Unsafe pointers

The `UnsafePointer` type creates an indirect reference to a location in memory. You can use an `UnsafePointer` to dynamically allocate and free memory, or to point to memory allocated by some other piece of code. You can use these pointers to write code that interacts with low-level interfaces, to interface with other programming languages, or to build certain kinds of data structures. But as the name suggests, they're inherently *unsafe*. For example, when using unsafe pointers, you're responsible for ensuring that memory gets allocated and freed correctly.

Note

In addition to unsafe pointers, Mojo supports a safe `Reference` type. See `UnsafePointer` and `Reference` for a brief comparison of the types.

What is a pointer?

An `UnsafePointer` is a type that holds an address to memory. You can store and retrieve values in that memory. The `UnsafePointer` type is *generic*—it can point to any type of value, and the value type is specified as a parameter. The value pointed to by a pointer is sometimes called a *pointee*.

```
from memory.unsafe_pointer import UnsafePointer

# Allocate memory to hold a value
var ptr = UnsafePointer[Int].alloc(1)
# Initialize the allocated memory
ptr.init_pointee_copy(100)
```

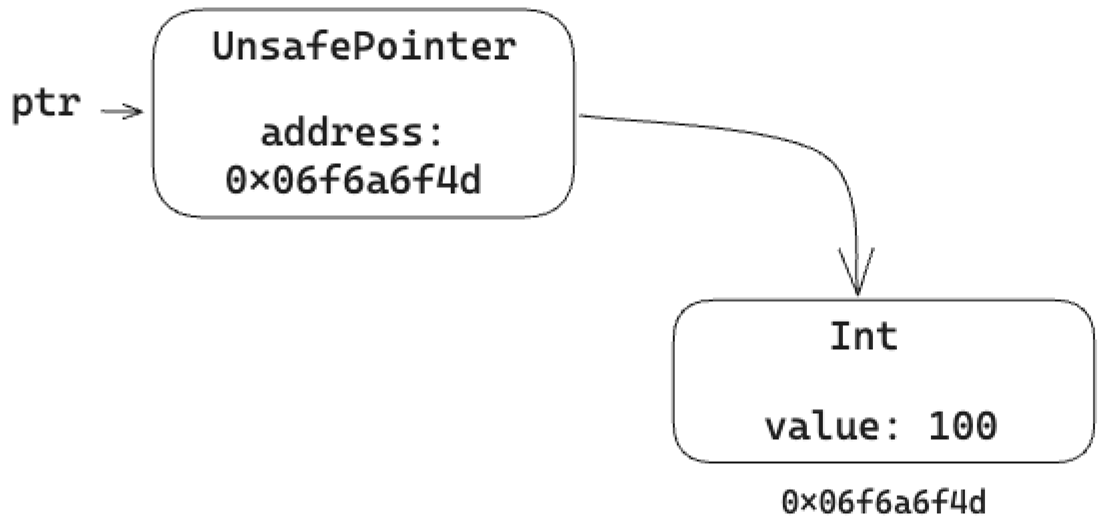

Figure 1. Pointer and pointee

Accessing the memory—to retrieve or update a value—is called *dereferencing* the pointer. You can dereference a pointer by following the variable name with an empty pair of square brackets:

```
# Update an initialized value
ptr[] += 10
# Access an initialized value
print(ptr[])
```

```
110
```

You can also allocate memory to hold multiple values to build array-like structures. For details, see Storing multiple values.

Lifecycle of a pointer

At any given time, a pointer can be in one of several states:

- Uninitialized. Just like any variable, a variable of type `UnsafePointer` can be declared but uninitialized.

  ```
  var ptr: UnsafePointer[Int]
  ```

- Null. A null pointer has an address of 0, indicating an invalid pointer.

  ```
  ptr = UnsafePointer[Int]()
  ```

- Pointing to allocated, uninitialized memory. The `alloc()` static method returns a pointer to a newly-allocated block of memory with space for the specified number of elements of the pointee's type.

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

  Trying to dereference a pointer to uninitialized memory results in undefined behavior.

- Pointing to initialized memory. You can initialize an allocated, uninitialized pointer by moving or copying an existing value into the memory. Or you can use the `address_of()` static method to get a pointer to an existing value.

  ```
  ptr.init_pointee_copy(value)
  # or
  ptr.init_pointee_move(value^)
  # or
  ptr = UnsafePointer[Int].address_of(value)
  ```

  Once the value is initialized, you can read or mutate it using the dereference syntax:

  ```
  oldValue = ptr[]
  ptr[] = newValue
  ```

- Dangling. When you free the pointer's allocated memory, you're left with a *dangling pointer*. The address still points to its previous location, but the memory is no longer allocated to this pointer. Trying to dereference the pointer, or calling any method that would access the memory location results in undefined behavior.

  ```
  ptr.free()
  ```

The following diagram shows the lifecycle of an `UnsafePointer`:

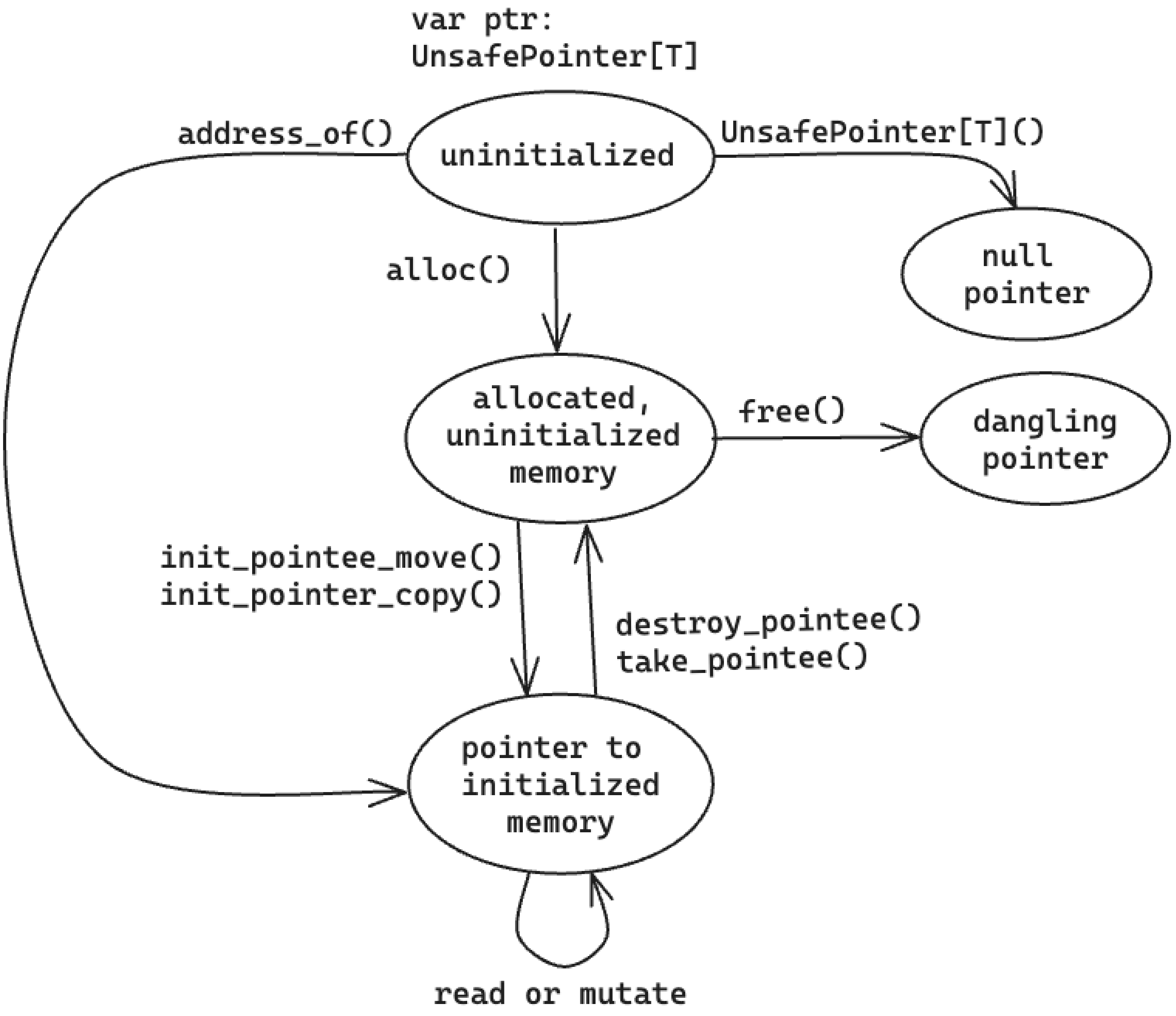

Figure 2. Lifecycle of an `UnsafePointer`

Allocating memory

Use the static `alloc()` method to allocate memory. The method returns a new pointer pointing to the requested memory. You can allocate space for one or more values of the pointee's type.

```
ptr = UnsafePointer[Int].alloc(10) # Allocate space for 10 Int values
```

The allocated space is *uninitialized*—like a variable that's been declared but not initialized.

Initializing the pointee

To initialize allocated memory, `UnsafePointer` provides the `init_pointee_copy()` and `init_pointee_move()` methods. For example:

```
ptr.init_pointee_copy(my_value)
```

To move a value into the pointer's memory location, use `init_pointee_move()`:

```
str_ptr.init_pointee_move(my_string^)
```

Note that to move the value, you usually need to add the transfer sigil (`^`), unless the value is a trivial type (like `Int`) or a newly-constructed, "owned" value:

```
str_ptr.init_pointee_move(str("Owned string"))
```

Alternately, you can get a pointer to an existing value using the static `address_of()` method. This is useful for getting a pointer to a value on the stack, for example.

```
var counter: Int = 5
ptr = UnsafePointer[Int].address_of(counter)
```

Note that when calling `address_of()`, you don't need to allocate memory ahead of time, since you're pointing to an existing value.

Dereferencing pointers

Use the `[]` dereference operator to access the value stored at a pointer (the "pointee").

```
# Read from pointee
print(ptr[])
# mutate pointee
ptr[] = 0
```

```
5
```

If you've allocated space for multiple values, you can use subscript syntax to access the values, as if they were an array, like `ptr[3]`. The empty subscript `[]` has the same meaning as `[0]`.

Caution

The dereference operator assumes that the memory being dereferenced is initialized. Dereferencing uninitialized memory results in undefined behavior.

You cannot safely use the dereference operator on uninitialized memory, even to *initialize* a pointee. This is because assigning to a dereferenced pointer calls lifecycle methods on the existing pointee (such as the destructor, move constructor or copy constructor).

```
str_ptr = UnsafePointer[String].alloc(1)
# str_ptr[] = "Testing" # Undefined behavior!
str_ptr.init_pointee_move("Testing")
str_ptr[] += " pointers" # Works now
```

Destroying or removing values

The `take_pointee()` method moves the pointee from the memory location pointed to by `ptr`. This is a consuming move—it invokes `__moveinit__()` on the destination value. It leaves the memory location uninitialized.

The `destroy_pointee()` method calls the destructor on the pointee, and leaves the memory location pointed to by `ptr` uninitialized.

Both `take_pointee()` and `destroy_pointee()` require that the pointer is non-null, and the memory location contains a valid, initialized value of the pointee's type; otherwise the function results in undefined behavior.

The `move_pointee_into(self, dst)` method moves the pointee from one pointer location to another. Both pointers must be non-null. The source location must contain a valid, initialized value of the pointee's type, and is left uninitialized after the call. The destination location is assumed to be uninitialized—if it contains a valid value, that value's destructor is not run. The value from the source location is moved to the destination location as a consuming move. This function also has undefined behavior if any of its prerequisites is not met.

Freeing memory

Calling `free()` on a pointer frees the memory allocated by the pointer. It doesn't call the destructors on any values stored in the memory—you need to do that explicitly (for example, using `destroy_pointee()` or one of the other functions described in Destroying or removing values).

Disposing of a pointer without freeing the associated memory can result in a memory leak—where your program keeps taking more and more memory, because not all allocated memory is being freed.

On the other hand, if you have multiple copies of a pointer accessing the same memory, you need to make sure you only call `free()` on one of them. Freeing the same memory twice is also an error.

After freeing a pointer's memory, you're left with a dangling pointer—its address still points to the freed memory. Any attempt to access the memory, like dereferencing the pointer results in undefined behavior.

Storing multiple values

As mentioned in Allocating memory, you can use an `UnsafePointer` to allocate memory for multiple values. The memory is allocated as a single, contiguous block. Pointers support arithmetic: adding an integer to a pointer returns a new pointer offset by the specified number of values from the original pointer:

```
third_ptr = first_ptr + 2
```

Pointers also support subtraction, as well as in-place addition and subtraction:

```
# Advance the pointer one element:
ptr += 1
```

Figure 3. Pointer arithmetic

For example, the following example allocates memory to store 6 `Float64` values, and initializes them all to zero.

```
float_ptr = UnsafePointer[Float64].alloc(6)
for offset in range(6):
    (float_ptr+offset).init_pointee_copy(0.0)
```

Once the values are initialized, you can access them using subscript syntax:

```
float_ptr[2] = 3.0
for offset in range(6):
    print(float_ptr[offset], end=", ")
```

```
0.0, 0.0, 3.0, 0.0, 0.0, 0.0,
```

Working with foreign pointers

When exchanging data with other programming languages, you may need to construct an `UnsafePointer` from an a foreign pointer. Mojo restricts creating `UnsafePointer` instances from arbitrary addresses, to avoid users accidentally creating pointers that *alias* each other (that is, two pointers that refer to the same location). However, there are specific methods you can use to get an `UnsafePointer` from a Python or C/C++ pointer.

When dealing with memory allocated elsewhere, you need to be aware of who's responsible for freeing the memory. Freeing memory allocated elsewhere can result in undefined behavior.

You also need to be aware of the format of the data stored in memory, including data types and byte order. For more information, see Converting data: bitcasting and byte order.

Creating a Mojo pointer from a Python pointer

The `PythonObject` type defines an `unsafe_get_as_pointer()` method to construct an `UnsafePointer` from a Python address.

For example, the following code creates a NumPy array and then accesses the data using a Mojo pointer:

```
from python import Python
from memory.unsafe_pointer import UnsafePointer

def share_array():
    np = Python.import_module("numpy")
    arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
    ptr = arr.__array_interface__["data"][0].unsafe_get_as_pointer[DType.int64]()
    for i in range(9):
        print(ptr[i], end=", ")

share_array()
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9,
```

NumPy arrays implement the array interface protocol, which defines the `__array_interface__` object used in the example, where `__array_interface__["data"][0]` is a Python integer holding the address of the underlying data. The `unsafe_get_as_pointer()` method constructs an `UnsafePointer` to this address.

Working with C/C++ pointers

If you call a C/C++ function that returns a pointer using the `external_call` function, you can specify the return type as an `UnsafePointer`, and Mojo will handle the type conversion for you.

```
from sys.ffi import external_call

def get_foreign_pointer() -> UnsafePointer[Int]:
    ptr = external_call[
        "my_c_function",   # external function name
        UnsafePointer[Int] # return type
    ]()
    return ptr
```

Converting data: bitcasting and byte order

Bitcasting a pointer returns a new pointer that has the same memory location, but a new data type. This can be useful if you need to access different types of data from a single area of memory. This can happen when you're reading binary files, like image files, or receiving data over the network.

The following sample processes a format that consists of chunks of data, where each chunk contains a variable number of 32-bit integers. Each chunk begins with an 8-bit integer that identifies the number of values in the chunk.

```
def read_chunks(owned ptr: UnsafePointer[UInt8]) -> List[List[UInt32]]:
    chunks = List[List[UInt32]]()
    # A chunk size of 0 indicates the end of the data
    chunk_size = int(ptr[])
    while (chunk_size > 0):
        # Skip the 1 byte chunk_size and get a pointer to the first
        # UInt32 in the chunk
        ui32_ptr = (ptr + 1).bitcast[UInt32]()
        chunk = List[UInt32](capacity=chunk_size)
        for i in range(chunk_size):
            chunk.append(ui32_ptr[i])
        chunks.append(chunk)
        # Move our pointer to the next byte after the current chunk
        ptr += (1 + 4 * chunk_size)
        # Read the size of the next chunk
        chunk_size = int(ptr[])
    return chunks
```

When dealing with data read in from a file or from the network, you may also need to deal with byte order. Most systems use little-endian byte order (also called least-signficicant byte, or LSB) where the least-significant byte in a multibyte value comes first. For example, the number 1001 can be represented in hexadecimal as 0x03E9, where E9 is the least-significant byte. Represented as a 16-bit little-endian integer, the two bytes are ordered E9 03. As a 32-bit integer, it would be represented as E9 03 00 00.

Big-endian or most-significant byte (MSB) ordering is the opposite: in the 32-bit case, 00 00 03 E9. MSB ordering is frequently used in file formats and when transmitting data over the network. You can use the `byte_swap()` function to swap the byte order of a SIMD value from big-endian to little-endian or the reverse. For example, if the method above was reading big-endian data, you'd just need to change a single line:

```
chunk.append(byte_swap(ui32_ptr[i]))
```

Working with SIMD vectors

The `UnsafePointer` type includes `load()` and `store()` methods for performing aligned loads and stores of scalar values. It also has methods supporting strided load/store and gather/scatter.

Strided load loads values from memory into a SIMD vector using an offset (the "stride") between successive memory addresses. This can be useful for extracting rows or columns from tabular data, or for extracting individual values from structured data. For example, consider the data for an RGB image, where each pixel is made up of three 8-bit values, for red, green, and blue. If you want to access just the red values, you can use a strided load or store.

Figure 4. Strided load

The following function uses the `strided_load()` and `strided_store()` methods to invert the red pixel values in an image, 8 values at a time. (Note that this function only handles images where the number of pixels is evenly divisible by eight.)

```
def invert_red_channel(ptr: UnsafePointer[UInt8], pixel_count: Int):
    # number of values loaded or stored at a time
    alias simd_width = 8
    # bytes per pixel, which is also the stride size
    bpp = 3
    for i in range(0, pixel_count * bpp, simd_width * bpp):
        red_values = ptr.offset(i).strided_load[width=simd_width](bpp)
        # Invert values and store them in their original locations
        ptr.offset(i).strided_store[width=simd_width](~red_values, bpp)
```

The `gather()` and `scatter()` methods let you load or store a set of values that are stored in arbitrary locations. You do this by passing in a SIMD vector of *offsets* to the current pointer. For example, when using `gather()`, the *n*th value in the vector is loaded from (pointer address) + *offset[n]*.

Safety

Unsafe pointers are unsafe for several reasons:

- Memory management is up to the user. You need to manually allocate and free memory, and be aware of when other APIs are allocating or freeing memory for you.
- `UnsafePointer` values are *nullable*—that is, the pointer is not guaranteed to point to anything. And even when a pointer points to allocated memory, that memory may not be *initialized*.
- Mojo doesn't track lifetimes for the data pointed to by an `UnsafePointer`. When you use an `UnsafePointer`, managing memory and knowing when to destroy objects is your responsibility.

`UnsafePointer` and `Reference`

The `Reference` type is essentially a safe pointer type. Like a pointer, you can derferences a `Reference` using the dereference operator, `[]`. However, the `Reference` type has several differences from `UnsafePointer` which make it safer:

- A `Reference` is *non-nullable*. A reference always points to something.
- You can't allocate or free memory using a `Reference`—only point to an existing value.
- A `Reference` only refers to a single value. You can't do pointer arithmetic with a `Reference`.
- A `Reference` has an associated *lifetime*, which connects it back to an original, owned value. The lifetime ensures that the value won't be destroyed while the reference exists.

The `Reference` type shouldn't be confused with the immutable and mutable references used with the `borrowed` and `inout` argument conventions. Those references do not require explicit dereferencing, unlike a `Reference` or `UnsafePointer`.