptr, its contents is 0, which is an invalid memory address for data

print\_address gives 0 (0x0) as memory address for the data. If you try to get a value out of this pointer with ptr[], the program crashes with a segmentation fault, which means the given memory address doesn’t exist:

var value = ptr[]

print(value) # => Mojo crashes: segmentation fault

We can check on the nullability of a pointer with an if – else test (see null\_pointer\_test.mojo):

from memory import UnsafePointer

fn main():

    var ptr: UnsafePointer[Int]

    ptr = UnsafePointer[Int]()  #A

    var n = 42

    var ptr1 = UnsafePointer[Int].address\_of(n) #B

**if ptr:**   #C

        print("ptr is not a null pointer", end= " - ")

        var value = ptr[]

        print("its value is:", value)

**else: #D**

        print("ptr is a null pointer")

    # => ptr is a null pointer

# A Create a null pointer

#B Create a pointer pointing to a value

#C The if-branch is executed when ptr is a valid pointer

#D The else-branch is executed when ptr is a null pointer

Testing a null pointer with if returns False, a valid pointer returns True. When changing if ptr: in the code above to if ptr1:, the output is:

ptr1 is not a null pointer, its value is: 42

Finally, using the alloc method which takes an integer argument, we allocate memory space on the heap for let’s say 7 integers of type Int, which are ptr’s data:

var ptr = UnsafePointer[Int].alloc(7) #A

#A Declaring the pointer and allocating memory for it

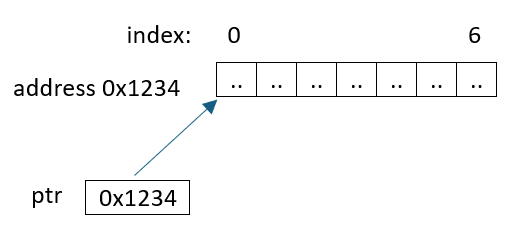


Figure 9.3 – The pointer ptr has now memory allocated to it, the size is 7 \* 8 = 56 bytes, but it contains no data yet!

The pointer ptr now points to *uninitialized memory* space, like a variable that has been declared, but not yet initialized. Note that by using var here, we simultaneously declared *and* allocated memory for the pointer, an advisable way to avoid null pointers. Trying to dereference it with [] gives UB. In our case, it results in 0, but it could have been an unreliable random left-over number from a previous computation:

var value = ptr[]

print(value) # => 0

  print\_address(ptr2)  # => 90148140228608

We better initialize the memory then and make ptr point to valid data that already exists, like value. This can be done in three different ways, first we’ll use the *static method* address\_of(value) we used before, which makes the pointer’s address the same as the address of an existing value:

var value = 108

    ptr = UnsafePointer[Int].**address\_of(value)** #A

    print\_address(ptr)  # => 140727043935248

 print(ptr[])  # =>  108 #B

#A Pointer ptr points to an existing value

#B Get the pointer’s value (the pointee) with the [ ] operator

Note that for this method it is not necessary to allocate memory for the pointer beforehand, the value’s memory space is already allocated and becomes the pointer’s memory. That’s why it is especially useful for getting a pointer to a value on the stack, but it also can be used for heap values.

Schematically:

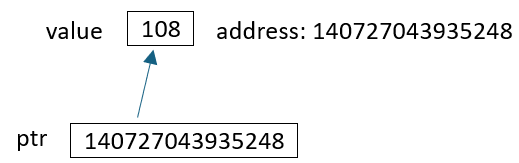


Figure 9.4 – Pointer ptr gets the address of an existing value

Another way is to *copy* an existing value to the pointer’s allocated memory:

ptr**.init\_pointee\_copy(value)** #A

    print\_address(ptr2)  # => 90148140228608

print(ptr[])  # =>  108

#A Copy an existing value to ptr’s data location 90148140228608

Notice in the schema that value now exists on two memory locations:

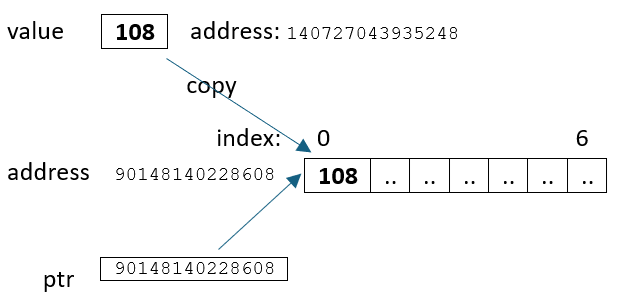


Figure 9.5 – init\_pointee\_copy copies the value to the pointer’s memory

The third way is to *move* an existing value to the pointer’s allocated memory:

ptr**.init\_pointee\_** **move(value^)** #A

    print\_address(ptr)  # => 90148140228608

print(ptr[])  # =>  108

#A Move an existing value to ptr’s data location 90148140228608

Notice in the schema that value now exists only in the pointer’s memory:

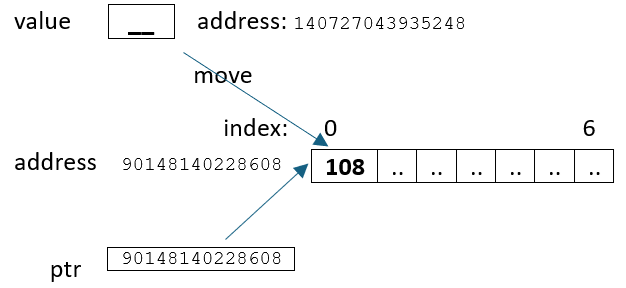


Figure 9.6 – init\_pointee\_move moves the value into the pointer’s memory

Notice we get the warning: *transfer from a value of trivial register type 'Int' has no effect and can be removed*. Because Int is a trivial type, the move is optimized out by the compiler. But we illustrated how the move effectively works for memory types.

With all three methods the value the pointer points to given by ptr[] is 108.

We got the value of the pointer with ptr[], but we can also change it with the same notation: ptr[] = new\_value, we effectively mutate the pointee:

**ptr2[] =** 256 #A

    print(ptr2[])  # =>  256

#A Changing the value a pointer points to

Note that changing the value of a pointer like this is NOT the way to initialize a pointer. Doing so results in UB, as we can see in the following example:

    ptr\_str = UnsafePointer[String].alloc(1)

**ptr\_str[] = "Bad way to initialize!"   #A**

    print(ptr\_str[])  # => #B

    ptr\_str.init\_pointee\_move("Good way to initialize")

    ptr\_str[] += " and to change"   #C

    print(ptr\_str[])  # => Good way to initialize and to change

    print\_address(ptr\_str)  # => 21021715349504

#A Undefined behavior

#B Program crashes!

#C This works

After the initialization with a String, we see that ptr\_str[] += works. The pointer’s memory space dynamically expands without having to call alloc() again.

Now comes the crucial part: how to dispose of a pointer properly and completely. This needs two phases:

1. *Destroy or remove the value(s) of the pointee:*

This is the biggest part of memory that needs to be regained. Here you can also use several methods to call on the pointer. These methods call the *destructor* on the stored values in the pointee. For these to work, the pointer cannot be a null pointer, and the memory to be freed must contain a valid instance of the data type of the pointer.

The most efficient method to call on the pointer is destroy\_pointee:

 ptr.destroy\_pointee()

It calls the destructor for the pointee and leaves the freed memory space uninitialized.

Other methods are take\_pointee() and move\_pointee\_into(dst), which are somewhat less efficient because they execute a \_\_moveinit\_\_ operation.

If you call the first method like this: ptr.take\_pointee(), you get the message: *'Int' value is unused*. That is because take\_pointee() moves the pointee value from the location pointed to by the pointer (calling \_\_moveinit\_\_) and consuming the value (no copy is left). Afterwards, the pointer’s memory location is uninitialized. If you want to keep the value somewhere else, call it as: var i1 = ptr.take\_pointee()

The second method is called like this:

  var dst = UnsafePointer[Int].alloc(1)

    ptr.**move\_pointee\_into(dst)**

Here dst is an UnsafePointer that is NOT a null pointer and is of the same type as ptr.

1. *Free the memory:*

After destroying the memory allocated by the pointer in the first step, you need to free it. This is done by calling free() on the pointer:

ptr\_str.**free()**

    print\_address(ptr\_str)  # => 21021715349504

  print(ptr\_str[])  # =>   # => �@���to initialize���

Freeing a pointer more than once is UB and can crash the program. Note how the pointer still contains the address of the memory it pointed to. Trying to get the previous value by dereferencing results in UB (in this example you can still see part of the previous String intact).

Freeing a null pointer results in UB. If you’re not sure whether a pointer is a null pointer or not, test it before freeing it, like this: if ptr: ptr.free()

Smart pointers

Why do we need two steps to completely cleanup an UnsafePointer and its memory? The reason is that an UnsafePointer *doesn’t own its memory* (the pointee). The pointee needs to be separately destroyed, as is the case for all non-owning pointers.

On the other hand, *smart pointers* own their pointees, which means that the value they point to is deallocated when the pointer itself is destroyed. UnsafePointer is not a smart pointer, the OwnedPointer and ArcPointer types which we’ll discuss in a later section own their pointee and are smart pointers.

NOTE

Calling the free() method on the pointer ptr in our example crashes the program. The reason is that the integer that the pointee of ptr contains is stored on the stack by optimization. Freeing something on the **stack** yields UB, in our case aprogram crash, No alloc() was called to create the ptr’s memory. You can only free() memory that was created with alloc().

To summarize, a pointer goes through the following states:

STATE 1: "Birth"

Declaring the data type T: var ptr = UnsafePointer[T]

Attaching data to the pointer with: ptr = UnsafePointer[Int].address\_of(value)

OR

Declaring the data type T and allocating memory:

var ptr = UnsafePointer[Int].alloc(1)

Copying or moving date into the pointer’s memory with:

ptr.init\_pointee\_copy(value) or ptr.init\_pointee\_move(value^)

***STATE 2: "Life"***

Get the pointer’s value with ptr[], change it with ptr[]=, and do other computations with it.

***STATE 3: "Destruction and Death, in 2 steps"***

1. Destroy the value with ptr.destroy\_pointee(), or var i1 = ptr.take\_pointee() or ptr.move\_pointee\_into(dst)
2. Free the memory with: ptr.free()

WARNING

If you forget to implement STATE 3, Mojo doesn’t warn you!

**Aliasing pointers**

It could happen that code contains multiple copies of a pointer accessing the same memory, which are called *aliasing pointers*. In such a case, you need to ensure that you call free() on only one of the copies. Freeing the same memory twice is an error.

NOTE

If the notation ptr[] to get the pointee’s value reminds you of an index, you’re correct! ptr[] works the same way as ptr[0]. We can use an index to change any location of a ptr, for example: ptr[3] = 42. But be warned, there is no bounds checking.

Experiment 9\_1 – An UnsafePointer to a String

Use the same code as in the example to make the UnsafePointer point to a String. Implement the complete lifecycle!

9.3.2 Storing multiple values

You might have wondered about this line in the previous code example:

ptr = UnsafePointer[Int].alloc(**7**)

We allocated memory locations for 7 Int values, but we only used one. Let’s see how we can use them all. We’ll also use it to store Strings instead of Int, to make sure we allocate on the heap. (see unsafe\_pointers2.mojo):

fn main() raises:

    var ptr = UnsafePointer[String].alloc(7) #A

    print\_address(ptr)  # => 127187938189312 #B

    var value = "Mojo"

    ptr.init\_pointee\_copy(value)

    print\_address(ptr)  # => 127187938189312

    print(ptr[])  # => "Mojo"

    print(ptr[0])  # => "Mojo"

  ptr[1] = "Mojo1" #C

    print(ptr[1])  # => "Mojo1"

    ptr[6] = "Mojo6" #C

    print(ptr[6])  # => "Mojo6"

    ptr[100] = "Mojo6"  # no bounds checking

ptr.destroy\_pointee() #D

    ptr.free()

#A Allocate memory for 7 String values

#B The first String is pointed to by ptr: this is the base-address

#C Initialize a memory location by index

#D These destroy value and memory free statements are obligatory.

We can add to or subtract an integer from a pointer, as shown here:

    print(sizeof[String]()) # => 24

print\_address(ptr)      # => 127187938189312

print\_address(ptr + 1)  # => 127187938189336

    print\_address(ptr + 2)  # => 127187938189360

    print\_address(ptr + 5)  # => 127187938189432

    ptr += 3

    print(ptr[])  # => Mojo3

    ptr -= 1

    print(ptr[])  # => Mojo2

Subsequent addresses are each 42 (bytes) higher, exactly the size of a String. All the locations are adjacent to one another, in a contiguous block. The number of locations measured from the base-address ptr is called the *offset*, which is also a method on the pointer: ptr + 2 is the same as ptr.offset(2).

This makes it easier to initialize the memory and printing out the values in a for-loop:

for offset in range(7): #A

        (ptr + offset).init\_pointee\_copy(String("Mojo" + String(offset)))

    for offset in range(7): #B

        print(ptr[offset], end=", ")

    # => Mojo0, Mojo1, Mojo2, Mojo3, Mojo4, Mojo5, Mojo6,

    #A Set the values

#B Print out the values

?? Pointer comparisons:

    print("p1 is at a lower address than p2:", p1 < p2)

    print("p1 and p2 are equal:", p1 == p2)

    print("p1 and p2 are not equal:", p1 != p2)

9.3.3 Working with SIMD values

UnsafePointer has special methods load and store to read and write SIMD values (which comprises all numerical types) from and to a pointer, (see unsafe\_pointers\_simd.mojo):

from memory import UnsafePointer

fn main() raises:

    var ptr = UnsafePointer[Float64].alloc(5)   #A

    for offset in range(5):

**ptr.store(offset, 0.0) #B**

    ptr[2] = 3.14

    for offset in range(5):

        print(**ptr.load(offset)**, end=", ") #C

    # => 0.0, 0.0, 3.14, 0.0, 0.0,

    print()

    # Freeing memory

    ptr.destroy\_pointee()

    ptr.free()

#A Allocate memory for a pointer ptr to 5 Float64 values

#B Use store in a for-loop to initialize them with value 0.0

#C Use load to read them back in

store and load work with only one value at a time. store needs the offset from the base address of the pointer and the SIMD value to be written at location (ptr + offset). load reads in the SIMD value at the given offset location.

The gather method makes (' gathers ') a SIMD vector from offsets of the current pointer (see gather.mojo):

from memory import UnsafePointer

fn main():

    float\_ptr = UnsafePointer[Float64].alloc(6) #A

    for offset in range(6): #B

        (float\_ptr + offset).init\_pointee\_copy(Float64(offset))

    offsets = SIMD[DType.int32, 4](0, 2, 4, 5) #C

    gathered\_values = **float\_ptr.gather[width=4](offsets)** #D

    print(gathered\_values)  # => [0.0, 2.0, 4.0, 5.0]

    # Now gathered\_values contains [float\_ptr[0], float\_ptr[2], float\_ptr[4], , float\_ptr[5]]

#A Create a pointer to store 6 Float64 values

#B Initialize the values with the offsets as floats

#C Create a SIMD vector of offsets

#D Use gather to load values at those offsets into a SIMD vector

We want to gather values at offsets 0, 2, 4 and 5 of float\_ptr, so we create a SIMD vector offsets with these offsets. Calling the gather method on float\_ptr with the offsets vector as argument creates a new SIMD vector gathered\_values. The width parameter specifies the size of the SIMD vector.

The scatter method distributes ('scatters ') a SIMD vector into offsets of the current pointer (see scatter.mojo):

from memory import UnsafePointer

fn main():

    var ptr = UnsafePointer[SIMD[DType.int32, 1]].alloc(10) #A

    for i in range(10): #B

        (ptr + i).init\_pointee\_copy(SIMD[DType.int32, 1](0))

    var values = SIMD[DType.int32, 4](10, 20, 30, 40) #C

    var offsets = SIMD[DType.int32, 4](1, 3, 6, 9) #D

**ptr.scatter(offsets, values)** #E

    for i in range(10): #F

        print(ptr[i], end=", ")

    # => 0, 10, 0, 20, 0, 0, 30, 0, 0, 40,

    ptr.free() #G

    #A Allocate memory for 10 integers

    #B Initialize all values to 0

    #C Create a SIMD vector with values to scatter

#D Create a SIMD vector with offsets where to scatter the values

    #E Scatter the values to the specified offsets

    #F Print all values to see the result

#G Free the allocated memory

We want to scatter the SIMD vector values at offsets 1, 3, 6 and 9 of pointer ptr, so we create a SIMD vector offsets with these offsets. Calling the scatter method on ptr with the offsets and values vectors as argument place these values at positions 1, 3, 6, and 9 in the pointer offsets range.

Similarly, there are two strided-methods, who use a constant offset to do the same operation:

* The strided\_load method reads from the offset range of a pointer using an offset ('stride ') and loads these values into a resulting SIMD vector.
* The strided\_store method writes values from a SIMD vector to the offset range of a pointer using an offset ('stride ').

XYZ

9.3.4 Bitcasting

9.3.5 Using random numbers to fill in a pointer range

Rand and randn from random: see 12.5 Random numbers

See dtypepointer1.mojo

tWEETORIAL

Tensor: rand(image.unsafe\_ptr(), image.num\_elements())

(see tensor1\_old in coll)

* 1. Low-level memory operations and Span

We know that memory space is much more restricted on the stack than it is on the heap. But on the other hand, our code can work much faster with data on the stack, than with data on the heap. Can we use the stack directly from code?

9.4.1 Stack allocation

Mojo has a function called stack\_allocation, that allows you *to allocate data space on the stack* and use this benefit of getting much quicker access to it. It needs as parameters the number of items (count) you want to allocate memory for, and their type (dtype or type). It comes in two versions, one for DType values, and one for any type:

stack\_allocation[count: Int, **dtype: DType**, /, …] -> UnsafePointer[SIMD[dtype, 1, …]

stack\_allocation[count: Int, **type: AnyType**, /, …] -> UnsafePointer[type, …]

(The parameters we left out with … have to do with alignment and address space).

Both versions return an UnsafePointer to the data. An *address space* is the memory space a pointer points to.

Here is a working example (see stack\_allocation.mojo):

from memory import stack\_allocation #A

fn main():

    var scratchpad = stack\_allocation[50, DType.int64]() #B

    scratchpad.store(0, 42)   #C

    var value = scratchpad.load(0)  #D

    print(value)  # => 42

#A Import stack\_allocation from module memory

#B Allocate 50 integers of type Int64 on the stack

#C Set values with store: store value 42 at index 0

#D Get values with load: load value from index 0

The allocation reserves memory space, but doesn’t fill them with values. For this, you need to use the store method from scratchpad, which is an UnsafePointer that points to the data. Read the value in on the given index with the load method. The memory locations in the address space scratchpad points to can be reached with the index notation (ix), where ix is the index starting at 0 and runs to count – 1, in our example 49.

If you want a more structured approach with simpler code, you can use IntArray, which provides a low-level implementation of a dynamically-sized integer array with direct memory management. (see intarray.mojo):

from layout.int\_tuple import IntArray #A

fn main():

    var scratchpad = IntArray(50) #B

    scratchpad[0] = 42 #C

  for i in range(5):

        scratchpad[i] = 42 + 1 #D

    var value = scratchpad[4] #D

    print(value)  # => 46

#A Import IntArray from module layout.int\_tuple

#B Allocate 50 integers of type Int on the stack

#C Write value 42 at index 0

#D Write values with a for loop

#E Read value from index 4

Knowing your dunder methods, you immediately see that IntArray implements the \_\_getitem\_\_ and \_\_setitem\_\_ methods. This type is also the underlying storage mechanism for IntTuple (see § 8.7.3) and related data structures, optimized for high-performance tensor operations.

The StaticTuple type discussed in § 8.1 also uses stack allocation.

9.4.2 Memory operations

When two variables or objects have the same memory address, they are said to be identical, in other words: they have the *same identity*. This can be checked with the following methods.

The is and isnot method

MBE 11

memset:

memcpy:

memset\_zero: § 12 pointers1.mojo / pointers2.mojo /

parallel\_memcpy()

Examples from ch 2 – python\_utils.mojo – Mojo 🡨> numpy

Mojo has a special type called a Span, which is very useful for all kinds of memory operations.

9.4.3 The Span type

A Span is defined as a struct in module *memory.span*. It is like a *view* on successive memory locations (contiguous memory), which the Span object does *not own*. It is used in many (low-level) operations on collections, so you will encounter it sooner than later. The type of the Span’s items has to implement the CollectionElement trait. For example, we can make a Span[Byte], which is a stream of UTF-8 encoded data.

It can be constructed in three ways:

* By giving *a pointer* to a memory location, *and a length*, *spanning* a number of memory locations, see Figure 9.2
* From a List
* From an InlineArray

A number and a pointer

Description automatically generated with medium confidence

Figure 9.2 – A Span with length 6, containing the integers 1 to 6, and starting in the location pointed to by the UnsafePointer.

The following code shows how to make a Span (see span.mojo):

from memory import Span #A

fn main():

    base\_data = List[Byte](1, 2, 3, 4, 5, 6) #B

    span1 = Span(base\_data) #C

print(span1[2])  # => 3 #C2

    span2 = Span(base\_data)[3:5] #D

  print(**len**(span2))  # => 2

    for i in span2:

        print(i[], end=" - ")  # => 4 - 5 -

    print()

    lst = base\_data.copy() #E

    lst.extend(span2) #F

    for i in lst:

        print(i[], end=" - ")  # => 1 - 2 - 3 - 4 - 5 - 4 - 5 -

    print()

    var str = String()

    str.**write\_bytes**(span2) #G

    span4 = str**.as\_bytes() #H**

    for cp in str.codepoints():

        print(Int(cp), end=" - ")  # => 4 - 5 -

    print()

#A Import Span from module memory

#B Create a base list of bytes

#C Make a Span covering the entire base list

#C2 Read an item from the span by index

#D Create a span slice from the base list

#E Copy the base list

#F Extend the copied list with a Span

#G Write the bytes of Span span2 to String str

#H as\_bytes() makes a Span span4 with the bytes of String str

The Span slice span2 creates a view on locations 3 and 4, containing items 4 and 5. The length len() function gives us the number of memory locations the Span views. We can read an item from a span by index: span[ix], because Span implements the \_\_getitem\_\_ method. Based on this, it has a swap\_elements method to swap items by index. However, you can’t change the underlying data of a Span, it’s only a view (\_\_setitem\_\_ is not implemented).

A List can also be extended with a Span: lst gets extended with span2. The code also demonstrates the as\_bytes() and write\_bytes() methods of String: the first method makes a Span of Byte with the contents of the string, the second one writes the bytes of a Span into a string.

Some other examples of using Span include:

In § 7.3.3 you saw an example of the Writable trait, which uses the Writer trait in its write\_to method. The Writer trait on the other hand has a write\_bytes method, which writes a Span[Byte] to the current Writer:

write\_bytes(mut self: \_Self, **bytes: Span[SIMD[uint8, 1], origin]**)

In the example code in chapter 7, we used this code:

fn write\_to[W: Writer](self, mut writer: W) -> None:

writer.write("Person(", self.name, ", ", self.age, ")")

But we can also write it like this, using a Span, produced by the String.as\_bytes method:

    fn write\_to[W: Writer](self, mut writer: W) -> None:

        writer.write\_bytes(String(self).as\_bytes())

In § 8.3.4 when discussing the sort method, we saw that all versions of sort take a Span as argument:

sort[…, **type: CollectionElement**, … //, **cmp\_fn: fn(type, type) capturing -> Bool**, \*, stable: Bool = False](**span: Span[type, …])**

This means the sort method takes a view on the collection or collection-slice it is going to sort.

* 1. Using UnsafePointer in collections
* A String contains a pointer that requires special constructor and destructor behavior to allocate and free memory: unsafe\_ptr
* List pointer, see lst.data in § 8.3.1
* Dict: get\_ptr
* /NDBUffer/Tensor unsafe\_ptr
  + - Example of gray\_scale\_image RGB
    - Explanations of the Model code of chapter 2 !!

Linked list, Node ??

* The PythonObject type defines an [unsafe\_get\_as\_pointer()](https://docs.modular.com/mojo/stdlib/python/object/PythonObject" \l "unsafe_get_as_pointer) method to construct an UnsafePointer from a Python address.
* Using the UnsafePointer type

var arr = InlineArray[Int, 3](1, 2, 3)  
var ptr = arr.unsafe\_ptr()  
print(ptr[0]) *# Prints 1*

**Returns:**

An UnsafePointer to the underlying array storage. The pointer's mutability matches that of the array reference.

* Create with Unsafe\_ptr
* Sorting with pointers, see ch12.txt in removed
  1. The lifecycle of a struct containing an UnsafePointer

Lifecycle! Struct with Examples of init, del, copy, move; see dtypepointer2.mojo, struct matrix

@value

struct ResourceOwner(AnyType):

var ptr: UnsafePointer[Int]

fn \_\_init\_\_(out self, size: Int):

self.ptr = UnsafePointer[Int].alloc(size)

fn \_\_del\_\_(owned self):

# Clean up owned resources

self.ptr.free()

* 1. Using the Pointer type

safe , non-nullable

* 1. Using the OwnedPointer type

safe, no free

* 1. Using the ArcPointer type

9.11 Lifetime, origins and references

**Returning a reference**

A function can also return a mutable or immutable reference using a ref return value. For details, see [Lifetimes, origins, and references](https://docs.modular.com/mojo/manual/values/lifetimes).

* As\_ref

Table with pointer properties (like in Manual) ??

Summary

* UnsafePointer is a manually managed pointer type. Allocate memory for it with the static alloc(n) method, where n is the number of memory locations.
* Initialize its memory with ptr.init\_pointee\_copy(value) or ptr.init\_pointee\_move(value^).
* Alternatively, without using alloc(), create a pointer to an existing value with UnsafePointer[T].address\_of(value), where T is the type of value.
* To free the memory: do ptr.destroy\_pointee() and then ptr.free().
* When working with SIMD values and data is a pointer to them, read the values with data.load(offset) and write values with data.store(offset, value).