Lifetimes and references

Work in progress

Both lifetimes and references are a work in progress and subject to change in future releases.

In Mojo, *lifetime* has two meanings:

- In general terms, a value's lifetime refers to the span of time when the value is valid.
- It also refers to a specific type of parameter value used to help track the lifetimes of values and references to values. For clarity, we'll use `lifetime` in code font to refer to the type.

The Mojo compiler includes a lifetime checker, a compiler pass that analyzes dataflow through your program. It identifies when variables are valid and inserts destructor calls when a value's lifetime ends.

The Mojo compiler uses `lifetime` values to track the validity of references. Specifically, a `lifetime` value answers two questions:

- What logical storage location "owns" this value?
- Can the value be mutated using this reference?

For example, consider the following code:

```
fn print_str(s: String):
    print(s)

name = String("Joan")
print_str(name)
```

```
Joan
```

The line `name = String("Joan")` declares a variable with an identifier (`name`) and logical storage space for a `String` value. When you pass `name` into the `print_str()` function, the function gets an immutable reference to the value. So both `name` and `s` refer to the same logical storage space, and have associated `lifetime` values that lets the Mojo compiler reason about them.

Most of the time, `lifetime` values are handled automatically by the compiler. However, in some cases you'll need to interact with `lifetime` values directly:

- When working with references—specifically `ref` arguments and `ref` return values.
- When working with types like `Reference` or `Span` which are parameterized on the `lifetime` of the data they refer to.

This section covers `ref` arguments and `ref` return values, which let functions take arguments and provide return values as references with parametric lifetimes.

Working with lifetimes

Mojo's `lifetime` values are unlike most other values in the language, because they're primitive values, not Mojo structs. Specifying a parameter that takes a `lifetime` value, you can't just say, `l: Lifetime`, because there's no `Lifetime` type. Likewise, because these values are mostly created by the compiler, you can't just create your own `lifetime` value—you usually need to derive a `lifetime` from an existing value.

Lifetime types

Mojo supplies a struct and a set of aliases that you can use to specify `lifetime` types. As the names suggest, the `ImmutableLifetime` and `MutableLifetime` aliases represent immutable and mutable lifetimes, respectively:

```
struct ImmutableRef[lifetime: ImmutableLifetime]:
    pass
```

Or you can use the `AnyLifetime` struct to specify a lifetime with parametric mutability:

```
struct ParametricRef[
    is_mutable: Bool,
    //,
    lifetime: AnyLifetime[is_mutable].type
]:
    pass
```

Note that `AnyLifetime` *isn't a lifetime value*, it's a helper for specifying a `lifetime` **type**. Lifetime types carry the mutability of a reference as a boolean parameter value, indicating whether the lifetime is mutable, immutable, or even with mutability depending on a parameter specified by the enclosing API.

The `is_mutable` parameter here is an infer-only parameter. It's never specified directly by the user, but always inferred from context. The `lifetime` value is often inferred, as well. For example, the following code creates a `Reference` to an existing value, but doesn't need to specify a lifetime—the `lifetime` is inferred from the variable passed in to the reference.

```
from memory import Reference

def use_reference():
    a = 10
    r = Reference(a)
```

Lifetime values

Most `lifetime` values are created by the compiler. As a developer, there are a few ways to specify `lifetime` values:

- Static lifetimes. The `ImmutableStaticLifetime` and `MutableStaticLifetime` aliases are lifetimes that last for the duration of the program.
- The `__lifetime_of()` magic function, which returns the lifetime associated with the value (or values) passed in.
- Inferred lifetime. You can use inferred parameters to capture the lifetime of a value passed in to a function.

Static lifetimes

You can use the static lifetimes `ImmutableStaticLifetime` and `MutableStaticLifetime` when you have a value that should never be destroyed; or when there's no way to construct a meaningful `lifetime` for a value.

For an example of the first case, the `StringLiteral` method [`as_string_slice()`]() returns a [`StringSlice`]() pointing to the original string literal. String literals are static—they're allocated at compile time and never destroyed—so the slice is created with an immutable, static lifetime.

Converting an [`UnsafePointer`]() into a `Reference` is an example of the second case: the `UnsafePointer`'s data doesn't carry a `lifetime`—one reason that it's considered unsafe—but you need to specify a `lifetime` when creating a `Reference`. In this case, there's no way to construct a meaningful `lifetime` value, so the new `Reference` is constructed with a `MutableStaticLifetime`. Mojo won't destroy this value automatically. As with any value stored using a pointer, it's up to the user to explicitly [destroy the value]().

Derived lifetimes

Use the `__lifetime_of()` magic function to obtain a value's lifetime. This can be useful, for example, when creating a container type. Consider the `List` type. Subscripting into a list (`list[4]`) returns a reference to the item at the specified position. The signature of the `__getitem__()` method that's called to return the subscripted item looks like this:

```
fn __getitem__(ref [_]self, idx: Int) -> ref [__lifetime_of(self)] T:
```

The syntax may be unfamiliar— `ref` arguments and `ref` return values are described in the following sections. For now it's enough to know that the return value is a reference of type `T` (where `T` is the element type stored in the list), and the reference has the same lifetime as the list itself. This means that as long as you hold the reference, the underlying list won't be destroyed.

> **Note**
>
> Ideally the returned reference's `lifetime` would be linked to the individual list item, rather than the list itself. Mojo doesn't yet have a mechanism to express this relationship.

Inferred lifetimes

The other common way to access a lifetime value is to *infer* it from the the arguments passed to a function or method. For example, the `Span` type has an associated `lifetime`:

```
struct Span[
    is_mutable: Bool, //,
    T: CollectionElement,
    lifetime: AnyLifetime[is_mutable].type,
](CollectionElementNew):
    """A non owning view of contiguous data.
```

One of its constructors creates a `Span` from an existing `List`, and infers its `lifetime` value from the list:

```
    fn __init__(inout self, ref [lifetime]list: List[T, *_]):
        """Construct a Span from a List.

        Args:
            list: The list to which the span refers.
        """
        self._data = list.data
        self._len = len(list)
```

Working with references

You can use the `ref` keyword with arguments and return values to specify a reference with parametric mutability. That is, they can be either mutable or immutable.

These references shouldn't be confused with the `Reference` type, which is basically a safe pointer type. A `Reference` needs to be dereferenced, like a pointer, to access the underlying value. A `ref` argument, on the other hand, looks like a `borrowed` or `inout` argument inside the function. A `ref` return value looks like any other return value to the calling function, but it is a *reference* to an existing value, not a copy.

`ref` arguments

The `ref` argument convention lets you specify an argument of parametric mutability: that is, you don't need to know in advance whether the passed argument will be mutable or immutable. There are several reasons you might want to use a `ref` argument:

- You want to accept an argument with parametric mutability.
- You want to tie the lifetime of one argument to the lifetime of another argument.
- When you want an argument that is guaranteed to be passed in memory: this can be important and useful for generic arguments that need an identity, irrespective of whether the concrete type is register passable.

The syntax for a `ref` argument is:

```
ref [lifetime] argName: argType
```

The `lifetime` parameter passed inside the square brackets can be replaced with an underscore character (`_`) to indicate that the parameter is *unbound*. Think of it as a wildcard that will accept any lifetime:

```
def add_ref(ref [_] a: Int, b: Int) -> Int:
    return a+b
```

You can also name the lifetime explicitly. This is useful if you want to specify an `ImmutableLifetime` or `MutableLifetime`, or if you want to bind to the `is_mutable` parameter.

```
def take_str_ref[
      is_mutable: Bool, //,
      life: AnyLifetime[is_mutable].type
    ](ref [life] s: String):
    @parameter
    if is_mutable:
        print("Mutable: " + s)
    else:
        print("Immutable: " + s)

def pass_refs(s1: String, owned s2: String):
    take_str_ref(s1)
    take_str_ref(s2)

pass_refs("Hello", "Goodbye")
```

```
Immutable: Hello
Mutable: Goodbye
```

`ref` return values

Like `ref` arguments, `ref` return values allow a function to return a mutable or immutable reference to a value. Like a `borrowed` or `inout` argument, these references don't need to be dereferenced.

`ref` return values can be an efficient way to handle updating items in a collection. The standard way to do this is by implementing the `__getitem__()` and `__setitem__()` dunder methods. These are invoked to read from and write to a subscripted item in a collection:

```
value = list[a]
list[b] += 10
```

With a `ref` argument, `__getitem__()` can return a mutable reference that can be modified directly. This has pros and cons compared to using a `__setitem__()` method:

- The mutable reference is more efficient—a single update isn't broken up across two methods. However, the referenced value must be in memory.
- A `__getitem__()` / `__setitem__()` pair allows for arbitrary to be run when values are retrieved and set. For example, `__setitem__()` can validate or constrain input values.

For example, in the following example, `NameList` has a `get()` method that returns a reference:

```
struct NameList:
    var names: List[String]

    def __init__(inout self, *names: String):
        self.names = List[String]()
        for name in names:
            self.names.append(name[])

    def __getitem__(ref [_] self: Self, index: Int) ->
        ref [__lifetime_of(self)] String:
        if (index >=0 and index < len(self.names)):
            return self.names[index]
        else:
            raise Error("index out of bounds")

def use_name_list():
    list = NameList("Thor", "Athena", "Dana", "Vrinda")
    print(list[2])
    list[2] += "?"
```

```
    print(list[2])

use_name_list()
```

```
Dana
Dana?
```

Note that this update succeeds, even though `NameList` doesn't define a `__setitem__()` method:

```
list[2] += "?"
```

Also note that the code uses the return value directly each time, rather than assigning the return value to a variable, like this:

```
name = list[2]
```

Since a variable needs to own its value, `name` would end up with an owned *copy* of the value that `list[2]` returns. Mojo doesn't currently have syntax to express that you want to keep the original reference in `name`. This will be added in a future release.

In cases where you need to be able to assign the return value to a variable—for example, an iterator which will be used in a `for..in` loop—you might consider returning a `Reference` instead of a `ref` return value. For example, see the [iterator for the `List` type](). You can assign a `Reference` to a variable, but you need to use the dereference operator (`[]`) to access the underlying value.

```
nums = List(1, 2, 3)
for item in nums: # List iterator returns a Reference
    print(item[])
```

```
1
2
3
```

Parametric mutability of return values

Another advantage of `ref` return arguments is the ability to support parametric mutability. For example, recall the signature of the `__getitem__()` method above:

```
def __getitem__(ref [_] self: Self, index: Int) ->
    ref [__lifetime_of(self)] String:
```

Since the `lifetime` of the return value is tied to the lifetime of `self`, the returned reference will be mutable if the method was called using a mutable reference. The method still works if you have an immutable reference to the `NameList`, but it returns an immutable reference:

```
fn pass_immutable_list(list: NameList) raises:
    print(list[2])
    # list[2] += "?" # Error, this list is immutable

def use_name_list_again():
    list = NameList("Sophie", "Jack", "Diana")
    pass_immutable_list(list)

use_name_list_again()
```

```
Diana
```

Without parametric mutability, you'd need to write two versions of `__getitem__()`, one that accepts an immutable `self` and another that accepts a mutable `self`.