Temporary Pointers

A temporary, or ephemeral, pointer in Swift is a pointer which is introduced by an implicit function argument conversion and is only valid for the lifetime of the function call it appears in. There are a few ways to create a temporary pointer:

- Using an inout-to-pointer conversion by passing an argument with `&`:

  ```
  func foo(bar: UnsafePointer<Int>) { /*...*/ }
  var x: Int = 42
  foo(bar: &x)
  ```

  In the example above, the `bar` passed to `foo` is a temporary pointer to `x` which is only valid until `foo` returns.

  Not all inout-to-pointer conversions result in a temporary pointer. Passing global variables and static properties inout can produce non-ephemeral pointers, as long as they are stored and have no observers. Additionally, if they are of a tuple or struct type, their stored members without observers may also be passed inout as non-ephemeral pointers.

- Using a string-to-pointer conversion:

  ```
  func foo(bar: UnsafePointer<Int8>) { /*...*/ }
  var x: String = "hello, world!"
  foo(bar: x)
  ```

  In the example above, the `bar` passed to `foo` is a temporary pointer to a buffer containing the UTF-8 code units of `x` which is only valid until `foo` returns.

- Using an array-to-pointer conversion:

  ```
  func foo(bar: UnsafePointer<Bool>) { /*...*/ }
  var x: [Bool] = [true, false, true]
  foo(bar: x)
  ```

  In the example above, the `bar` passed to `foo` is a temporary pointer to the elements of `x` which is only valid until `foo` returns.

Temporary pointers may only be passed as arguments to functions which do not store the pointer value or otherwise allow it to escape the function's scope. The Swift compiler is able to diagnose some, but not all, violations of this rule. Misusing a temporary pointer by allowing it to outlive the enclosing function call results in undefined behavior. For example, consider the following incorrect code:

```
var x = 42
let ptr = UnsafePointer(&x)
// Do something with ptr.
```

This code is invalid because the initializer of `UnsafePointer` stores its argument, causing it to outlive the `UnsafePointer` initializer call. Instead, this code should use `withUnsafePointer` to access a pointer to `x` with an explicitly defined scope:

```
var x = 42
withUnsafePointer(to: &x) { ptr in
  // Do something with ptr, but don't allow it to escape this closure!
}
```

It's important to note that the `withUnsafe*` functions can also result in undefined behavior if used improperly. For example, the following incorrect code is equivalent to the original temporary pointer example:

```
var x = 42
let ptr = withUnsafePointer(to: &x) { $0 }
// Do something with ptr.
```

This code is invalid because the pointer to `x` is only valid until `withUnsafePointer` returns, but it escapes the closure when it is returned and assigned to `ptr`.

To learn more about correctly using unsafe pointer APIs, see the Swift standard library documentation of `UnsafePointer` and related types.