Structure

UnsafePointer

A pointer for accessing data of a specific type.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
@frozen
struct UnsafePointer<Pointee> where Pointee : ~Copyable
```

Mentioned in

Using Imported C Functions in Swift

Overview

You use instances of the `UnsafePointer` type to access data of a specific type in memory. The type of data that a pointer can access is the pointer's `Pointee` type. `UnsafePointer` provides no automated memory management or alignment guarantees. You are responsible for handling the life cycle of any memory you work with through unsafe pointers to avoid leaks or undefined behavior.

Memory that you manually manage can be either *untyped* or *bound* to a specific type. You use the `UnsafePointer` type to access and manage memory that has been bound to a specific type.

Understanding a Pointer's Memory State

The memory referenced by an `UnsafePointer` instance can be in one of several states. Many pointer operations must only be applied to pointers with memory in a specific state—you must keep track of the state of the memory you are working with and understand the changes to that

state that different operations perform. Memory can be untyped and uninitialized, bound to a type and uninitialized, or bound to a type and initialized to a value. Finally, memory that was allocated previously may have been deallocated, leaving existing pointers referencing unallocated memory.

Uninitialized Memory

Memory that has just been allocated through a typed pointer or has been deinitialized is in an *uninitialized* state. Uninitialized memory must be initialized before it can be accessed for reading.

Initialized Memory

Initialized memory has a value that can be read using a pointer's `pointee` property or through subscript notation. In the following example, `ptr` is a pointer to memory initialized with a value of 23:

```
let ptr: UnsafePointer<Int> = ...
// ptr.pointee == 23
// ptr[0] == 23
```

Accessing a Pointer's Memory as a Different Type

When you access memory through an `UnsafePointer` instance, the `Pointee` type must be consistent with the bound type of the memory. If you do need to access memory that is bound to one type as a different type, Swift's pointer types provide type-safe ways to temporarily or permanently change the bound type of the memory, or to load typed instances directly from raw memory.

An `UnsafePointer<UInt8>` instance allocated with eight bytes of memory, `uint8Pointer`, will be used for the examples below.

```
let uint8Pointer: UnsafePointer<UInt8> = fetchEightBytes()
```

When you only need to temporarily access a pointer's memory as a different type, use the `withMemoryRebound(to:capacity:)` method. For example, you can use this method to call an API that expects a pointer to a different type that is layout compatible with your pointer's `Pointee`.

The following code temporarily rebinds the memory that `uint8Pointer` references from `UInt8` to `Int8` to call the imported C `strlen` function.

```
// Imported from C
func strlen(_ __s: UnsafePointer<Int8>!) -> UInt

let length = uint8Pointer.withMemoryRebound(to: Int8.self, capacity: 8) {
    return strlen($0)
}
// length == 7
```

When you need to permanently rebind memory to a different type, first obtain a raw pointer to the memory and then call the `bindMemory(to:capacity:)` method on the raw pointer. The following example binds the memory referenced by `uint8Pointer` to one instance of the `UInt64` type:

```
let uint64Pointer = UnsafeRawPointer(uint8Pointer)
                          .bindMemory(to: UInt64.self, capacity: 1)
```

After rebinding the memory referenced by `uint8Pointer` to `UInt64`, accessing that pointer's referenced memory as a `UInt8` instance is undefined.

```
var fullInteger = uint64Pointer.pointee          // OK
var firstByte = uint8Pointer.pointee             // undefined
```

Alternatively, you can access the same memory as a different type without rebinding through untyped memory access, so long as the bound type and the destination type are trivial types. Convert your pointer to an `UnsafeRawPointer` instance and then use the raw pointer's `load(fromByteOffset:as:)` method to read values.

```
let rawPointer = UnsafeRawPointer(uint64Pointer)
let fullInteger = rawPointer.load(as: UInt64.self)   // OK
let firstByte = rawPointer.load(as: UInt8.self)      // OK
```

Performing Typed Pointer Arithmetic

Pointer arithmetic with a typed pointer is counted in strides of the pointer's `Pointee` type. When you add to or subtract from an `UnsafePointer` instance, the result is a new pointer of the same

type, offset by that number of instances of the `Pointee` type.

```
// 'intPointer' points to memory initialized with [10, 20, 30, 40]
let intPointer: UnsafePointer<Int> = ...

// Load the first value in memory
let x = intPointer.pointee
// x == 10

// Load the third value in memory
let offsetPointer = intPointer + 2
let y = offsetPointer.pointee
// y == 30
```

You can also use subscript notation to access the value in memory at a specific offset.

```
let z = intPointer[2]
// z == 30
```

Implicit Casting and Bridging

When calling a function or method with an `UnsafePointer` parameter, you can pass an instance of that specific pointer type, pass an instance of a compatible pointer type, or use Swift's implicit bridging to pass a compatible pointer.

For example, the `printInt(atAddress:)` function in the following code sample expects an `UnsafePointer<Int>` instance as its first parameter:

```
func printInt(atAddress p: UnsafePointer<Int>) {
    print(p.pointee)
}
```

As is typical in Swift, you can call the `printInt(atAddress:)` function with an `UnsafePointer` instance. This example passes `intPointer`, a pointer to an `Int` value, to `print(address:)`.

```
printInt(atAddress: intPointer)
// Prints "42"
```

Because a mutable typed pointer can be implicitly cast to an immutable pointer with the same `Pointee` type when passed as a parameter, you can also call `printInt(atAddress:)` with an `UnsafeMutablePointer` instance.

```
let mutableIntPointer = UnsafeMutablePointer(mutating: intPointer)
printInt(atAddress: mutableIntPointer)
// Prints "42"
```

Alternatively, you can use Swift's *implicit bridging* to pass a pointer to an instance or to the elements of an array. The following example passes a pointer to the `value` variable by using inout syntax:

```
var value: Int = 23
printInt(atAddress: &value)
// Prints "23"
```

An immutable pointer to the elements of an array is implicitly created when you pass the array as an argument. This example uses implicit bridging to pass a pointer to the elements of `numbers` when calling `printInt(atAddress:)`.

```
let numbers = [5, 10, 15, 20]
printInt(atAddress: numbers)
// Prints "5"
```

You can also use inout syntax to pass a mutable pointer to the elements of an array. Because `printInt(atAddress:)` requires an immutable pointer, although this is syntactically valid, it isn't necessary.

```
var mutableNumbers = numbers
printInt(atAddress: &mutableNumbers)
```

No matter which way you call `printInt(atAddress:)`, Swift's type safety guarantees that you can only pass a pointer to the type required by the function—in this case, a pointer to an `Int`.

Important

The pointer created through implicit bridging of an instance or of an array's elements is only valid during the execution of the called function. Escaping the pointer to use after the execution of the function is undefined behavior. In particular, do not use implicit bridging when calling an `UnsafePointer` initializer.

```
var number = 5
let numberPointer = UnsafePointer<Int>(&number)
// Accessing 'numberPointer' is undefined behavior.
```

Topics

Instance Properties

`var customPlaygroundQuickLook: PlaygroundQuickLook`

A custom playground Quick Look for this instance.

`var pointee: Pointee`

Accesses the instance referenced by this pointer.

Instance Methods

`func bytes() -> MIDIPacket.ByteCollection`

`func deallocate()`

Deallocates the memory block previously allocated at this pointer.

`func pointer<Property>(to: KeyPath<Pointee, Property>) -> UnsafePointer<Property>?`

Obtain a pointer to the stored property referred to by a key path.

`func sequence() -> MIDIEventPacket.WordSequence`

`func sequence() -> MIDIPacket.ByteSequence`

`func unsafeSequence() -> MIDIPacketList.UnsafeSequence`

`func unsafeSequence() -> MIDIEventList.UnsafeSequence`

```
func withMemoryRebound<T, E, Result>(to: T.Type, capacity: Int, (Unsafe Pointer<T>) throws(E) -> Result) throws(E) -> Result
```

Executes the given closure while temporarily binding memory to the specified number of instances of type T.

```
func words() -> MIDIEventPacket.WordCollection
```

Subscripts

```
subscript(Int) -> Pointee
```

Accesses the pointee at the specified offset from this pointer.

Type Aliases

```
typealias Distance
```

A type that represents the distance between two pointers.

Default Implementations

- AtomicOptionalRepresentable Implementations
- AtomicRepresentable Implementations
- Comparable Implementations
- CustomReflectable Implementations
- Equatable Implementations
- Hashable Implementations
- Strideable Implementations

Relationships

Conforms To

`AtomicOptionalRepresentable`

Conforms when `Pointee` conforms to `Escapable`.

`AtomicRepresentable`

Conforms when `Pointee` conforms to `Escapable`.

`BitwiseCopyable`

Conforms when `Pointee` conforms to `Escapable`.

`CVarArg`

Conforms when `Pointee` conforms to `Escapable`.

`Comparable`

Conforms when `Pointee` conforms to `Escapable`.

`Copyable`

Conforms when `Pointee` conforms to `Escapable`.

`CustomDebugStringConvertible`

Conforms when `Pointee` conforms to `Escapable`.

`CustomReflectable`

Conforms when `Pointee` conforms to `Escapable`.

`Equatable`

Conforms when `Pointee` conforms to `Escapable`.

`Hashable`

Conforms when `Pointee` conforms to `Escapable`.

`Strideable`

Conforms when `Pointee` conforms to `Escapable`.

See Also

Typed Pointers

`struct UnsafeMutablePointer`

A pointer for accessing and manipulating data of a specific type.

`struct UnsafeBufferPointer`

A nonowning collection interface to a buffer of elements stored contiguously in memory.

`struct UnsafeMutableBufferPointer`

A nonowning collection interface to a buffer of mutable elements stored contiguously in memory.