







UNIVERSITY

ЛЕКЦІЯ 4

“Управління пам'яттю”



Problem of memory management

*“memory is always a **limited** resource “*

(Objective C Memory Management Essentials)



Problem of memory management

Solution designed by Apple - **ReferenceCounter**



Підрахунок посилань (Reference counting)

збільшує к-ть посилань при використанні об'єкту

зменшує при звільненні використання об'єкту

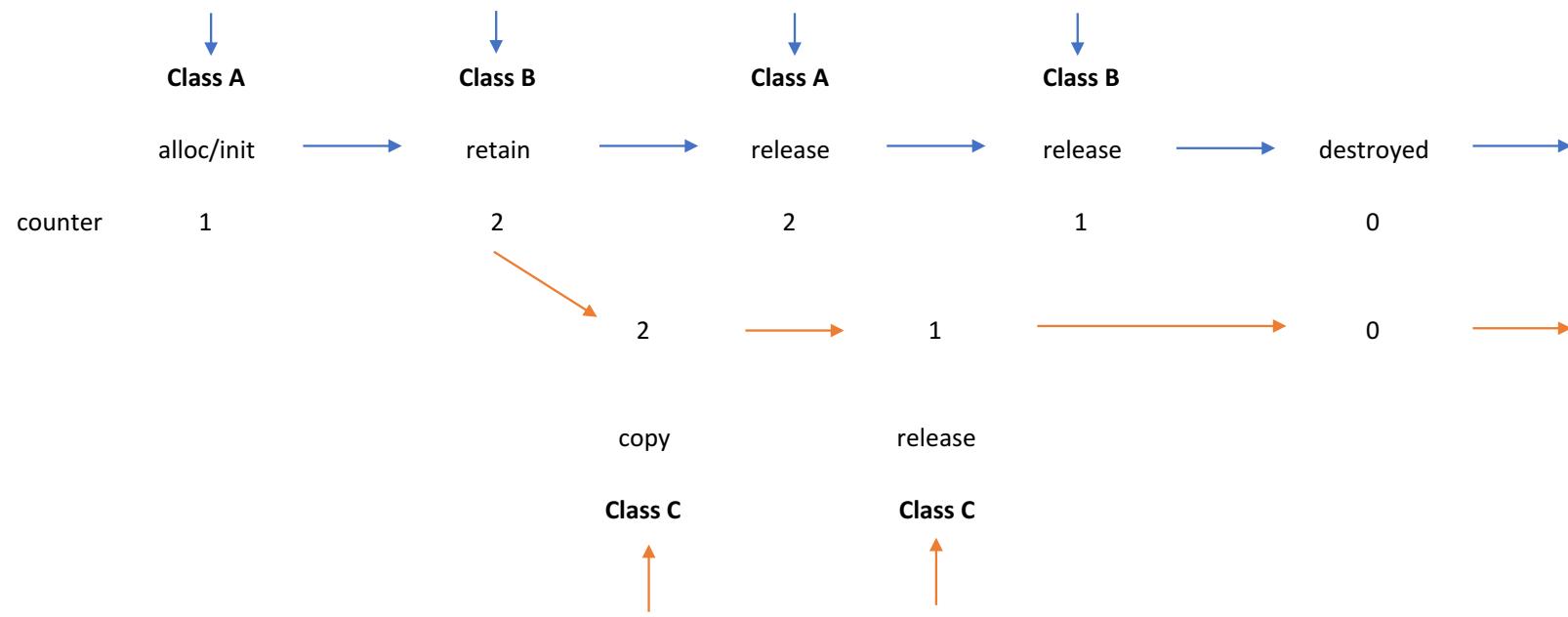
очущує пам'ять коли ніхто не використовує об'єкт

Підрахунок посилань (Reference counting)

«referenceCounter» - це просто змінна в яку записуються к-ть посилань

Action	Change	RefCount
Я хочу дивитися телевізор, я створюю телевізор	1	
Мій товариш хоче дивитися телевізор - він приєднується до мене	1	2
Мені набрид телевізор, я перестав дивитися	-1	1
Кіно завершилося, телевізор нікому не потрібен	-1	0

Retain counter



Проблеми управління пам'яттю

Memory Leak

«danglingPointer»



Memory Leak

“is when your program loses track of a piece of memory that was allocated and has forgotten to release it.” (Objective C Memory Management Essentials)

asciiville by J.T.Presta



“Hey! Your application has a memory leak.”

Memory Organization

Stack

Heap



Memory Organization

Stack

Heap

Stack

Static in memory and allocation happens
only during compile time.

<Struct>

<Bool>

<Int>

Memory Organization

Stack

More efficient when allocating and deallocating data

Heap

Stacks store value types, such as structs and enums

Data stored in the stack is only there temporarily

Makes memory lookup and access very fast from how well it is organized.

The most frequently reserved block is the first to be freed

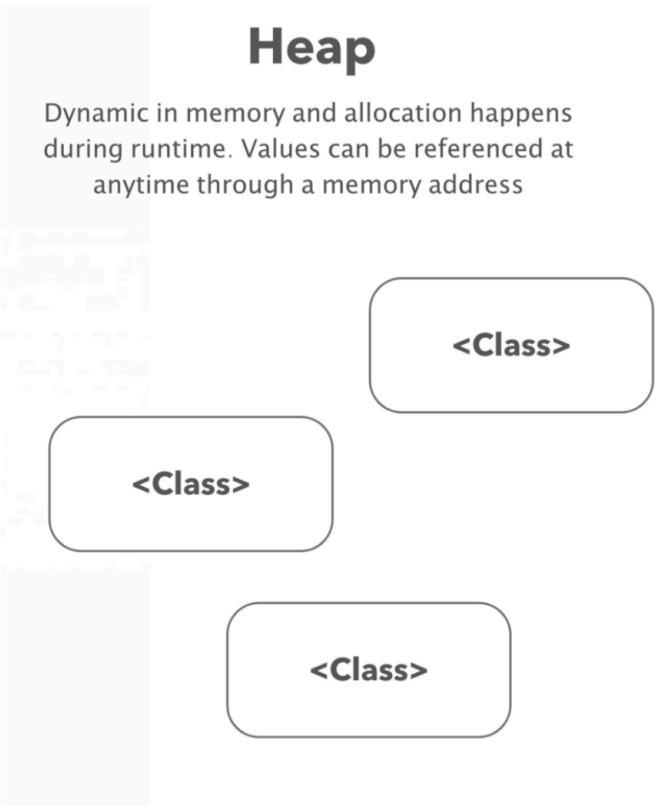
Memory Organization

Stack

Heap

Heap

Dynamic in memory and allocation happens during runtime. Values can be referenced at anytime through a memory address



Memory Organization

Stack

It's more dynamic but less efficient than the stack

Heap

Can grow and shrink in size

Stores reference types such as classes

Goes through 3 steps: *allocation*, *tracking reference counts*, and *deallocation*. As such, this process is less efficient when compared to stacks.

Memory Leak

Increases memory footprint of the app

Unwanted side effects

Crashes

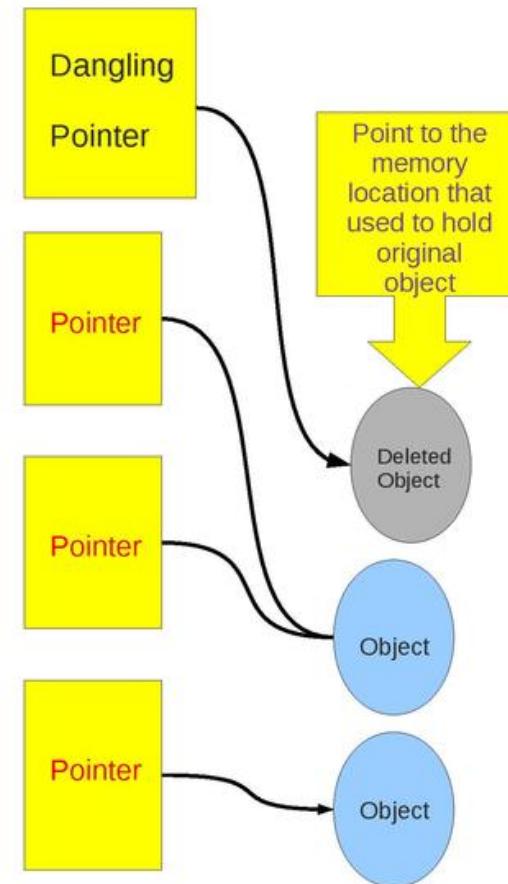


Memory Leak

Action	Change	RefCount
Я хочу дивитися телевізор, я створюю телевізор	1	
Мій товариш хоче дивитися телевізор - він приєднується до мене	1	2
Мій товариш пропав і ніхто про це не дізнався :)	2	
Мені набрид телевізор, я перестав дивитися	-1	1
Так як мій товариш пропав, то посилання на телевізор ще залишилося і він ще працює	1	

DanglingPointer

“is when your pointers that do not point to a valid object of the appropriate type” (wiki)



DanglingPointer

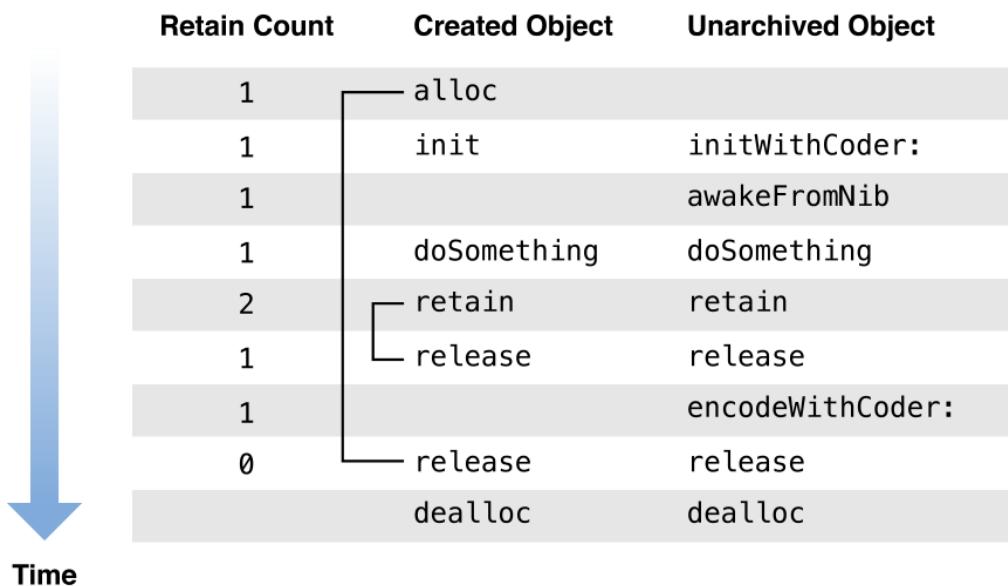
Action	Change	RefCount
Я хочу дивитися телевізор, я створюю телевізор	1	
Мій товариш хоче дивитися телевізор - він приєднується до мене, але нікому про це не говорить	1	
Мені набрид телевізор, я перестав дивитися	-1	0
Телевізора вже не існує	0	
Так як мій товариш ще хоче переглядати кіно, але посилання на телевізор вже знищилося то він використовує мертвє посилання	0	



Життєвий цикл об'єкту

“its runtime life from its creation to its destruction»

<Apple>



Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Value vs Reference Type

Value Types

Reference Types



Value vs Reference Type

Value Types

Reference Types

Кожен об'єкт має унікальну адресу і копіює всі дані

- структури (масиви та словники також)
- перерахування (enums)
- об'єднання (tuples)
- базові типи даних (примітиви) - bool, int, float etc

Value vs Reference Type

Value Types

Reference Types

Кожен об'єкт - має унікальну адресу і копіює всі дані

```
struct MyAwesomeStruct {  
    var someProperty: String?  
}  
  
let tuple: (Int, Float, Bool)?  
  
let isUniqueValueByRef: Bool = true  
  
let array = [1,2,3]  
  
let some = «This is also value type, because String in Swift is struct»
```



Value vs Reference Type

Value Types

Reference Types

Кожен об'єкт - має унікальну адресу і копіює всі дані

```
struct MyAwesomeStruct {  
    var someProperty: String?  
}
```



```
let myStruct = MyAwesomeStruct(someProperty: «strValue»)  
let anotherStruct = myStruct
```

```
myStruct.someProperty = «Updated Value»  
// anotherStruct.someProperty == «strValue» because this is value type and  
// each instance has own memory
```

Value vs Reference Type

Value Types

Reference Types

Коли використовувати

якщо будете порівнювати **дані** об'єкту які можуть змінитися (`==`)

хочете копіювати об'єкти і мати незалежний стан

дані в проекті будуть використані в різних потоках

Value vs Reference Type

Value Types

Reference Types

Кожен об'єкт може використовувати одне й те ж посилання на ячейку пам'яті

класи

інші типи (рідше)



Value vs Reference Type

Value Types

Reference Types

Кожен об'єкт - використовує одне й те ж посилання на ячейку пам'яті

```
class MyValue {  
    var someSubValue: Int?  
}
```

```
let class = MyValue()  
let class2 = class
```

now address of class same as class2

```
class.someSobValue = 1  
// class2.somSubValue == 1 because objects share memory
```



Value vs Reference Type

Value Types

Коли використовувати

Reference Types

якщо будете порівнювати **об'єкти** які можуть змінитися (==)

хочете створити/модифікувати shared state об'єкту

Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Управління пам'яттю в Obj-C

Retain counter

4 Основних правила ручного управління пам'яттю

ARC

NSZombie



MRC

Manual Reference Counting до iOS6

Треба використовувати retain/release

Компілятор не додає retain/release в код автоматично

Developer звільняє об'єкт

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object



4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

OBJ-C

```
NSString* iOwnThis1 = [[NSString alloc] initWithString:@"hello"];
NSString* iOwnThis2 = [someOtherString copy];
NSMutableString* iOwnThis3 = [someOtherString mutableCopy];
NSString* iOwnThis4 = [NSString new];

[iOwnThis1 release];
[iOwnThis2 release];
[iOwnThis3 release];
[iOwnThis4 release];
```

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object



4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object

```
[donkey retain];  
[eagle retain];  
[eagle retain];  
[eagle retain];
```

```
[donkey release];  
[eagle release];  
[eagle release];  
[eagle release];
```

OBJ-C

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object

If you own something - release it

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object

If you own something - release it

```
NSString* iMadeThis = [[NSString alloc] init]; // Rule 1  
[iMadeThis release];  
[imSharingThis retain]; // Rule 2  
[imSharingThis release];  
//can own the same object many times  
Pigeon* pigeon = [[Pigeon alloc] init]; // Rule 1  
[pigeon retain]; // Rule 2  
[pigeon release];  
[pigeon release];
```

OBJ-C

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object

If you own something - release it

If you keep pointer to an object - you MUST own object (with some exceptions)

4 Основних правила ручного управління пам'яттю

If you create an object, using «alloc», «copy», «new» - you own object

If you retain ab object - you own object

If you own something - release it

If you keep pointer to an object - you MUST own object (with some exceptions)

string literals are never deallocated

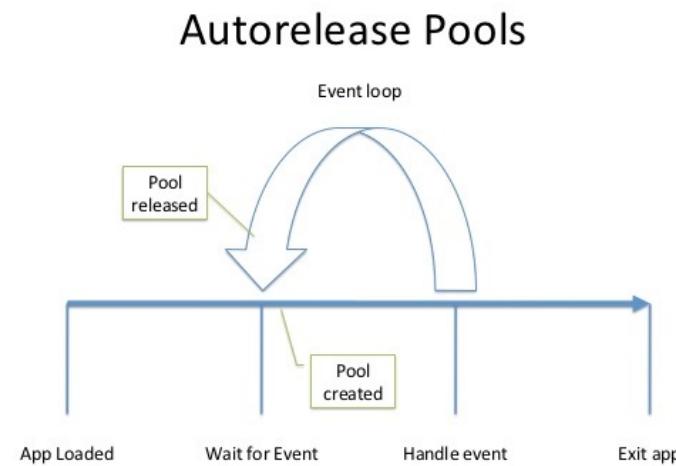
NSAutoreleasePool And autorelease

common mistakes



Autorelease POOL

“a mechanism whereby you can relinquish ownership of an object, but avoid the possibility of it being deallocated immediately”



ARC

Automatic Reference Counting представлене в iOS5

Не треба використовувати retain/release

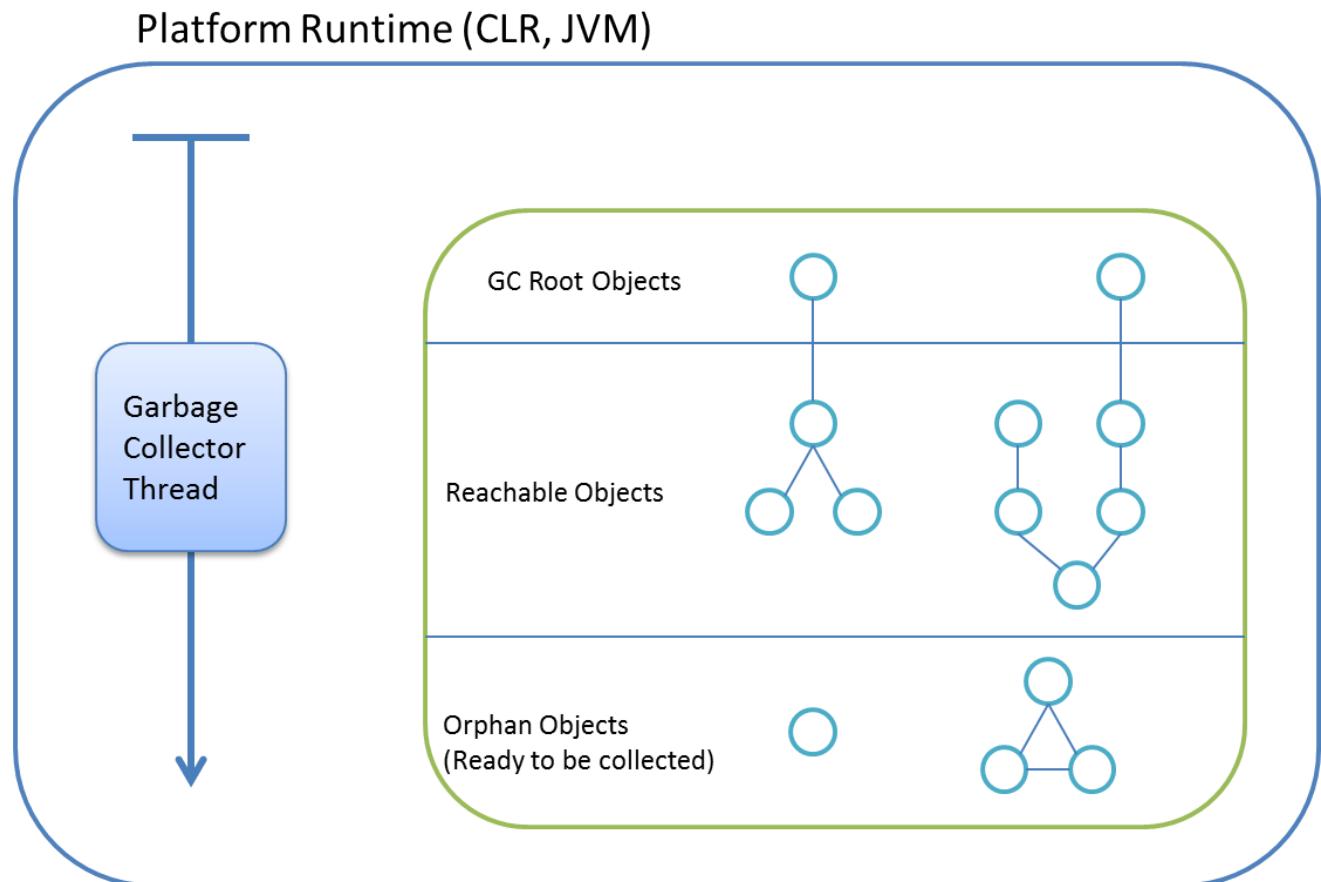
Компілятор додає retain/release в код автоматично

ARC звільняє об'єкт як тільки всі strong посилання зникають

Це не Garbage Collector

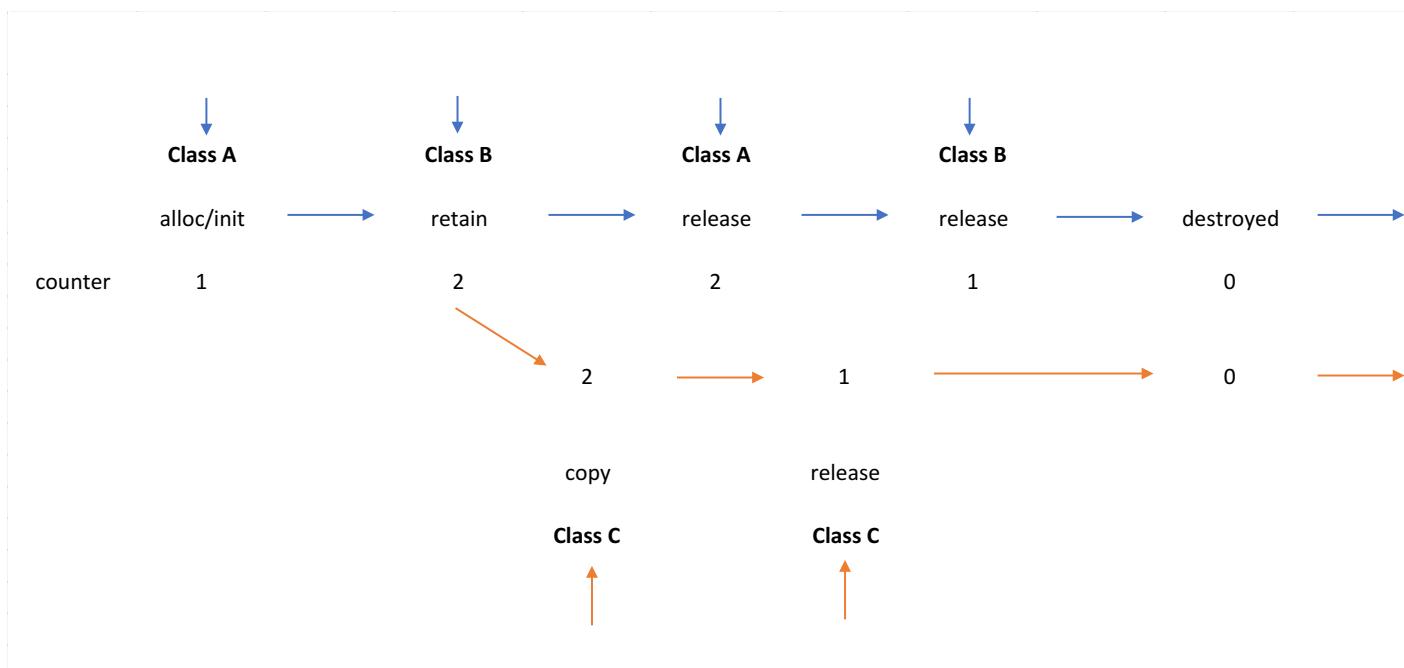
Це не Garbage Collector

Garbage Collector:



Це не Garbage Collector

ARC

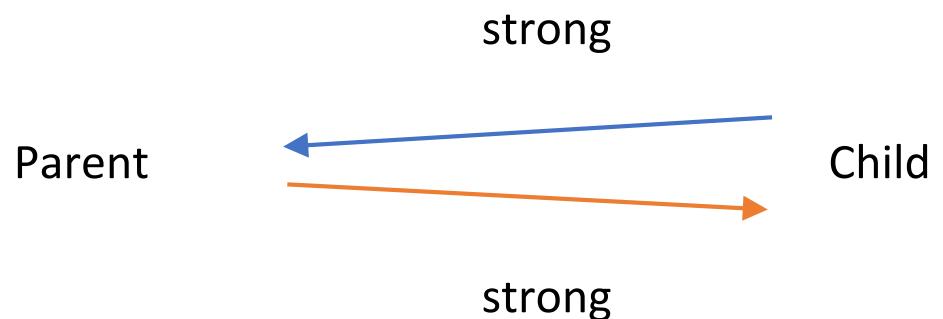


ARC - проблеми

Retain cycles

Retain cycles

коли обєкти тримають посилання один на один



Retain cycles - Правила для уникання

Об'єкт ніколи не має тримати strong посилатися на «батьківський» об'єкт

Object не має тримати strong посилання на будь-який об'єкт який
стоїть вище в ієрархії

"Connection" об'єкти не повинні тримати strong посилання на їх цілі (delegate,
outlets, observers)

Retain cycles - Правила для уникання

strong - protects the referred object from getting deallocated by ARC

weak - don't protects the referred object from getting deallocated by ARC

“Use a **weak** reference whenever it is valid for that reference to become nil at some point during its lifetime. Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization.” (Apple)

Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Управління пам'яттю

Value vs Reference Type

Управління пам'яттю в Obj-C

Управління пам'яттю в Swift



Управління пам'яттю

Obj-C vs Swift memory Management

Swift memory management principles

Strong Reference Cycles

Resolving Strong Reference Cycles

Capture List



Swift memory management

“in Swift, memory management is made to be as painless as possible“

“this mean that memory management just work in Swift “ (Apple)

Obj-C vs Swift memory Management

is similar, but Swift use ARC

Swift is strongly typed and type safe, so all variables must have a known type and non-nil value (unless declared optional)



How ARC Work in Swift

Create object - ARC allocate chunk of memory with additional information about object

Deallocate object - ARC frees memory

Tell what type of relationship between your classes

Access to deallocated obj crash app

Deallocated obj has referenceCount equal to 0



ARC - проблеми

Retain cycles



Strong reference cycle

when instance of a class never get reference count equal to 0



Resolving Strong reference cycle

*“Use a **weak** reference whenever it is valid for that reference to become nil at some point during its lifetime. Conversely, use an **unowned** reference when you know that the reference will never be nil once it has been set during initialization.” (Apple)*

Weak reference

“a reference that does not keep a strong hold on an object and so does not stop ARC from disposing” (Apple)

weak variable can be optional

Indicate as:

```
weak var myVariable: SomeObject?
```



Unowned reference

“a reference that does not keep a strong hold on an object and so does not stop ARC from disposing, but it assumed to always have a value” (Apple)

unowned variable should always have value

Indicate as:

```
unowned var person: Person
```



weak vs unowned

« Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.»

«Define a capture as a weak reference when the captured reference may become nil at some point in the future.»

weak vs unowned vs strong

	var	let	optional	non-optional
strong	x	x	x	x
weak	x		x	
unowned	x	x		x

Strong reference cycles for closures

“Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.” (Apple)

```
closure = {  
    self.string = "Hello, World! I'm immortal string and cause memory leak"  
}
```



```
closure = { [unowned self] in  
    self.string = "Hello, World! I will die soon :("
```

Capture list

“Each item in a capture list is a pairing of the weak or unowned keyword with a reference to a class instance (such as self) or a variable initialized with some value (such as delegate = self.delegate!)” (Apple)



```
lazy var someClosure: ( Int, String ) -> String = {  
    [unowned self, weak delegate = self.delegate!] (index: Int , stringToProcess: String) -> String in  
        // closure body goes here - do some action here  
}
```

Debug toolset

Instruments

LLVM and Clang

Xcode tools



Debug toolset

Instruments

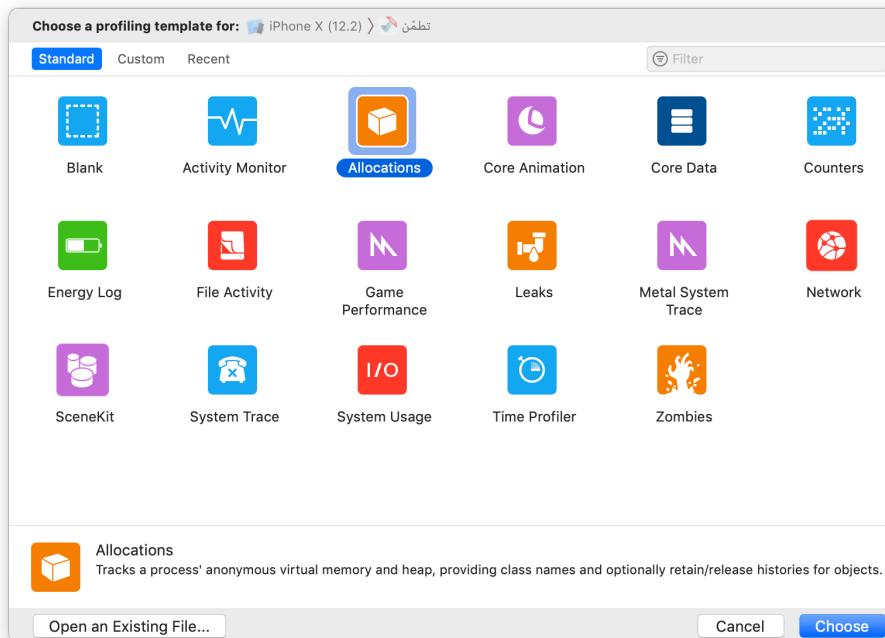
LLVM and Clang

Xcode tools



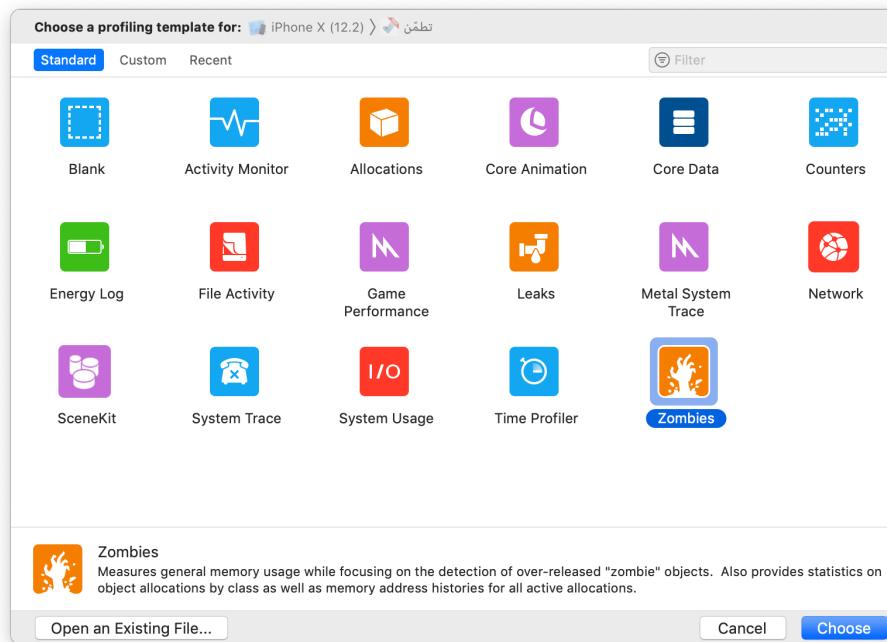
Allocations

Track memory allocations



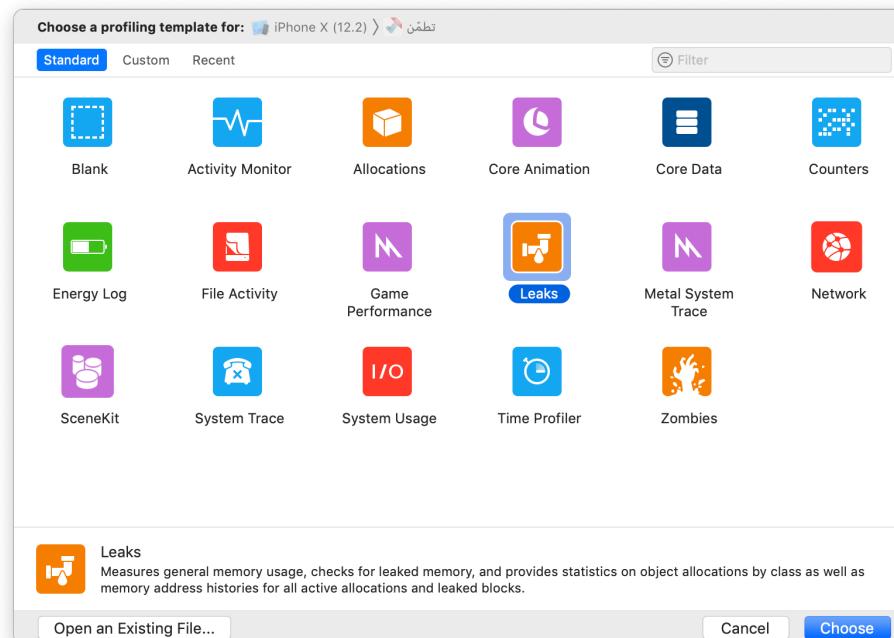
Zombies

“is a memory debugging aid which can help you debug subtle over-release/ autorelease problems.”



Leaks

Memory usage, leaked memory



Debug toolset

Instruments

LLVM and Clang

Xcode tools



Debug toolset

Instruments

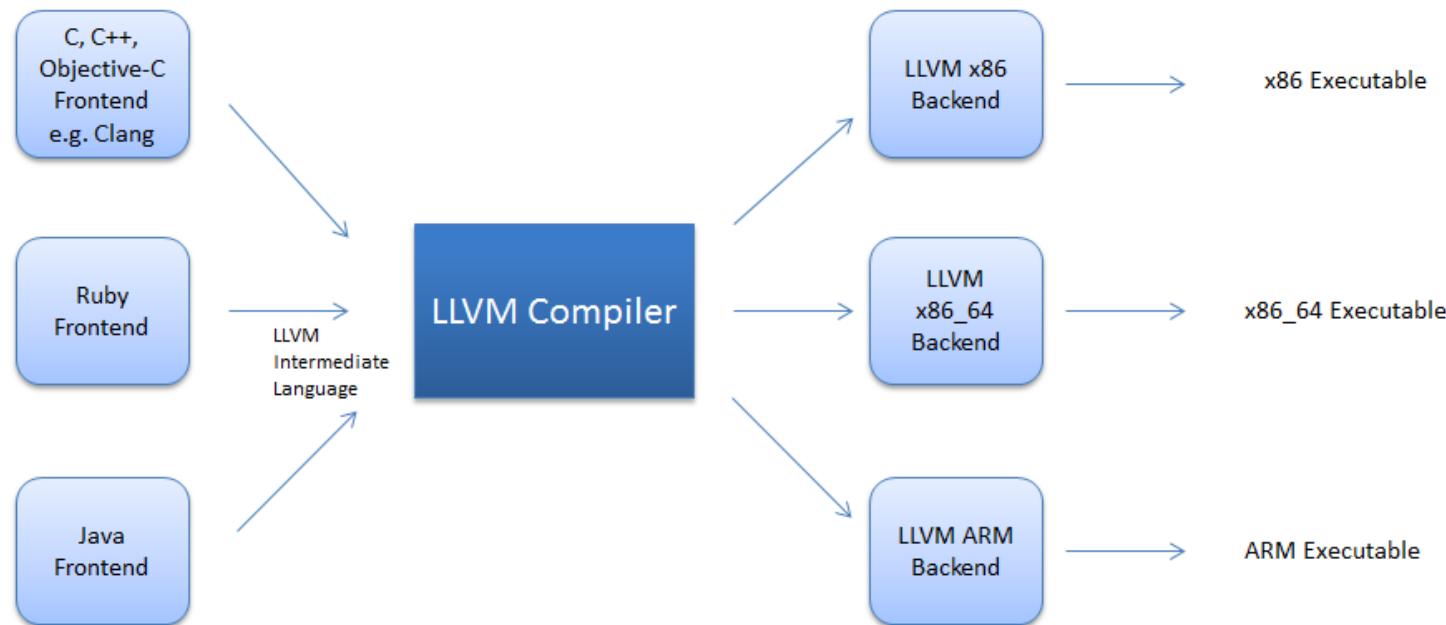
LLVM and Clang

Xcode tools



LLVM and Clang

“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies”
(Apple)



LLVM and Clang

“Clang is where the ARC magic happens”
(Apple)



Debug toolset

Instruments

LLVM and Clang

Xcode tools



Debug toolset

Instruments

LLVM and Clang

Xcode tools



xCode tools

scheme debug option

analyze tool

memory graph tool



xCode tools

scheme debug option
analyze tool
memory graph tool

Memory Management Malloc Scribble
 Malloc Guard Edges
 Guard Malloc
 Zombie Objects
 Memory Graph on Resource Exception

Logging Malloc Stack
 Dynamic Linker API Usage
 Dynamic Library Loads



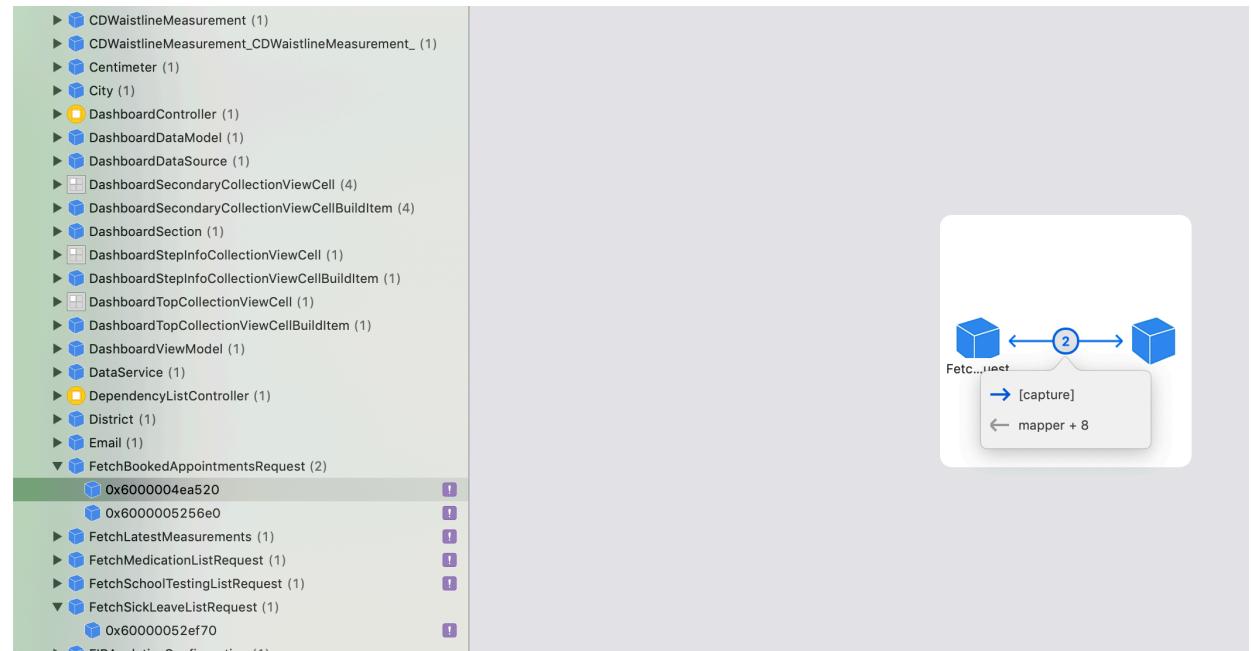
xCode tools

scheme debug option
analyze tool
memory graph tool

```
73
74 - (void)viewDidLoad
75 {
76     [super viewDidLoad];
77
78     [self.navigationBar applyBrandAppearance];
79
80     self.aroundMeButton = [[[UIBarButtonItem alloc] initWithTitle:@"OEMAroundTitle"
81                                                               style:UIBarButtonItemStyleBordered
82                                                               target:self
83                                                               action:@selector(launchAroundMe:)] autorelease];
84
85     self.fromAddressButton = [[[UIBarButtonItem alloc] initWithTitle:@"OEMFromAddress"
86                                                               style:UIBarButtonItemStyleBordered
87                                                               target:self
88                                                               action:@selector(launchAddressSearch:)] autorelease];
89
90
91     if (!IOS7_USER_INTERFACE_IDIOM()) {
92
93         UIBarButtonItem *fixedSpace = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFixedSpace
94                                         target:nil
95                                         action:nil];
96
97         fixedSpace.width = 25;
98
99         [self.navigationBar.topItem setLeftBarButtonItems:@[self.aroundMeButton, fixedSpace, self.fromAddressButton]];
100    } else {
101        [self.navigationBar.topItem setLeftBarButtonItems:@[self.aroundMeButton, self.fromAddressButton]];
102    }
103 }
```

xCode tools

scheme debug option
analyze tool
memory graph tool



Список корисних ресурсів

About memory management (URL).

Мэтт Гэлловей - Сила Objective-C 2.0 - 2014. (Chapter 5).

Objective-C Memory Management Essentials By Gibson Tang, Maxim Vasilkov
(978-1-84969-712-5).

А. Махер - "Программирование для iPhone. Высший уровень" (раздел 1.3)

Список корисних ресурсів

Useful explanation week vs unowned (URL).

Objective-C Memory Management Essentials By Gibson Tang, Maxim Vasilkov
(978-1-84969-712-5).

The Swift Programming Language - 2015, Apple Inc,
Chapter Automatic Reference Counting

Weak vs Unowned (URL)



SIGMA
Software

UNIVERSITY