TODAY

- Memory safety
- Generic Functions
- Generic Types
- Extending Generic Types
- Type Constraints
- Where Clauses
- Alerts
- Midterm
- (maybe) assign2
CONFLICTING ACCESS TO PROPERTIES

The issue:

• structs, tuples, and enumerations are value-based
• mutating any piece of the value mutates the whole value

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

• .health and .energy are the same tuple, so conflicting writes

CONFLICTING ACCESS TO STRUCTS

• .health and .energy are the same tuple, so conflicting writes

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

• Part of the above problem is because holly global,
  • use a local variable to constrain compiler proofs:

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

• Compiler will only allow access that it can prove safe
• “Memory safety” is goal,
  • “exclusive access” stronger than needed
• Compiler can prove “exclusive access” for access iff:
  • accessing stored properties only
  • local variable
  • not captured by closures
    (or only by non-escaping closures)

• non-escaping closures are:
  • passed to a function
  • executed by function before it returns
  • not stored
GENERIC FUNCTIONS

```swift
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

- Have to build another function for strings, etc...

---

GENERIC FUNCTIONS

```swift
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let tempA = a
    a = b
    b = tempA
}
```

- “T” is a placeholder (also called "type parameter")
- doesn't specify a type
- does say `a`, `b` must be same type
- Naming type parameters
  - use descriptive names, such as `Element`, or `Item`
  - or `T`, `U`, `V`
**GENERIC TYPES**

```swift
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

- **Element placeholder used:**
  - to create property "items"
  - to specify arg to `push()`
  - return value of `pop()`

```swift
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
```

---

**GENERIC TYPES**

- var numbers = [1,2,3]
- var numbers: [Int] = [1,2,3]
- var numbers: Array<Int> = [1,2,3]  // full generic annotation
- Generic types are custom **structs, classes, and enumerations** that can accept any type
EXTENSIONS OF GENERIC TYPES

- Let’s say we want to extend a generic parameter.
  - type parameter name must be same

```swift
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

TYPE CONSTRAINTS

- Stack defined previous slide can work w/ any type
- Sometimes useful to enforce type constraints
  - by adding a required type or protocol to type parameter
- For example, Dictionary requires keys to be hashable
  - otherwise, none could be found, or inserted

```swift
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

- Single class or protocol constraint, either:
  - someT must be a subclass of SomeClass, or
  - someU is required to conform to the protocol SomeProtocol.
TYPE CONSTRAINTS

- Maybe want to search for a string in an array....

```swift
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

- but might be useful for non-strings as well, so....

```swift
func findIndex<T>(of valueToFind: T, in array: [T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

- Doesn’t work, needs `T` to be constrained to `Equatable`

Instances that are `equatable` can be compared for equality:

```swift
func findIndex<T: Equatable>(of valueToFind: T, in array: [T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```
TYPE ASSOCIATIONS

- Protocols use specific types, what if we want them abstract?
  - “type associations”

```swift
protocol Container {
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

- Container requires three capabilities:
  - `append()` Item func
  - `count()` Items func
  - retrieval of Item through subscripting

- Doesn't specify:
  - how the items are stored
  - their type

- Does specify:
  - base type must be equatable

---

```swift
struct IntStack: Container {
    // original IntStack implementation
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias Item = Int
    mutating func append(_ item: Item) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

Concrete type not actually needed because of type inference from funcs.
TYPE ASSOCIATIONS W/ GENERICS

```swift
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

GENERIC WHERE CLAUSES

- A Generic Where Clause can require:
  - an associated type (“in generic protocols”) to conform to protocol
  - certain type parameters and associated types to be the same
- Write it right before opening curly brace of type or func
func allItemsMatch<C1: Container, C2: Container>(someContainer: C1, anotherContainer: C2) -> Bool
where C1.Item == C2.Item, C1.Item: Equatable {

    // Check that both containers contain the same number of items.
    if someContainer.count != anotherContainer.count {
        return false
    }

    // Check each pair of items to see if they're equivalent.
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }

    // All items match, so return true.
    return true
}

Remember: Item is the associated type for container

allItemsMatch() checks to see if instances contain same items in same order
  • they don't have to be the same type of container
  • they do have to be the same type of item

The type constraints and generic where clause require:
  (1) C1 must conform to the Container protocol
  (2) C2 must also conform to the Container protocol
  (3) The Item for C1 must be the same as the Item for C2
  (4) The Item for C1 must conform to the Equatable protocol
  • (3) and (4) mean that C2 must also conform to Equatable
func allItemsMatch<C1: Container, C2: Container>(someContainer: C1, anotherContainer: C2) -> Bool where C1.Item == C2.Item, C1.Item: Equatable {
    // Check that both containers contain the same number of items.
    if someContainer.count != anotherContainer.count {
        return false
    }
    // Check each pair of items to see if they're equivalent.
    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }
    // All items match, so return true.
    return true
}

// P r i n t s  " A l l  i t e m s  m a t c h . "

In practice:

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
// Prints "All items match."

• Both Stack and Array conform to Container
• Both stackOfStrings, arrayOfStrings contain String
**Generic Where Clauses**

- Generic where clauses can even be used in defining an abstract protocol’s own type constraints:

```swift
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

```swift
extension Stack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack {
        var result = Stack()
        for index in (count-size)...<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack.
}
```

```
var stackOfInts = Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
let suffix = stackOfInts.suffix(2)
// suffix is a stack containing 20 and 30
```

**Extension W/ Generic Where**

- Extensions can be made to only work for conforming base types
  - extension defined w/ a `Where` clause
  - `func`s in extension only defined for base types that conform

```swift
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```
MODAL INTERACTION

- What is modal interaction?
  - all focus is on one control, user can’t click elsewhere

- Not something we love
  - interrupts flow, need to make sure can back out easily

- Two kinds:
  - action sheets
    - slide in from bottom or top
    - multiple answers
  - alerts
    - middle of screen
    - yes/no, forward/back

ALERTS
let alert = UIAlertController(
    title: "New Game?",
    message: "All data will be lost.",
    preferredStyle: .alert)

// not shown in picture
alert.addTextField(configurationHandler: {
    tf in tf.text = "default text"})
let alert = UIAlertController(
    title: "New Game?",
    message: "All data will be lost.",
    preferredStyle: .alert)

// not shown in picture
alert.addTextField{tf in tf.text = "default text"}

alert.addAction(UIAlertAction(
    title: "OK", style: .default,
    handler: { (action: UIAlertAction!) in
        print("yes: \(alert.textFields![0].text!)")
    })
)

alert.addAction(UIAlertAction(
    title: "Cancel",
    style: .cancel,
    handler: { (action: UIAlertAction!) in
        print("nah")
    })
)

present(alert, animated: true, completion: nil)
ACTION SHEETS

```swift
let alert = UIAlertController(
    title: "Which path?",
    message: "All hope is almost lost.",
    preferredStyle: .actionSheet)

alert.addAction(UIAlertAction(
    title: "LEFT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("LEFT") })))
```

ACTION SHEETS

```swift
let alert = UIAlertController(
    title: "Which path?",
    message: "All hope is almost lost.",
    preferredStyle: .actionSheet)

alert.addAction(UIAlertAction(
    title: "LEFT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("LEFT") })))
```
ACTION SHEETS

let alert = UIAlertController(
    title: "Which path?",
    message: "All hope is almost lost.",
    preferredStyle: .actionSheet)

alert.addAction(UIAlertAction(
    title: "LEFT",
    style: .default,
   handler: { (action: UIAlertAction!) in
        print("LEFT") }))

alert.addAction(UIAlertAction(
    title: "STRAIGHT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("STRAIGHT") }))

alert.addAction(UIAlertAction(
    title: "RIGHT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("RIGHT") }))
### ACTION SHEETS

```swift
let alert = UIAlertController(
    title: "Which path?",
    message: "All hope is almost lost.",
    preferredStyle: .actionSheet)

alert.addAction(UIAlertAction(
    title: "LEFT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("LEFT") })))

alert.addAction(UIAlertAction(
    title: "STRAIGHT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("STRAIGHT") })))

alert.addAction(UIAlertAction(
    title: "RIGHT",
    style: .default,
    handler: { (action: UIAlertAction!) in
        print("RIGHT") })))
present(alert, animated: true, completion: nil)
```

### MID-TERM

- Coverage: everything through this slide
  - (almost) all about swift
- Study materials
  - slides (on schedule)
  - demo videos (on schedule)
  - previous midterm1
    - material is NOT entirely the same
  - code practice sheet (released 5pm tomorrow on piazza)
- Test specifics
  - closed everything
  - there will be code (generics / memory)
    - see practice sheet