

Funciones y clausuras



Funciones

En computación, una subrutina o subprograma (también llamada procedimiento, función o rutina), como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

Definición de una función

```
func greet(person: String) -> String {  
    let greeting = "Hello, " + person + "!"  
    return greeting  
}
```

Llamada a la función

```
print(greet(person: "Anna"))
```

```
print(greet(person: "Brian"))
```

Características de las funciones

- Tienen nombre
- Disponen de una lista de parámetros
- Disponen de un valor de retorno
- En la llamada se añaden argumentos que tienen que encajar con los parámetros de la función

Tipos de funciones

Funciones con parámetros

```
func greetAgain(person: String) -> String {  
    return "Hello again, " + person + "!"  
}
```

```
print(greetAgain(person: "Anna"))
```


Funciones sin parámetros

```
func sayHelloWorld() -> String {  
    return "hello, world"  
}
```

```
print(sayHelloWorld())
```

Funciones con múltiples parámetros

```
func greet(person: String, alreadyGreeted: Bool) -> String {  
    if alreadyGreeted {  
        return greetAgain(person: person)  
    } else {  
        return greet(person: person)  
    }  
}
```

```
print(greet(person: "Tim", alreadyGreeted: true))
```

Funciones sin valor de retorno

```
func greet(person: String) {  
    print("Hello, \(person)!")  
}
```

```
greet(person: "Dave")
```

Funciones con valor de retorno

```
func sayHelloWorld() -> String {  
    return "hello, world"  
}
```

```
print(sayHelloWorld())
```

Funciones con múltiples valores de retorno

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..<array.count] {  
        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```

Funciones con múltiples valores de retorno

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])  
  
print("min is \(bounds.min) and max is \(bounds.max)")
```

Retorno de tuplas opcionales

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {  
    if array.isEmpty { return nil }  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..<array.count] {  
        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```

Retorno de tuplas opcionales

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {  
    print("min is \(bounds.min) and max is \(bounds.max)")  
}
```


Parámetros y argumentos

Parámetros y argumentos

- Nombres de parámetros: para utilizar dentro de la función
- Etiquetas de argumentos: para usarlos al hacer la llamada a la función

Parámetros y argumentos

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {  
    // In the function body, firstParameterName and secondParameterName  
    // refer to the argument values for the first and second parameters.  
}
```

```
someFunction(firstParameterName: 1, secondParameterName: 2)
```

Parámetros y argumentos

- Por defecto, los parámetros usan su nombre como etiqueta de argumento
- Todos los parámetros tienen que tener nombres únicos

Etiquetas de argumentos explícitas

```
func someFunction(argumentLabel parameterName: Int) {  
    // In the function body, parameterName refers to the argument value  
    // for that parameter.  
}
```

Etiquetas de argumentos explícitas

```
func greet(person: String, from hometown: String) -> String {  
    return "Hello \$(person)! Glad you could visit from \$(hometown)."  
}
```

```
print(greet(person: "Bill", from: "Cupertino"))  
// Prints "Hello Bill! Glad you could visit from Cupertino."
```

Anular una etiqueta de argumento

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
    // In the function body, firstParameterName and secondParameterName  
    // refer to the argument values for the first and second parameters.  
}
```

```
someFunction(1, secondParameterName: 2)
```

Documentación de funciones

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
    // In the function body, firstParameterName and secondParameterName  
    // refer to the argument values for the first and second parameters.  
}
```

```
someFunction(1, secondParameterName: 2)
```

```
someFunction(_:secondParameterName:) // En la documentación aparece así
```


Parámetros por defecto

- Permiten fijar un valor para un parámetro si no se incluye en los argumentos de la llamada
- Es conveniente que estén al final de la lista de parámetros

Parámetros por defecto

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {  
    // If you omit the second argument when calling this function, then  
    // the value of parameterWithDefault is 12 inside the function body.  
}
```

```
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)  
// parameterWithDefault is 6
```

```
someFunction(parameterWithoutDefault: 4)  
// parameterWithDefault is 12
```

Parámetros indeterminados

- Son parámetros que permiten introducir múltiples valores
- Se declaran poniendo ... detrás del tipo de dato
- Los valores tienen que ser del mismo tipo
- Los valores llegan a la función como un array del tipo apropiado
- Sólo puede haber uno y tiene que ser siempre el último de la lista

Parámetros indeterminados

```
func arithmeticMean(_ numbers: Double...) -> Double {  
    var total: Double = 0  
    for number in numbers {  
        total += number  
    }  
    return total / Double(numbers.count)  
}
```

```
arithmeticMean(1, 2, 3, 4, 5)  
// returns 3.0, which is the arithmetic mean of these five numbers
```

```
arithmeticMean(3, 8.25, 18.75)  
// returns 10.0, which is the arithmetic mean of these three numbers
```

Parámetros indeterminados

```
func sumarNumeros(_ numeros: Double...) -> Double {  
    return numeros.reduce(0, +)  
}
```

```
let suma = sumarNumeros(1, 2, 3, 4, 5)
```

```
print(suma)
```

Parámetros InOut

- Son parámetros cuyo valor puede ser modificado por la función y persiste después de terminar esta
- Se generan marcando con `inout` el parámetro
- En la llamada, las variables que se pasan se marcan con `&`
- No se pueden pasar literales o constantes como parámetros
- No pueden tener valor por defecto ni ser indeterminados

Parámetros InOut

```
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)")  
// Prints "someInt is now 107, and anotherInt is now 3"
```

Funciones como tipos de datos

Funciones como tipos de datos

- Toda función tiene tipo
- Está definido por los tipos de los parámetros y el tipo del valor de retorno

Tipo de dato de una función

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

```
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a * b  
}
```

// (Int, Int) -> Int mismo tipo de dato en ambas funciones

Tipo de dato de una función

```
func printHelloWorld() {  
    print("hello, world")  
}
```

```
// () -> Void
```

Utilizar funciones como tipos

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Utilizar funciones como tipos

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

```
print("Result: \(mathFunction(2, 3))")
```

Utilizar funciones como tipos

```
mathFunction = multiplyTwoInts  
print("Result: \ (mathFunction(2, 3))")
```

```
let anotherMathFunction = addTwoInts
```

Tipos de función como parámetros

- Podemos definir un parámetro de una función del tipo de otra función
- Permite que la implementación de la función varíe dependiendo de lo que le pasemos como parámetro (que será una función)

Tipos de función como parámetros

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFunction(a, b))")  
}
```

```
printMathResult(addTwoInts, 3, 5)  
// Prints "Result: 8"
```

```
printMathResult(multiplyTwoInts, 3, 5)  
// Prints "Result: 15"
```


Tipos de función como valor de retorno

- Podemos definir el valor de retorno de una función del tipo de otra función
- Después de la \rightarrow de la función describimos el tipo de función

Tipos de función como valor de retorno

```
func stepForward(_ input: Int) -> Int {  
    return input + 1  
}
```

```
func stepBackward(_ input: Int) -> Int {  
    return input - 1  
}
```

Tipos de función como valor de retorno

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
    return backward ? stepBackward : stepForward  
}
```

```
var currentValue = 3  
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
// moveNearerToZero now refers to the stepBackward() function
```

Tipos de función como valor de retorno

```
var currentValue = 3
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

Funciones anidadas

- Una función puede definir dentro de ella otra función anidada
- La función anidada no es visible desde fuera de la función que la engloba
- La función que la engloba puede llamarla
- La función que la engloba la puede devolver como valor de retorno a un ámbito diferente

Funciones anidadas

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
  func stepForward(input: Int) -> Int { return input + 1 }  
  func stepBackward(input: Int) -> Int { return input - 1 }  
  return backward ? stepBackward : stepForward  
}
```

```
var currentValue = -4  
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
// moveNearerToZero now refers to the nested stepForward() function
```

```
while currentValue != 0 {  
  print("\(currentValue)... ")  
  currentValue = moveNearerToZero(currentValue)  
}
```

```
print("zero!")  
// -4...  
// -3...  
// -2...  
// -1...  
// zero!
```

Clausuras

En Informática, una clausura es una función que es evaluada en un entorno conteniendo una o más variables dependientes de otro entorno. Cuando es llamada, la función puede acceder a estas variables.

Diferencias entre clousures y funciones

- **Funciones:**

- Bloques de código nombrados.
- Pueden ser reutilizados múltiples veces.
- Se declaran con la palabra clave `func`.

```
func saludar(nombre: String) {  
    print("¡Hola, \(nombre)!")  
}  
saludar(nombre: "María") // Salida: ¡Hola, María!
```

- **Clousures:**

- Bloques de código anónimos.
- Pueden asignarse a variables o pasarse como parámetros.
- Se declaran usando llaves `{}`.

```
let saludarClosure = { (nombre: String) in  
    print("¡Hola, \(nombre)!")  
}  
saludarClosure("Juan") // Salida: ¡Hola, Juan!
```

Clausuras

- Bloques de funcionalidad autocontenidos
- Capturan referencias a las variables y constantes del ámbito en el que están definidas
- Las funciones son tipos especiales de clausuras
- Las clausuras son tipos por referencia cuando se asignan a variables o se pasan a funciones

Tipos de clausuras

- **Funciones globales:** clausuras con nombre y que no capturan variables
- **Funciones anidadas:** clausuras con nombre y que capturan las variables del ámbito de la función que las engloba
- **Expresiones de clausura:** clausuras sin nombre escritas en una notación simple que pueden capturar las variables de su entorno

Sintaxis de una clausura

```
{ (parameters) -> return_type in  
  statements  
}
```

Ejemplo: `sorted(by:)`

- La función `sorted(by:)` de la **librería** estándar de Swift ordena un array en función de un criterio expresado por una clausura
- `sorted(by:)` recibe como parámetros un array de elementos y una clausura que compara dos elementos y devuelve verdadero o falso

Ejemplo: sorted(by:)

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

```
func backward(_ s1: String, _ s2: String) -> Bool {  
    return s1 > s2  
}
```

```
var reversedNames = names.sorted(by: backward)  
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Ejemplo: sorted(by:)

// Clausura

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in  
    return s1 > s2  
})
```

// En una línea

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 } )
```

Ejemplo: sorted(by:)

// Inferencia de tipos

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

// Retorno implícito para expresiones de una sola línea

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```


Ejemplo: sorted(by:)

// Nombres de parámetros abreviados

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

// Función operador

```
reversedNames = names.sorted(by: >)
```

Clausuras posteriores

- Se pueden usar cuando la clausura es el último parámetro
- Se suelen usar si el código de la clausura es largo

Clausuras posteriores

```
func someFunctionThatTakesAClosure(closure: () -> Void) {  
    // function body goes here  
}
```

```
// Normal  
someFunctionThatTakesAClosure(closure: {  
    // closure's body goes here  
})
```

```
// Clausura posterior  
someFunctionThatTakesAClosure() {  
    // trailing closure's body goes here  
}
```

Clausuras posteriores

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

```
reversedNames = names.sorted() { $0 > $1 }
```

```
reversedNames = names.sorted { $0 > $1 }
```

Clausuras posteriores

```
let digitNames = [  
  0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four",  
  5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"  
]
```

```
let numbers = [16, 58, 510]
```

```
let strings = numbers.map {  
  (number) -> String in  
  var number = number  
  var output = ""  
  repeat {  
    output = digitNames[number % 10]! + output  
    number /= 10  
  } while number > 0  
  return output  
}
```

```
// strings is inferred to be of type [String]
```

```
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

Captura de valores

- Una clausura "captura" los valores del ámbito que la engloba
- La clausura puede acceder y modificar esos valores

Captura de valores

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {  
    var runningTotal = 0  
    func incrementer() -> Int {  
        runningTotal += amount  
        return runningTotal  
    }  
    return incrementer  
}
```

Captura de valores

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

```
incrementByTen()
```

```
// returns a value of 10
```

```
incrementByTen()
```

```
// returns a value of 20
```

```
incrementByTen()
```

```
// returns a value of 30
```


Captura de valores

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
```

```
incrementBySeven()  
// returns a value of 7
```

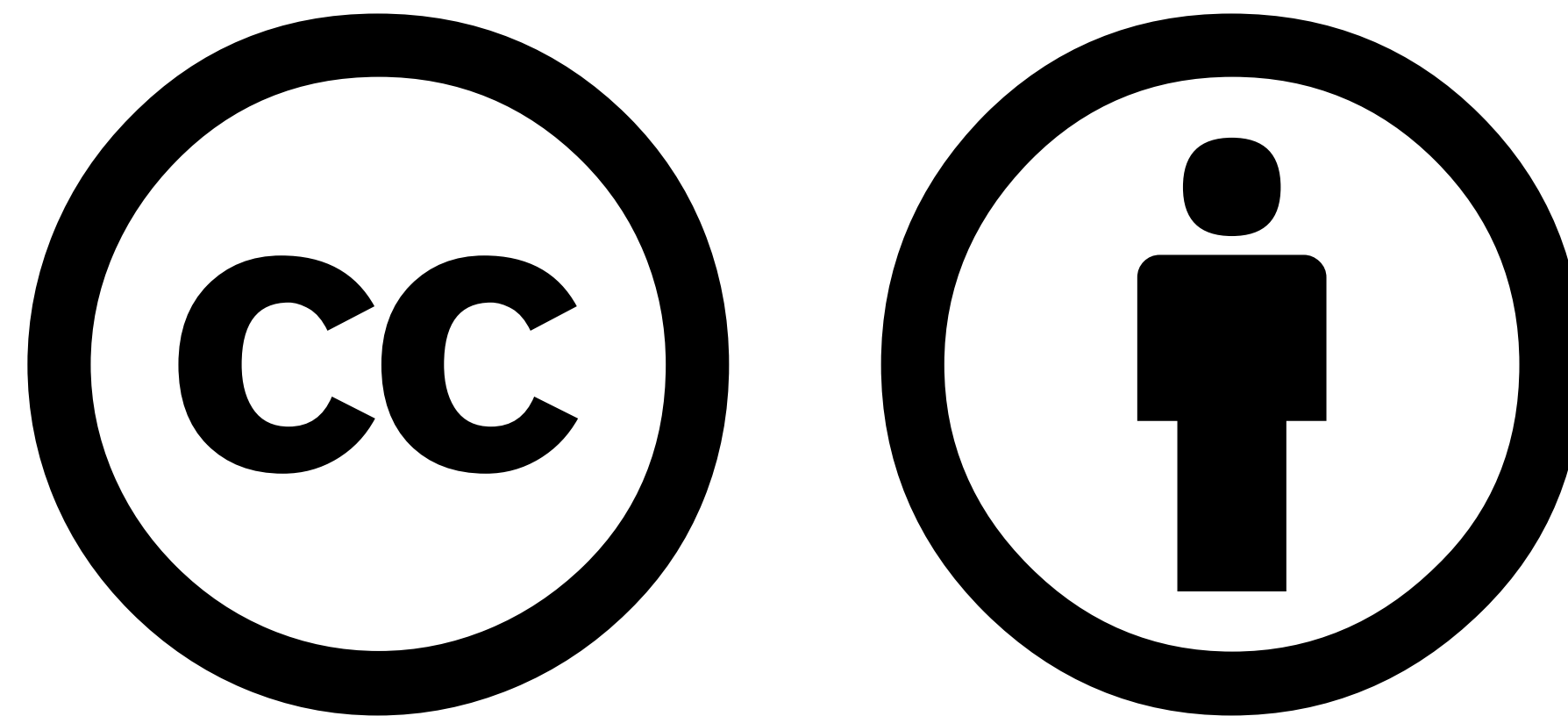
```
incrementByTen()  
// returns a value of 40
```

Las clausuras son tipos por referencia

```
let alsoIncrementByTen = incrementByTen
```

```
alsoIncrementByTen()  
// returns a value of 50
```

Escribe un closure llamado `multiplicarPorTres` que reciba un número entero y devuelva el resultado de multiplicarlo por 3. Luego, llama al closure con el número 5.



Excepto si se especifica lo contrario, esta presentación está bajo licencia

<https://creativecommons.org/licenses/by/4.0/>

© 2017 Ion Jaureguialzo Sarasola. Algunos derechos reservados.
© 2024 Inés Larrañaga Fdez. De Pinedo. Algunos derechos reservados.