

Life of a value

The life of a value in Mojo begins when a variable is initialized and continues up until the value is last used, at which point Mojo destroys it. This page describes how every value in Mojo is created, copied, and moved. (The next page describes [how values are destroyed](#).)

All data types in Mojo—including basic types in the standard library such as [Bool](#), [Int](#), and [String](#), up to complex types such as [SIMD](#) and [object](#)—are defined as a [struct](#). This means the creation and destruction of any piece of data follows the same lifecycle rules, and you can define your own data types that work exactly the same way.

Mojo structs don't get any default lifecycle methods, such as a constructor, copy constructor, or move constructor. That means you can create a struct without a constructor, but then you can't instantiate it, and it would be useful only as a sort of namespace for static methods. For example:

```
struct NoInstances:
    var state: Int

    @staticmethod
    fn print_hello():
        print("Hello world!")
```

Without a constructor, this cannot be instantiated, so it has no lifecycle. The `state` field is also useless because it cannot be initialized (Mojo structs do not support default field values—you must initialize them in a constructor).

So the only thing you can do is call the static method:

```
NoInstances.print_hello()
```

```
Hello world!
```

Constructor

To create an instance of a Mojo type, it needs the `__init__()` constructor method. The main responsibility of the constructor is to initialize all fields. For example:

```
struct MyPet:
  var name: String
  var age: Int

  fn __init__(inout self, name: String, age: Int):
    self.name = name
    self.age = age
```

Now we can create an instance:

```
var mine = MyPet("Loki", 4)
```

An instance of `MyPet` can also be [borrowed](#) and destroyed, but it currently can't be copied or moved.

We believe this is a good default starting point, because there are no built-in lifecycle events and no surprise behaviors. You—the type author—must explicitly decide whether and how the type can be copied or moved, by implementing the copy and move constructors.

Note

Mojo does not require a destructor to destroy an object. As long as all fields in the struct are destructible (every type in the standard library is destructible, except for [pointers](#)), then Mojo knows how to destroy the type when its lifetime ends. We'll discuss that more in [Death of a value](#).

Overloading the constructor

Like any other function/method, you can [overload](#) the `__init__()` constructor to initialize the object with different arguments. For example, you might want a default constructor that sets some default values and takes no arguments, and then additional constructors that accept more arguments.

Just be aware that, in order to modify any fields, each constructor must declare the `self` argument with the [inout convention](#). If you want to call one constructor from another, you simply call upon that constructor as you would externally (you don't need to pass `self`).

For example, here's how you can delegate work from an overloaded constructor:

```
struct MyPet:
  var name: String
  var age: Int

  fn __init__(inout self):
    self.name = ""
    self.age = 0
```

```
fn __init__(inout self, name: String):
    self = MyPet()
    self.name = name
```

Field initialization

Notice in the previous example that, by the end of each constructor, all fields must be initialized. That's the only requirement in the constructor.

In fact, the `__init__()` constructor is smart enough to treat the `self` object as fully initialized even before the constructor is finished, as long as all fields are initialized. For example, this constructor can pass around `self` as soon as all fields are initialized:

```
fn use(arg: MyPet):
    pass

struct MyPet:
    var name: String
    var age: Int

fn __init__(inout self, name: String, age: Int, cond: Bool):
    self.name = name
    if cond:
        self.age = age
        use(self) # Safe to use immediately!

    self.age = age
    use(self) # Safe to use immediately!
```

Constructors and implicit conversion

Mojo supports implicit conversion from one type to another. Implicit conversion can happen when one of the following occurs:

- You assign a value of one type to a variable with a different type.
- You pass a value of one type to a function that requires a different type.

In both cases, implicit conversion is supported when the target type defines a constructor that takes a single required, non-keyword argument of the source type. For example:

```
var a = Source()
var b: Target = a
```

Mojo implicitly converts the `Source` value in `a` to a `Target` value if `Target` defines a matching constructor like this:

```
struct Target:
    fn __init__(inout self, s: Source): ...
```

With implicit conversion, the assignment above is essentially identical to:

```
var b = Target(a)
```

The constructor used for implicit conversion can take optional arguments, so the following constructor would also support implicit conversion from `Source` to `Target`:

```
struct Target:
    fn __init__(inout self, s: Source, reverse: Bool = False): ...
```

Implicit conversion also occurs if the type doesn't declare its own constructor, but instead uses the [@value decorator](#), and the type has only one field. That's because Mojo automatically creates a member-wise constructor for each field, and when there is only one field, that synthesized constructor works exactly like a conversion constructor. For example, this type also can convert a `Source` value to a `Target` value:

```
@value
struct Target:
    var s: Source
```

Implicit conversion can fail if Mojo can't unambiguously match the conversion to a constructor. For example, if the target type has two overloaded constructors that take different types, and each of those types supports an implicit conversion from the source type, the compiler has two equally-valid paths to convert the values:

```
struct A:
    fn __init__(inout self, s: Source): ...

struct B:
    fn __init__(inout self, s: Source): ...

struct Target:
    fn __init__(inout self, a: A): ...
    fn __init__(inout self, b: B): ...

# Fails
var t = Target(Source())
```

In this case, removing either one of the target type's constructors will fix the problem.

If you want to define a single-argument constructor, but you **don't** want the types to implicitly convert, you can define the constructor with a [keyword-only argument](#):

```
struct Target:
    # does not support implicit conversion
    fn __init__(inout self, *, source: Source): ...

# the constructor must be called with a keyword
var t = Target(source=a)
```

Note

In the future we intend to provide a more explicit method of declaring whether a constructor should support implicit conversion.

Copy constructor

When Mojo encounters an assignment operator (=), it tries to make a copy of the right-side value by calling upon that type's copy constructor: the `__copyinit__()` method. Thus, it's the responsibility of the type author to implement `__copyinit__()` so it returns a copy of the value.

For example, the `MyPet` type above does not have a copy constructor, so this code fails to compile:

```
var mine = MyPet("Loki", 4)
var yours = mine # This requires a copy, but MyPet has no copy constructor
```

To make it work, we need to add the copy constructor, like this:

```
struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, name: String, age: Int):
        self.name = name
        self.age = age

    fn __copyinit__(inout self, existing: Self):
        self.name = existing.name
        self.age = existing.age
```

Note

`Self` (capital "S") is an alias for the current type name (`MyPet`, in this example). Using this alias is a best practice to avoid any mistakes when referring to the current struct name.

Also, notice that the `existing` argument in `__copyinit__()` is immutable because the default argument convention in an `fn` function is borrowed—this is a good thing because this function should not modify the contents of the value being copied.

Now this code works to make a copy:

```
var mine = MyPet("Loki", 4)
var yours = mine
```

What makes Mojo's copy behavior different, compared to other languages, is that `__copyinit__()` is designed to perform a deep copy of all fields in the type (as per value semantics). That is, it copies heap-allocated values, rather than just copying the pointer.

However, the Mojo compiler doesn't enforce this, so it's the type author's responsibility to implement `__copyinit__()` with value semantics. For example, here's a new `HeapArray` type that performs a deep copy in the copy constructor:

```
struct HeapArray:
  var data: UnsafePointer[Int]
  var size: Int
  var cap: Int

  fn __init__(inout self, size: Int, val: Int):
    self.size = size
    self.cap = size * 2
    self.data = UnsafePointer[Int].alloc(self.cap)
    for i in range(self.size):
      (self.data + i).init_pointee_copy(val)

  fn __copyinit__(inout self, existing: Self):
    # Deep-copy the existing value
    self.size = existing.size
    self.cap = existing.cap
    self.data = UnsafePointer[Int].alloc(self.cap)
    for i in range(self.size):
      (self.data + i).init_pointee_copy(existing.data[i])
    # The lifetime of `existing` continues unchanged

  fn __del__(owned self):
    # We must free the heap-allocated data, but
    # Mojo knows how to destroy the other fields
```

```

    for i in range(self.size):
        (self.data + i).destroy_pointee()
    self.data.free()

fn append(inout self, val: Int):
    # Update the array for demo purposes
    if self.size < self.cap:
        (self.data + self.size).init_pointee_copy(val)
        self.size += 1
    else:
        print("Out of bounds")

fn dump(self):
    # Print the array contents for demo purposes
    print("[", end="")
    for i in range(self.size):
        if i > 0:
            print(", ", end="")
        print(self.data[i], end="")
    print("]")

```

Notice that `__copyinit__()` does not copy the `UnsafePointer` value (doing so would make the copied value refer to the same `data` memory address as the original value, which is a shallow copy). Instead, we initialize a new `UnsafePointer` to allocate a new block of memory, and then copy over all the heap-allocated values (this is a deep copy).

Thus, when we copy an instance of `HeapArray`, each copy has its own value on the heap, so changes to one value do not affect the other, as shown here:

```

fn copies():
    var a = HeapArray(2, 1)
    var b = a    # Calls the copy constructor
    a.dump()     # Prints [1, 1]
    b.dump()     # Prints [1, 1]

    b.append(2)  # Changes the copied data
    b.dump()     # Prints [1, 1, 2]
    a.dump()     # Prints [1, 1] (the original did not change)

```

Note

In `HeapArray`, we must use the `__del__()` destructor to free the heap-allocated data when the `HeapArray` lifetime ends, but Mojo automatically destroys all other fields when their respective lifetimes end. We'll discuss this destructor more in [Death of a value](#).

If your type doesn't use any pointers for heap-allocated data, then writing the constructor and copy constructor is all boilerplate code that you shouldn't have to write. For most structs that don't manage memory explicitly, you can just add the [@value decorator](#) to your struct definition and Mojo will synthesize the `__init__()`, `__copyinit__()`, and `__moveinit__()` methods.

Note

Mojo also calls upon the copy constructor when a value is passed to a function that takes the argument as owned *and* when the lifetime of the given value does *not* end at that point. If the lifetime of the value does end there (usually indicated with the transfer sigil `^`), then Mojo instead invokes the move constructor.

Move constructor

Although copying values provides predictable behavior that matches Mojo's [value semantics](#), copying some data types can be a significant hit on performance. If you're familiar with reference semantics, then the solution here might seem clear: instead of making a copy when passing a value, share the value as a reference. And if the original variable is no longer needed, nullify the original to avoid any double-free or use-after-free errors. That's generally known as a move operation: the memory block holding the data remains the same (the memory does not actually move), but the pointer to that memory moves to a new variable.

To support moving a value, implement the `__moveinit__()` method. The `__moveinit__()` method performs a consuming move: it [transfers ownership](#) of a value from one variable to another when the original variable's lifetime ends (also called a "destructive move").

Note

A move constructor is **not required** to transfer ownership of a value. Unlike in Rust, transferring ownership is not always a move operation; the move constructors are only part of the implementation for how Mojo transfers ownership of a value. You can learn more in the section about [ownership transfer](#).

When a move occurs, Mojo immediately invalidates the original variable, preventing any access to it and disabling its destructor. Invalidating the original variable is important to avoid memory errors on heap-allocated data, such as use-after-free and double-free errors.

Here's how to add the move constructor to the `HeapArray` example:

```
struct HeapArray:
  var data: UnsafePointer[Int]
  var size: Int
  var cap: Int
```



```

fn __init__(inout self, size: Int, val: Int):
    self.size = size
    self.cap = size * 2
    self.data = UnsafePointer[Int].alloc(self.size)
    for i in range(self.size):
        (self.data + i).init_pointee_copy(val)

fn __copyinit__(inout self, existing: Self):
    # Deep-copy the existing value
    self.size = existing.size
    self.cap = existing.cap
    self.data = UnsafePointer[Int].alloc(self.cap)
    for i in range(self.size):
        (self.data + i).init_pointee_copy(existing.data[i])
    # The lifetime of `existing` continues unchanged

fn __moveinit__(inout self, owned existing: Self):
    print("move")
    # Shallow copy the existing value
    self.size = existing.size
    self.cap = existing.cap
    self.data = existing.data
    # Then the lifetime of `existing` ends here, but
    # Mojo does NOT call its destructor

fn __del__(owned self):
    # We must free the heap-allocated data, but
    # Mojo knows how to destroy the other fields
    for i in range(self.size):
        (self.data + i).destroy_pointee()
    self.data.free()

fn append(inout self, val: Int):
    # Update the array for demo purposes
    if self.size < self.cap:
        (self.data + self.size).init_pointee_copy(val)
        self.size += 1
    else:
        print("Out of bounds")

fn dump(self):
    # Print the array contents for demo purposes
    print("[", end="")
    for i in range(self.size):
        if i > 0:
            print(", ", end="")
        print(self.data[i], end="")
    print("]")

```

The critical feature of `__moveinit__()` is that it takes the incoming value as `owned`, meaning this method gets unique ownership of the value. Moreover, because this is a dunder method that Mojo calls only when performing a move (during ownership transfer), the `existing` argument is guaranteed to be a mutable reference to the original value, *not a copy* (unlike other methods that may declare an argument as `owned`, but might receive the value as a copy if the method is called without the [^ transfer sigil](#)). That is, Mojo calls this move constructor *only* when the original variable's lifetime actually ends at the point of transfer.

Here's an example showing how to invoke the move constructor for `HeapArray`:

```
fn moves():
    var a = HeapArray(3, 1)

    a.dump()    # Prints [1, 1, 1]

    var b = a^ # Prints "move"; the lifetime of `a` ends here

    b.dump()    # Prints [1, 1, 1]
    #a.dump()   # ERROR: use of uninitialized value 'a'
```

Notice that `__moveinit__()` performs a shallow copy of the existing field values (it copies the pointer, instead of allocating new memory on the heap), which is what makes it useful for types with heap-allocated values that are expensive to copy.

To go further and ensure your type can never be copied, you can make it "move-only" by implementing `__moveinit__()` and *excluding* `__copyinit__()`. A move-only type can be passed to other variables and passed into functions with any argument convention (`borrowed`, `inout`, and `owned`)—the only catch is that you must use the `^` transfer sigil to end the lifetime of a move-only type when assigning it to a new variable or when passing it as an `owned` argument.

Note

For types without heap-allocated fields, you get no real benefit from the move constructor. Making copies of simple data types on the stack, like integers, floats, and booleans, is very cheap. Yet, if you allow your type to be copied, then there's generally no reason to disallow moves, so you can synthesize both constructors by adding the [`@value` decorator](#).

Simple value types

Because copy and move constructors are opt-in, Mojo provides great control for exotic use cases (such as for atomic values that should never be copied or moved), but most structs are simple aggregations of other types that should be easily copied and moved, and we don't want to write a lot of boilerplate constructors for those simple value types.

To solve this, Mojo provides the [@value decorator](#), which synthesizes the boilerplate code for the `__init__()`, `__copyinit__()`, and `__moveinit__()` methods.

For example, consider a simple struct like this:

```
@value
struct MyPet:
    var name: String
    var age: Int
```

Mojo sees the `@value` decorator and notices that you don't have a member-wise initializer (a constructor with arguments for each field), a copy constructor, or a move constructor, so it synthesizes them for you. The result is as if you had actually written this:

```
struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, owned name: String, age: Int):
        self.name = name^
        self.age = age

    fn __copyinit__(inout self, existing: Self):
        self.name = existing.name
        self.age = existing.age

    fn __moveinit__(inout self, owned existing: Self):
        self.name = existing.name^
        self.age = existing.age
```

Mojo synthesizes each lifecycle method only when it doesn't exist, so you can use `@value` and still define your own versions to override the default behavior. For example, it is fairly common to use the default member-wise and move constructor, but create a custom copy constructor. Another common pattern is to use `@value` to create a member-wise constructor, and add overloads that take different sets of arguments. For example, if you want to create a `MyPet` struct without specifying an age, you could add an overloaded constructor.

```
@value
struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, owned name: String):
        self.name = name^
        self.age = 0
```

Note that this overloaded constructor **doesn't** prevent the `@value` decorator from synthesizing the member-wise constructor. To override this default constructor, you'd need to add a constructor with the same signature as the default member-wise constructor.

Something you can see in this code that we didn't mention yet is that the `__init__()` method takes all arguments as `owned`, because the constructor must take ownership to store each value. This is a useful micro-optimization and enables the use of move-only types. Trivial types like `Int` are also passed as `owned`, but because ownership doesn't mean anything for integers, we can elide that declaration and the transfer sigil (`^`) for simplicity. The transfer operator is also just a formality in this case, because, even if it's not used with `self.name = name^`, the Mojo compiler will notice that `name` is last used here and convert this assignment into a move, instead of a copy+delete.

Note

If your type contains any move-only fields, Mojo will not generate the copy constructor because it cannot copy those fields. Further, the `@value` decorator won't work at all if any of your members are neither copyable nor movable. For example, if you have something like `Atomic` in your struct, then it probably isn't a true value type, and you don't want the copy/move constructors anyway.

Also notice that the `MyPet` struct above doesn't include the `__del__()` destructor (the `@value` decorator does not synthesize this), because Mojo doesn't need it to destroy fields, as discussed in [Death of a value](#)

Trivial types

So far, we've talked about values that live in memory, which means they have an identity (an address) that can be passed around among functions (passed "by reference"). This is great for most types, and it's a safe default for large objects with expensive copy operations. However, it's inefficient for tiny things like a single integer or floating point number. We call these types "trivial" because they are just "bags of bits" that should be copied, moved, and destroyed without invoking any custom lifecycle methods.

Trivial types are the most common types that surround us, and from a language perspective, Mojo doesn't need special support for these written in a struct. Usually, these values are so tiny that they should be passed around in CPU registers, not indirectly through memory.

As such, Mojo provides a struct decorator to declare these types of values: `@register_passable("trivial")`. This decorator tells Mojo that the type should be copyable and movable but that it has no user-defined logic (no lifecycle methods) for doing this. It also tells Mojo to pass the value in CPU registers whenever possible, which has clear performance benefits.

You'll see this decorator on types like `Int` in the standard library:

```
@register_passable("trivial")
struct Int:
```

```
var value: __mlir_type.index

fn __init__(value: __mlir_type.index) -> Int:
    return Self {value: value}
...
```


We expect to use this decorator pervasively on Mojo standard library types, but it is safe to ignore for general application-level code.

For more information, see the [@register_passable documentation](#).

TODO

This decorator is due for reconsideration. Lack of custom copy/move/destroy logic and "passability in a register" are orthogonal concerns and should be split. This former logic should be subsumed into a more general `@value("trivial")` decorator, which is orthogonal from `@register_passable`.

Was this page helpful?  

 Edit this page