Memory Safety

On This Page

By default, Swift prevents unsafe behavior from happening in your code. For example, Swift ensures that variables are initialized before they're used, memory isn't accessed after it's been deallocated, and array indices are checked for out-of-bounds errors.

Swift also makes sure that multiple accesses to the same area of memory don't conflict, by requiring code that modifies a location in memory to have exclusive access to that memory. Because Swift manages memory automatically, most of the time you don't have to think about accessing memory at all. However, it's important to understand where potential conflicts can occur, so you can avoid writing code that has conflicting access to memory. If your code does contain conflicts, you'll get a compile-time or runtime error.

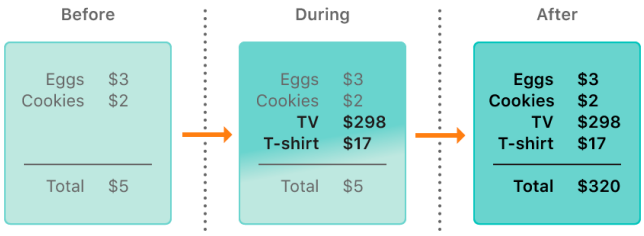
Understanding Conflicting Access to Memory

Access to memory happens in your code when you do things like set the value of a variable or pass an argument to a function. For example, the following code contains both a read access and a write access:

```
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We're number \(one)!")
```

A conflicting access to memory can occur when different parts of your code are trying to access the same location in memory at the same time. Multiple accesses to a location in memory at the same time can produce unpredictable or inconsistent behavior. In Swift, there are ways to modify a value that span several lines of code, making it possible to attempt to access a value in the middle of its own modification.

You can see a similar problem by thinking about how you update a budget that's written on a piece of paper. Updating the budget is a two-step process: First you add the items' names and prices, and then you change the total amount to reflect the items currently on the list. Before and after the update, you can read any information from the budget and get a correct answer, as shown in the figure below.



While you're adding items to the budget, it's in a temporary, invalid state because the total amount hasn't been updated to reflect the newly added items. Reading the total amount during the process of adding an item gives you incorrect information.

This example also demonstrates a challenge you may encounter when fixing conflicting access to memory: There are sometimes multiple ways to fix the conflict that produce different answers, and it's not always obvious which answer is correct. In this example, depending on whether you wanted the original total amount or the updated total amount, either $5 or $320 could be the correct answer. Before you can fix the conflicting access, you have to determine what it was intended to do.

> NOTE
>
> If you've written concurrent or multithreaded code, conflicting access to memory might be a familiar problem. However, the conflicting access discussed here can happen on a single thread and *doesn't* involve concurrent or multithreaded code.

> If you have conflicting access to memory from within a single thread, Swift guarantees that you'll get an error at either compile time or runtime. For multithreaded code, use Thread Sanitizer to help detect conflicting access across threads.

Characteristics of Memory Access

There are three characteristics of memory access to consider in the context of conflicting access: whether the access is a read or a write, the duration of the access, and the location in memory being accessed. Specifically, a conflict occurs if you have two accesses that meet all of the following conditions:

- At least one is a write access.
- They access the same location in memory.
- Their durations overlap.

The difference between a read and write access is usually obvious: a write access changes the location in memory, but a read access doesn't. The location in memory refers to what is being accessed—for example, a variable, constant, or property. The duration of a memory access is either instantaneous or long-term.

An access is *instantaneous* if it's not possible for other code to run after that access starts but before it ends. By their nature, two instantaneous accesses can't happen at the same time. Most memory access is instantaneous. For example, all the read and write accesses in the code listing below are instantaneous:

```
func oneMore(than number: Int) -> Int {
    return number + 1
}

var myNumber = 1
myNumber = oneMore(than: myNumber)
print(myNumber)
// Prints "2"
```

However, there are several ways to access memory, called *long-term* accesses, that span the execution of other code. The difference between instantaneous access and long-term access is that it's possible for other code to run after a long-term access starts but before it ends, which is called *overlap*. A long-term access can overlap with other long-term accesses and instantaneous accesses.

Overlapping accesses appear primarily in code that uses in-out parameters in functions and methods or mutating methods of a structure. The specific kinds of Swift code that use long-term accesses are discussed in the sections below.

Conflicting Access to In-Out Parameters

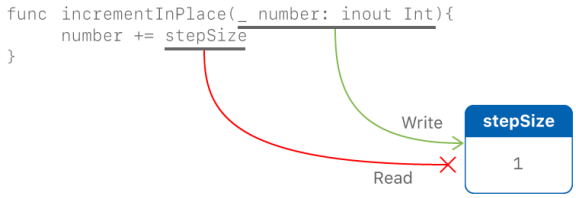
A function has long-term write access to all of its in-out parameters. The write access for an in-out parameter starts after all of the non-in-out parameters have been evaluated and lasts for the entire duration of that function call. If there are multiple in-out parameters, the write accesses start in the same order as the parameters appear.

One consequence of this long-term write access is that you can't access the original variable that was passed as in-out, even if scoping rules and access control would otherwise permit it—any access to the original creates a conflict. For example:

```
var stepSize = 1

func incrementInPlace(_ number: inout Int) {
    number += stepSize
}

incrementInPlace(&stepSize)
```

```
// Error: conflicting accesses to stepSize
```

In the code above, `stepSize` is a global variable, and it is normally accessible from within `incrementInPlace(_:)`. However, the read access to `stepSize` overlaps with the write access to `number`. As shown in the figure below, both `number` and `stepSize` refer to the same location in memory. The read and write accesses refer to the same memory and they overlap, producing a conflict.



One way to solve this conflict is to make an explicit copy of `stepSize`:

```
// Make an explicit copy.
var copyOfStepSize = stepSize
incrementInPlace(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
```

When you make a copy of `stepSize` before calling `incrementInPlace(_:)`, it's clear that the value of `copyOfStepSize` is incremented by the current step size. The read access ends before the write access starts, so there isn't a conflict.

Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

```
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}
var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore)  // OK
balance(&playerOneScore, &playerOneScore)
// Error: conflicting accesses to playerOneScore
```

The `balance(_:_:)` function above modifies its two parameters to divide the total value evenly between them. Calling it with `playerOneScore` and `playerTwoScore` as arguments doesn't produce a conflict—there are two write accesses that overlap in time, but they access different locations in memory. In contrast, passing `playerOneScore` as the value for both parameters produces a conflict because it tries to perform two write accesses to the same location in memory at the same time.

> NOTE
>
> Because operators are functions, they can also have long-term accesses to their in-out parameters. For example, if `balance(_:_:)` was an operator function named `<^>`, writing `playerOneScore <^> playerOneScore` would result in the same conflict as `balance(&playerOneScore, &playerOneScore)`.

Conflicting Access to self in Methods

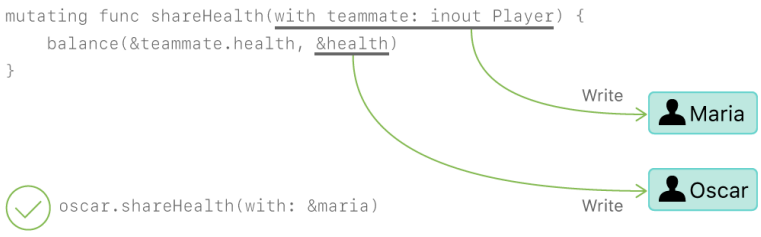
A mutating method on a structure has write access to `self` for the duration of the method call. For example, consider a game where each player has a health amount, which decreases when taking damage, and an energy amount, which decreases when using special abilities.

```
struct Player {
    var name: String
    var health: Int
    var energy: Int

    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
}
```

In the `restoreHealth()` method above, a write access to `self` starts at the beginning of the method and lasts until the method returns. In this case, there's no other code inside `restoreHealth()` that could have an overlapping access to the properties of a `Player` instance. The `shareHealth(with:)` method below takes another `Player` instance as an in-out parameter, creating the possibility of overlapping accesses.

```
extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)  // OK
```

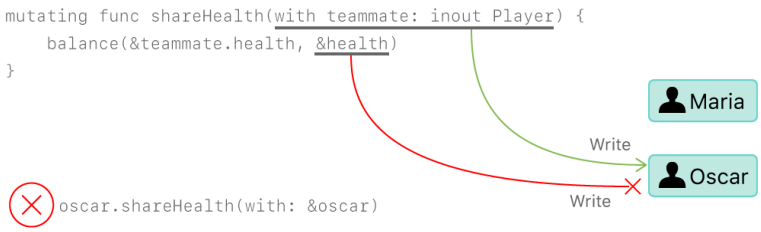
In the example above, calling the `shareHealth(with:)` method for Oscar's player to share health with Maria's player doesn't cause a conflict. There's a write access to `oscar` during the method call because `oscar` is the value of `self` in a mutating method, and there's a write access to `maria` for the same duration because `maria` was passed as an in-out parameter. As shown in the figure below, they access different locations in memory. Even though the two write accesses overlap in time, they don't conflict.



However, if you pass `oscar` as the argument to `shareHealth(with:)`, there's a conflict:

```
oscar.shareHealth(with: &oscar)
// Error: conflicting accesses to oscar
```

The mutating method needs write access to `self` for the duration of the method, and the in-out parameter needs write access to `teammate` for the same duration. Within the method, both `self` and `teammate` refer to the same location in memory—as shown in the figure below. The two write accesses refer to the same memory and they overlap, producing a conflict.



Conflicting Access to Properties

Types like structures, tuples, and enumerations are made up of individual constituent values, such as the properties of a structure or the elements of a tuple. Because these are value types, mutating any piece of the value mutates the whole value, meaning read or write access to one of the properties requires read or write access to the whole value. For example, overlapping write accesses to the elements of a tuple produces a conflict:

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// Error: conflicting access to properties of playerInformation
```

In the example above, calling `balance(_:_:)` on the elements of a tuple produces a conflict because there are overlapping write accesses to `playerInformation`. Both `playerInformation.health` and `playerInformation.energy` are passed as in-out parameters, which means `balance(_:_:)` needs write access to them for the duration of the function call. In both cases, a write access to the tuple element requires a write access to the entire tuple. This means there are two write accesses to `playerInformation` with durations that overlap, causing a conflict.

The code below shows that the same error appears for overlapping write accesses to the properties of a structure that's stored in a global variable.

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy)  // Error
```

In practice, most access to the properties of a structure can overlap safely. For example, if the variable `holly` in the example above is changed to a local variable instead of a global variable, the compiler can prove that overlapping access to stored properties of the structure is safe:

```
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10, energy: 10)
    balance(&oscar.health, &oscar.energy)  // OK
}
```

In the example above, Oscar's health and energy are passed as the two in-out parameters to `balance(_:_:)`. The compiler can prove that memory safety is preserved because the two stored properties don't interact in any way.

The restriction against overlapping access to properties of a structure isn't always necessary to preserve memory safety. Memory safety is the desired guarantee, but exclusive access is a stricter requirement than memory safety—which means some code preserves memory safety, even though it violates exclusive access to memory. Swift allows this memory-safe code if the compiler can prove that the nonexclusive access to memory is still safe. Specifically, it can prove that overlapping access to properties of a structure is safe if the following conditions apply:

- You're accessing only stored properties of an instance, not computed properties or class properties.
- The structure is the value of a local variable, not a global variable.
- The structure is either not captured by any closures, or it's captured only by nonescaping closures.

If the compiler can’t prove the access is safe, it doesn’t allow the access.