Parameter Packs

or how Swift decided to Become C++

Vincent Pradeilles 🇫🇷 (@v_pradeilles) – PhotoRoom

First, let’s talk about SwiftUI…

```
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
}
```

```swift
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
}
```

```
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
}
```

Extra argument in call

Building content

```
static func buildBlock() -> EmptyView
```

Builds an empty view from a block containing no statements.

```
static func buildBlock<Content>(Content) -> Content
```

Passes a single view written as a child view through unmodified.

```
static func buildBlock<C0, C1>(C0, C1) -> TupleView<(C0, C1)>

static func buildBlock<C0, C1, C2>(C0, C1, C2) -> TupleView<(C0, C1, C2)>

static func buildBlock<C0, C1, C2, C3>(C0, C1, C2, C3) -> TupleView<(C0, C1, C2, C3)>

static func buildBlock<C0, C1, C2, C3, C4>(C0, C1, C2, C3, C4) -> TupleView<(C0, C1,
C2, C3, C4)>

static func buildBlock<C0, C1, C2, C3, C4, C5>(C0, C1, C2, C3, C4, C5) -> Tuple
View<(C0, C1, C2, C3, C4, C5)>

static func buildBlock<C0, C1, C2, C3, C4, C5, C6>(C0, C1, C2, C3, C4, C5, C6) -> Tuple
View<(C0, C1, C2, C3, C4, C5, C6)>

static func buildBlock<C0, C1, C2, C3, C4, C5, C6, C7>(C0, C1, C2, C3, C4, C5, C6, C7)
-> TupleView<(C0, C1, C2, C3, C4, C5, C6, C7)>

static func buildBlock<C0, C1, C2, C3, C4, C5, C6, C7, C8>(C0, C1, C2, C3, C4, C5, C6,
C7, C8) -> TupleView<(C0, C1, C2, C3, C4, C5, C6, C7, C8)>

static func buildBlock<C0, C1, C2, C3, C4, C5, C6, C7, C8, C9>(C0, C1, C2, C3, C4, C5,
C6, C7, C8, C9) -> TupleView<(C0, C1, C2, C3, C4, C5, C6, C7, C8, C9)>
```

Xcode 15

```
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
}
```

```swift
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
}
```

```swift
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
    Text("12")
    Text("13")
    Text("14")
    Text("15")
    Text("16")
}
```

```swift
VStack {
    Text("1")
    Text("2")
    Text("3")
    Text("4")
    Text("5")
    Text("6")
    Text("7")
    Text("8")
    Text("9")
    Text("10")
    Text("11")
    Text("12")
    Text("13")
    Text("14")
    Text("15")
    Text("16")
    Text("17")
    Text("18")
    Text("19")
    Text("20")
    Text("21")
    Text("22")
    Text("23")
    Text("24")
    Text("25")
    Text("26")
    Text("27")
    Text("28")
    Text("29")
    Text("30")
    Text("31")
}
```

Building content

```
static func buildBlock() -> EmptyView
```

Builds an empty view from a block containing no statements.

```
static func buildBlock<Content>(Content) -> Content
```

Passes a single view written as a child view through unmodified.

```
static func buildBlock<each Content>(repeat each Content) -> TupleView<(repeat each Content)>
```

```
static func buildExpression<Content>(Content) -> Content
```

Builds an expression within the builder.

```
public static func buildBlock<each Content>(
    _ content: repeat each Content
) -> TupleView<(repeat each Content)>
where repeat each Content : View
```

```
public static func buildBlock<each Content>(
    _ content: repeat each Content
) -> TupleView<(repeat each Content)>
where repeat each Content : View
```

Use Case #01

```swift
let tupleOfOne  = groupInTuple(23)

let tupleOfTwo  = groupInTuple(23, "Hello!")

let tupleOfFive = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷")

let tupleOfSix  = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷", 34.5)
```

```swift
// returns: (23)
let tupleOfOne  = groupInTuple(23)
// returns: (23, "Hello!")
let tupleOfTwo  = groupInTuple(23, "Hello!")
// returns: (23, "Hello!", "Swift", "Connection", "🇫🇷")
let tupleOfFive = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷")
// returns: (23, "Hello!", "Swift", "Connection", "🇫🇷", 34.5)
let tupleOfSix  = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷", 34.5)
```

```
func groupInTuple
```

```
func groupInTuple<each T>
```

```
func groupInTuple<each T>(
    _ value:
)
```

```
func groupInTuple<each T>(
    _ value: repeat
)
```

```
func groupInTuple<each T>(
    _ value: repeat each T
)
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> ( )
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T)
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {

}
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {
    return ( )
}
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {
    return (repeat )
}
```

```swift
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {
    return (repeat each value)
}
```

```
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {
    return (repeat each value)
}
```

```swift
func groupInTuple<each T>(
    _ value: repeat each T
) -> (repeat each T) {
    return (repeat each value)
}
```

```swift
// returns: (23)
let tupleOfOne  = groupInTuple(23)
// returns: (23, "Hello!")
let tupleOfTwo  = groupInTuple(23, "Hello!")
// returns: (23, "Hello!", "Swift", "Connection", "🇫🇷")
let tupleOfFive = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷")
// returns: (23, "Hello!", "Swift", "Connection", "🇫🇷", 34.5)
let tupleOfSix  = groupInTuple(23, "Hello!", "Swift", "Connection", "🇫🇷", 34.5)
```

Use Case #02

```
makePairs(
    firsts: 1, 2, "c", "d",
    seconds: "a", "b", 3.0, 4.0
)
```

```
// returns: ((1, "a"), (2, "b"), ("c", 3.0), ("d", 4.0))
makePairs(
    firsts: 1, 2, "c", "d",
    seconds: "a", "b", 3.0, 4.0
)
```

```
func makePairs
```

```
func makePairs<each First>
```

```
func makePairs<each First, each Second>
```

```
func makePairs<each First, each Second>(

)
```

```
func makePairs<each First, each Second>(
    firsts: ,
    seconds:
)
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
)
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> ( )
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat ( ))
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat (each First, each Second))
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat (each First, each Second)) {

}
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat (each First, each Second)) {
    return ( )
}
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat (each First, each Second)) {
    return (repeat ( ))
}
```

```
func makePairs<each First, each Second>(
    firsts: repeat each First,
    seconds: repeat each Second
) -> (repeat (each First, each Second)) {
    return (repeat (each firsts, each seconds))
}
```

```
// returns: ((1, "a"), (2, "b"), ("c", 3.0), ("d", 4.0))
makePairs(
    firsts: 1, 2, "c", "d",
    seconds: "a", "b", 3.0, 4.0
)
```

Use Case #03

```
totalCount(
    [1, 2, 3]
)

totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"]
)

totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!”],
    [42.0, 100.0, 0.0]
)
```

```swift
// returns: 3
totalCount(
    [1, 2, 3]
)

// returns: 5
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"]
)

// returns: 8
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"],
    [42.0, 100.0, 0.0]
)
```

```
// returns: 3
totalCount(
    [1, 2, 3]
```

```
// returns: 3
totalCount(
    [1, 2, 3]
```

```swift
// returns: 3
totalCount(
    [1, 2, 3]
)

// returns: 5
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"]
)

// returns: 8
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"],
    [42.0, 100.0, 0.0]
)
```

```
func totalCount
```

```
func totalCount<each C>
```

```
func totalCount<each C: Collection>
```

```
func totalCount<each C: Collection>(

)
```

```
func totalCount<each C: Collection>(
    _ collection:
)
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
)
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {

}
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    return result
}
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat ( )

    return result
}
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat (result += )

    return result
}
```

```swift
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat (result += (each collection))

    return result
}
```

```swift
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat (result += (each collection).count)

    return result
}
```

```swift
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat result += (each collection).count

    return result
}
```


This is when the fun begins


This is when the fun begins

```swift
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    repeat result += (each collection).count

    return result
}
```

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    func inner<T: Collection>(
        _ collection: T
    ) -> Int {
        collection.count
    }

    repeat result += inner(each collection)

    return result
}
```

```swift
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    func inner<T: Collection>(
        _ collection: T
    ) -> Int {
        collection.count
    }
```

All list operations can be expressed using pack expansion expressions by factoring code involving statements into a function or closure.

```
func totalCount<each C: Collection>(
    _ collection: repeat each C
) -> Int {
    var result = 0

    func inner<T: Collection>(
        _ collection: T
    ) -> Int {
        collection.count
    }

    repeat result += inner(each collection)

    return result
}
```

```swift
// returns: 3
totalCount(
    [1, 2, 3]
)

// returns: 5
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"]
)

// returns: 8
totalCount(
    [1, 2, 3],
    ["Hello", "Swift Connection!"],
    [42.0, 100.0, 0.0]
)
```

Use Case #04

```swift
areEqual(
    (42, "Hello", "Swift Connection"),
    (42, "Hello", "Swift Connection")
)

areEqual(
    (42, "Hello"),
    (42, "Swift Connection")
)
```

```swift
// returns: true
areEqual(
    (42, "Hello", "Swift Connection"),
    (42, "Hello", "Swift Connection")
)

// returns: false
areEqual(
    (42, "Hello"),
    (42, "Swift Connection")
)
```

```
// returns: true
```

```
// returns: true
```

```swift
// returns: true
areEqual(
    (42, "Hello", "Swift Connection"),
    (42, "Hello", "Swift Connection")
)

// returns: false
areEqual(
    (42, "Hello"),
    (42, "Swift Connection")
)
```

```
func areEqual
```

```
func areEqual<each Element>
```

```
func areEqual<each Element: Equatable>
```

```
func areEqual<each Element: Equatable>(
    _ left: ,
    _ right:
)
```

```swift
func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
)
```

```swift
func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
) -> Bool
```

```swift
func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
) -> Bool {

}
```

```swift
struct NotEqual: Error {}

func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
) -> Bool {

    func throwIfNotEqual<T: Equatable>(
        _ left: T,
        _ right: T
    ) throws {
        guard left == right else { throw NotEqual() }
    }

}
```

```swift
struct NotEqual: Error {}

func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
) -> Bool {

    func throwIfNotEqual<T: Equatable>(
        _ left: T,
        _ right: T
    ) throws {
        guard left == right else { throw NotEqual() }
    }

    do {
        repeat try throwIfNotEqual(each left, each right)
    } catch {
        return false
    }

}
```

```swift
struct NotEqual: Error {}

func areEqual<each Element: Equatable>(
    _ left: (repeat each Element),
    _ right: (repeat each Element)
) -> Bool {

    func throwIfNotEqual<T: Equatable>(
        _ left: T,
        _ right: T
    ) throws {
        guard left == right else { throw NotEqual() }
    }

    do {
        repeat try throwIfNotEqual(each left, each right)
    } catch {
        return false
    }

    return true
}
```

```swift
// returns: true
areEqual(
    (42, "Hello", "Swift Connection"),
    (42, "Hello", "Swift Connection")
)

// returns: false
areEqual(
    (42, "Hello"),
    (42, "Swift Connection")
)
```

Recap

Recap

Recap

- Parameter Packs are a new feature of Swift 5.9

Recap

- Parameter Packs are a new feature of Swift 5.9
- They lift a long lasting limitation of Swift

Recap

- Parameter Packs are a new feature of Swift 5.9
- They lift a long lasting limitation of Swift
- They are quite powerful when writing framework or tooling level code

Recap

- Parameter Packs are a new feature of Swift 5.9
- They lift a long lasting limitation of Swift
- They are quite powerful when writing framework or tooling level code
- However, their ergonomics are not (yet) perfect


That's all Folks!

Newsletter

https://www.photoroom.com/company/