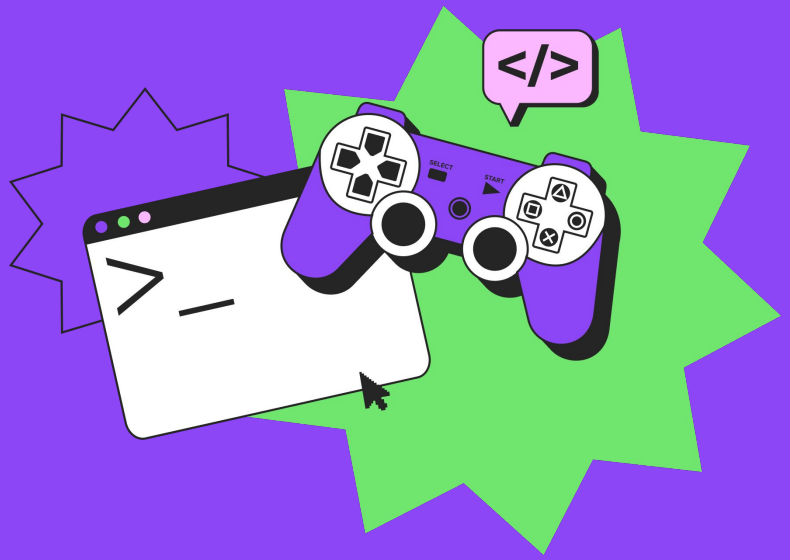
GeekBrains

Работа с памятью в Objective – C, ARC и MRC

Урок 3

Знакомство с языком Objective-C. Изучение его предшественников. Отличия от других языков. Основные типы данных и арифметические операции. Обзор среды разработки Xcode. Организация файлов.



План курса

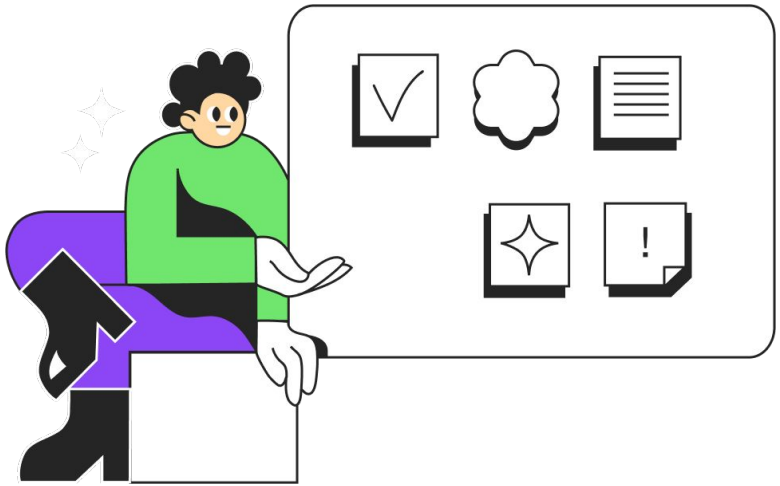
1. Basic knowledge of Objective – C
2. ООП + POP
3. MRC + ARC
4. PThread, GCD, Operation
5. URL/URLSession
6. Objective–C Runtime

Что будет на уроке сегодня

- 📌 Вспомним области памяти в Objective – C / Swift
- 📌 Вспомним разницу Value vs Reference type
- 📌 MRC
- 📌 ARC
- 📌 Side Table
- 📌 Property Аттрибуты

Области памяти

Память необходима для того, чтобы хранить данные,
с которыми работает программа во время своего выполнения.



Для чего используется оперативная память?

Области памяти

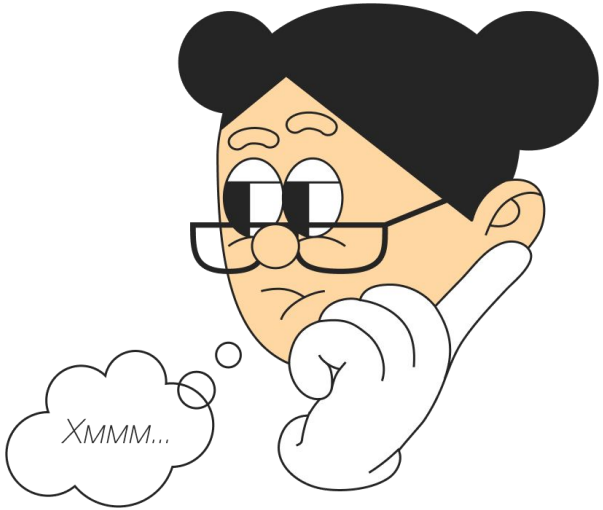
Когда программа выполняется в операционный системе компьютера,
она нуждается в доступе к оперативной памяти (RAM) для того, чтобы:

1. Загружать свой собственный байт-код для выполнения;
2. Хранить значения переменных и структуры данных, которые используются в процессе работы;
3. Загружать внешние модули, которые необходимы программе для выполнения задач.

💡 Помимо места, используемого для загрузки своего собственного байт-кода, программа использует при работе две области в оперативной памяти — стек (stack) и кучу (heap).

Области памяти

1. Stack
2. Heap
3. Global



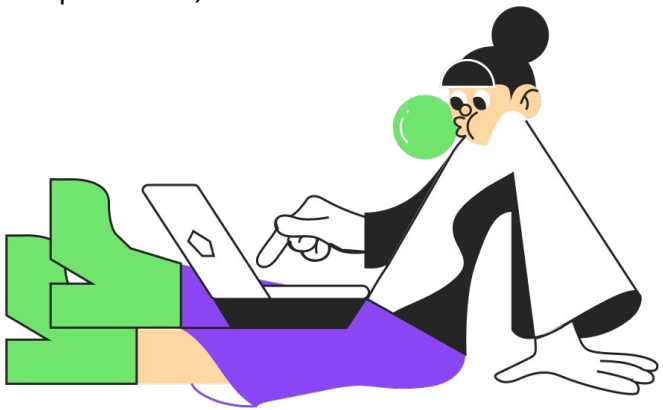
Stack

Stack — переменные, выделенные в стеке, хранятся непосредственно в памяти, и доступ к этой памяти очень быстрый, и ее выделение определяется при компиляции программы.

Stack

1. **Стек используется для статичного выделения памяти.** Он организован по принципу «последним пришёл — первым вышел» (LIFO). Можно представить стек как стопку книг — разрешено взаимодействовать только с самой верхней книгой: прочитать её или положить на неё новую.

2. **Стек позволяет очень быстро выполнять операции с данными — все манипуляции производятся с «верхней книгой в стопке».** Книга добавляется в самый верх, если нужно сохранить данные, либо берётся сверху, если данные требуется прочитать;

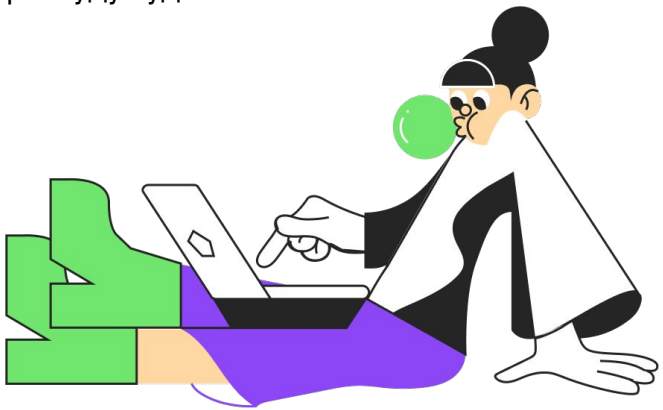


Stack

3. Существует ограничение в том, что данные, которые предполагается хранить в стеке, обязаны быть конечными и статичными — их размер должен быть известен ещё на этапе компиляции;

4. Каждый поток многопоточного приложения имеет доступ к своему собственному стеку;

5. Когда функция вызывается, все локальные экземпляры этой функции будут помещены в текущий стек. И как только функция вернется, все экземпляры будут удалены из стека.

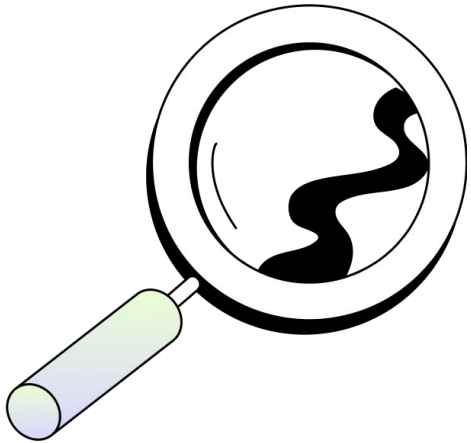


```c
int main(int argc, const char * argv[]) {

    if (1 != 2)
    {
        int a = 0;
        printf("%d", a);
    }
    /**
     has no access to `a`,
not in the same scope
     */
    // printf("%d", a);

    return 0;
}
```

Stack variable

Heap

Heap — куча используется для динамического выделения памяти, однако, в отличие от стека, данные в куче первым делом требуется найти с помощью «оглавления». Можно представить, что куча это такая большая многоуровневая библиотека, в которой, следуя определённым инструкциям, можно найти необходимую книгу.



Heap

1. Операции на куче производятся несколько медленнее, чем на стеке, так как требуют дополнительного этапа для поиска данных;

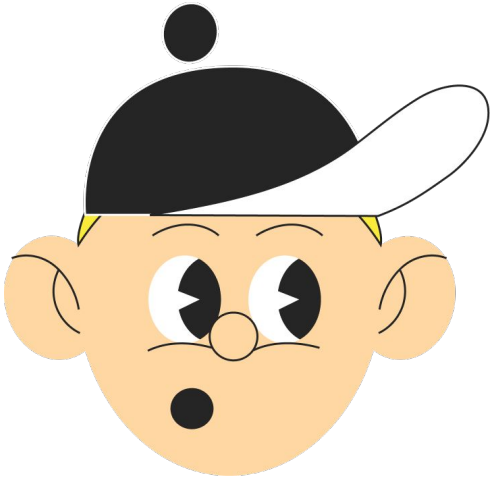
2. В куче хранятся данные динамических размеров, например, список, в который можно добавлять произвольное количество элементов;

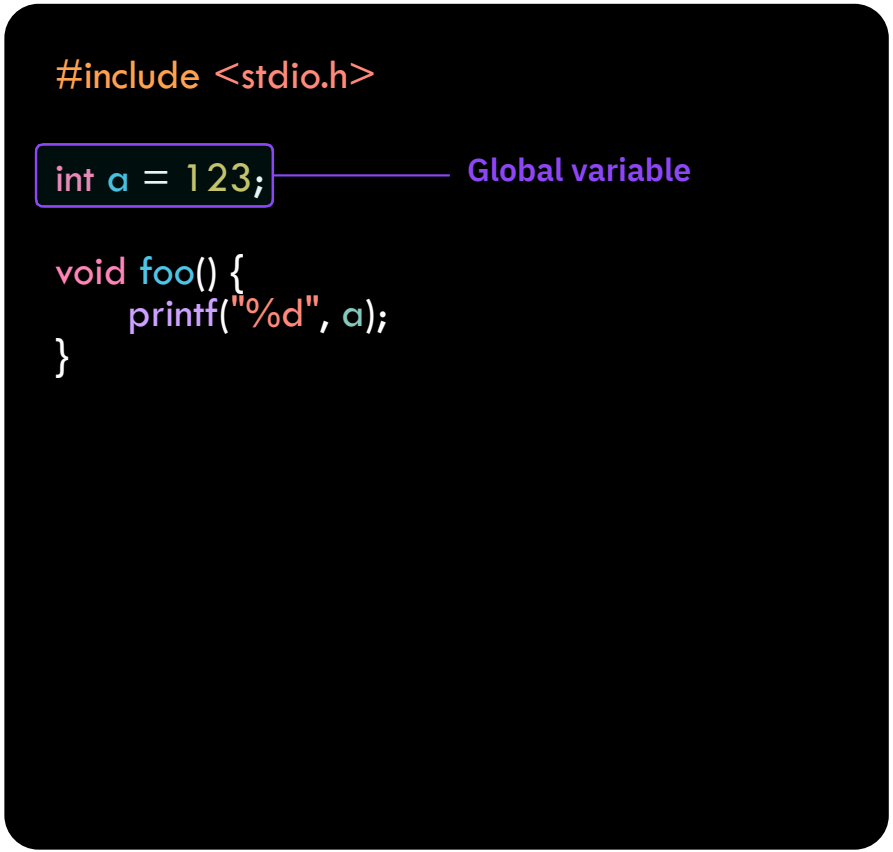
3. Куча общая для всех потоков приложения;

Global

Глобальная переменная в программировании — переменная, областью видимости которой является вся программа, если только она не перекрыта.

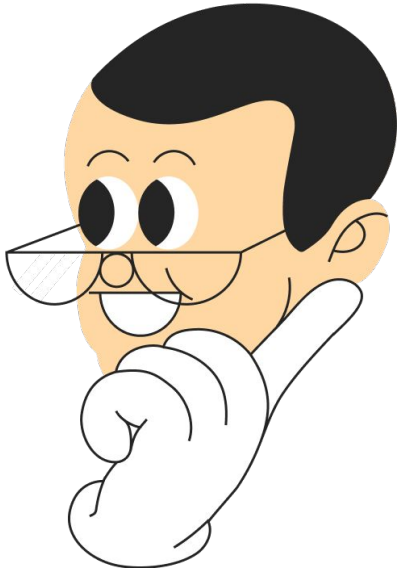
Механизмы взаимодействия с глобальными переменными называют механизмами доступа к глобальному окружению или состоянию.




#include <stdio.h>
int a = 123;
Global variable
void foo() {
printf("%d", a);
}

Типы значений и ссылок

Каждый тип можно рассматривать как тип значения или как ссылочный тип.



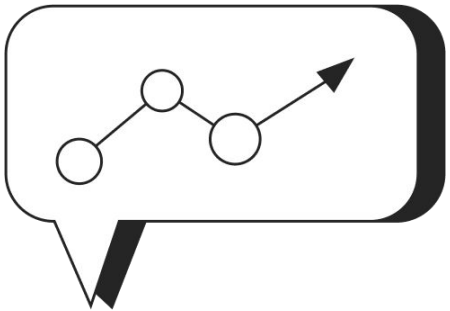
Value type

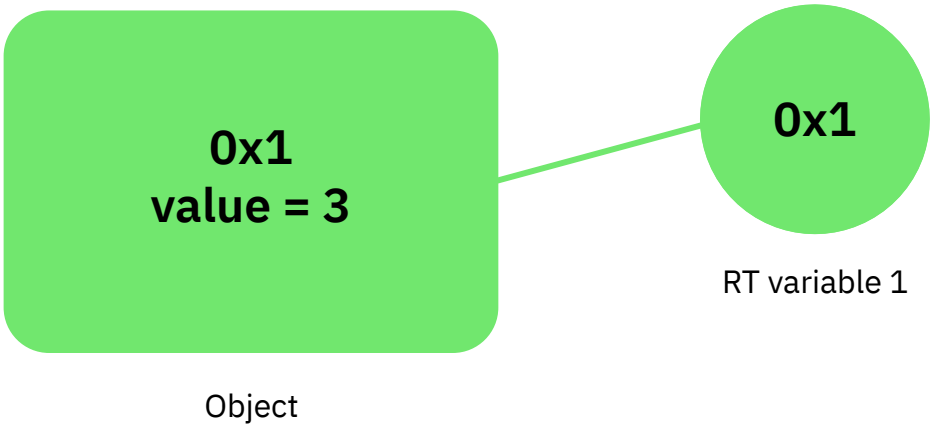
Value Type — каждая переменная типа значения имеет свою собственную копию данных, и операции с одной не влияют на другую. За него отвечает стэк.

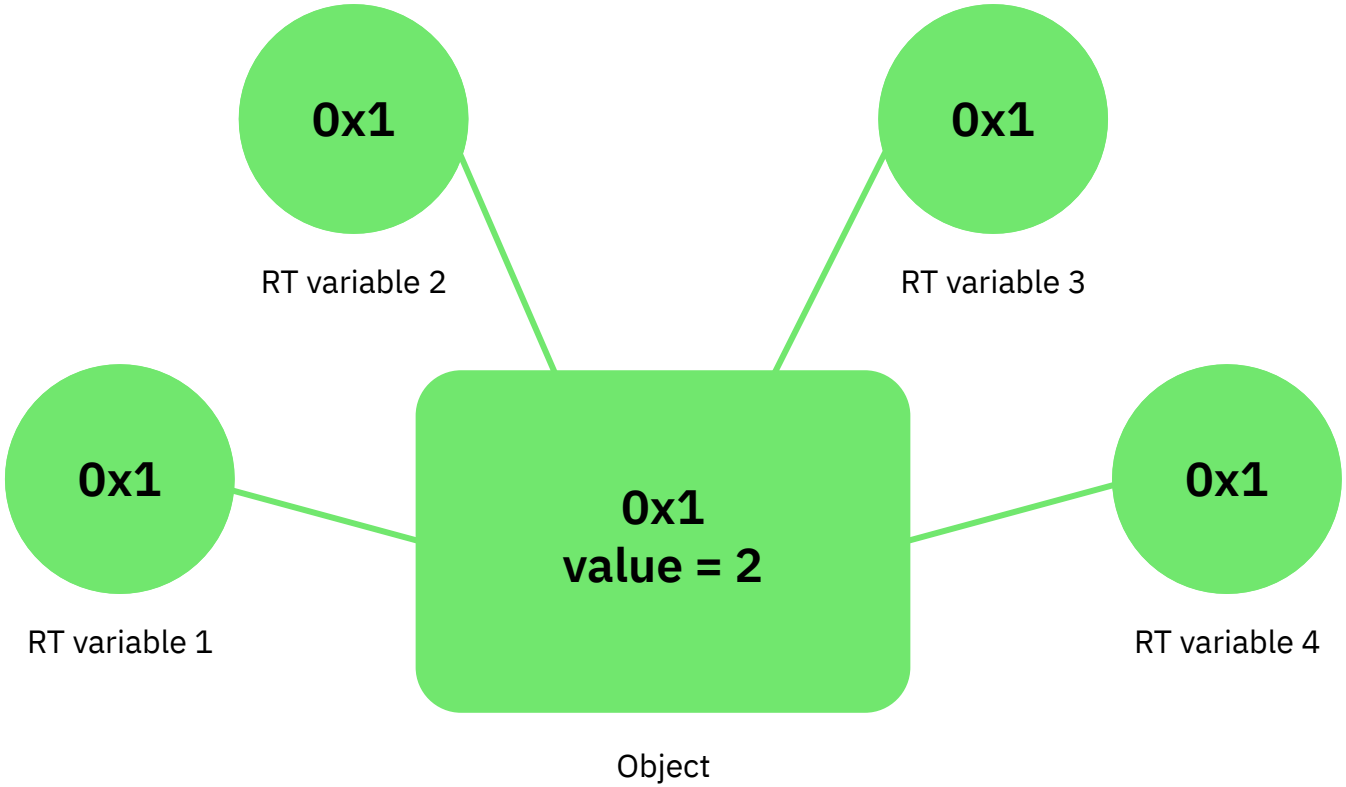
Reference type

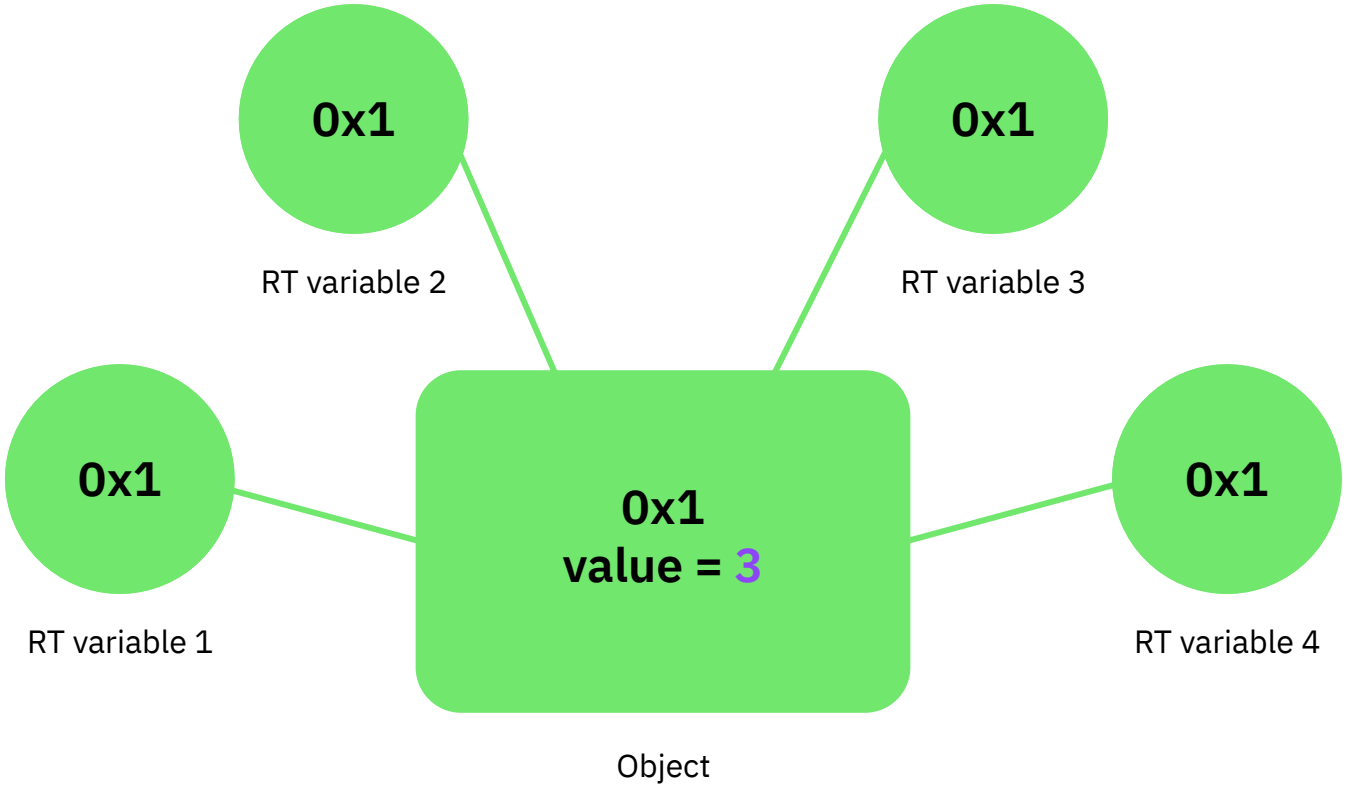
Reference Type — у нас есть ссылка, указывающая на это место в памяти.

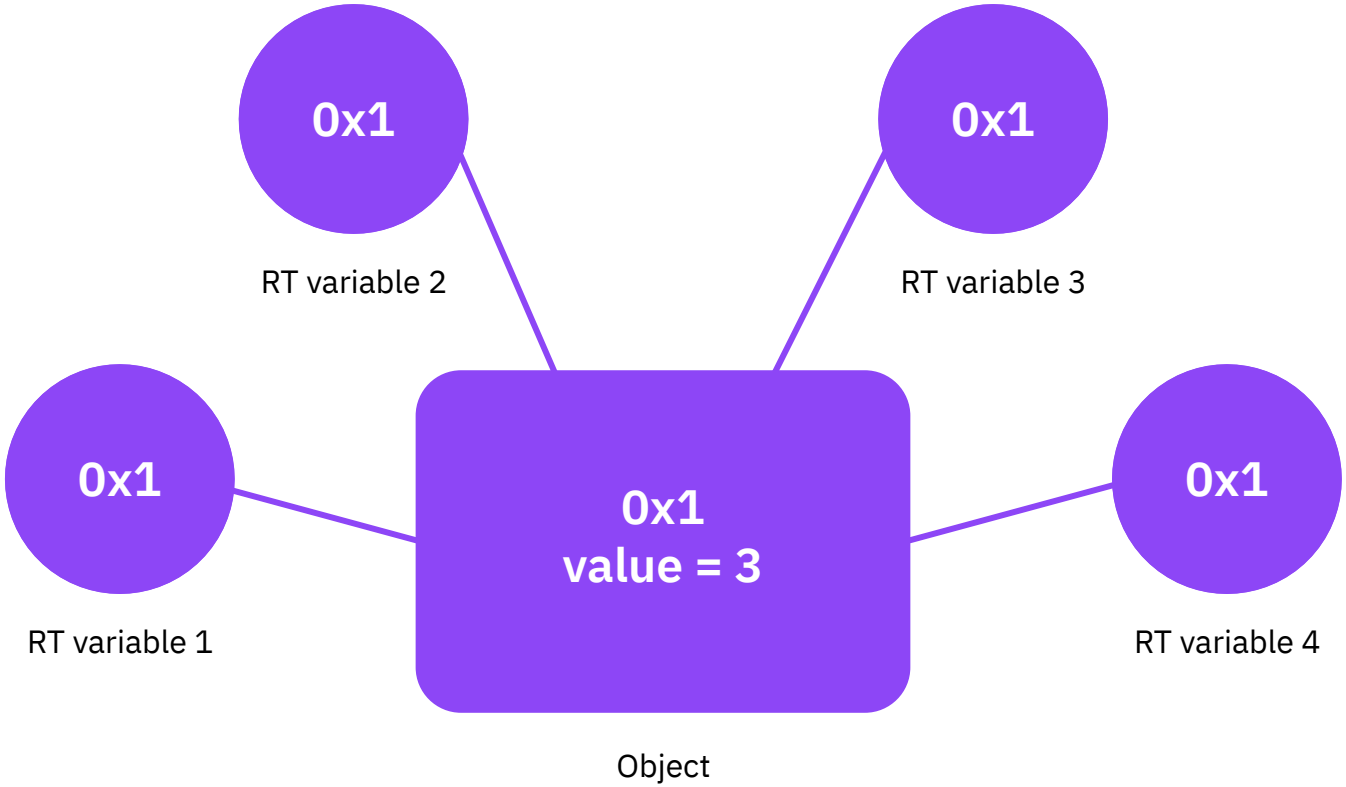
Переменные ссылочного типа могут указывать на одни и те же данные; следовательно, операции с одной переменной могут повлиять на данные, указанные другой переменной. За него отвечает куча.




0x1
value = 3
0x1
RT variable 1
Object


0x1
RT variable 2
0x1
RT variable 3
0x1
RT variable 1
0x1
value = 2
0x1
RT variable 4
Object


0x1
RT variable 2
0x1
RT variable 3
0x1
RT variable 1
0x1
value = 3
0x1
RT variable 4
Object


0x1
RT variable 2
0x1
RT variable 3
0x1
RT variable 1
0x1
value = 3
Object
0x1
RT variable 4

MRC & ARC

Модель памяти Objective-C основана на подсчете ссылок.

1. MRC — ручной подсчет ссылок
2. ARC — автоматический подсчет ссылок

MRC – manual reference counter

MRC – это ручное управление ссылками через код. В самом начале и в давние времена разработчики сами управляли подсчетом ссылок через команды.

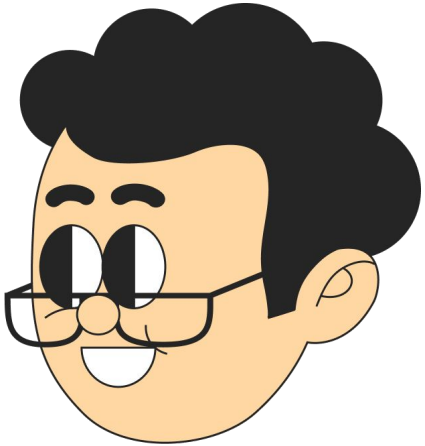
1. alloc – создание объекта (создаем ссылку)
2. retain – обращение к нему (+1 к ссылке)
3. release – уменьшаем счетчик ссылок (-1)
4. dealloc – если счетчик ссылок равен 0 = выгрузка из памяти

По сути, вы выделяете объект, сохраняете его в какой-то момент, а затем отправляете один выпуск для каждого отправленного вами выделения/сохранения. Метод dealloc вызывается для объекта, когда он удаляется из памяти.

MRC – manual reference counter

По сути, вы выделяете объект, сохраняете его в какой-то момент,
а затем отправляете один выпуск для каждого отправленного вами выделения /
сохранения. Метод dealloc вызывается для объекта, когда он удаляется из памяти.

Проблемы:
1. Нужно постоянно считать retain, release
2. Крэш при обращении из выгруженного из памяти
3. Забыли поставить релиз – утечка памяти



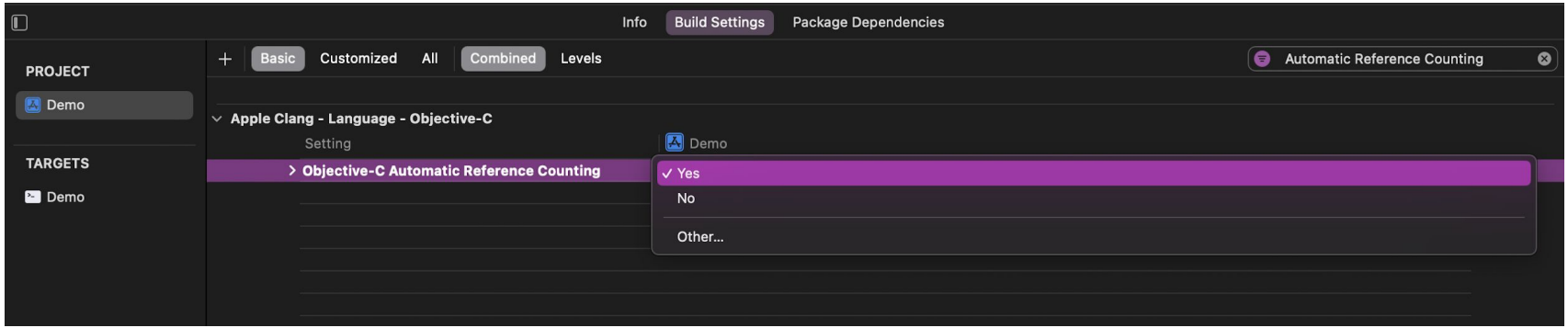
MRC – manual reference counter

```objc
int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Object *object = [[Object alloc] init];

        [object retain];    // count - 2
        NSLog(@"count - 2");

        [object retain];    // count - 3
        NSLog(@"count - 3");

        [object release]; // count - 2
        NSLog(@"count - 2");

        [object release]; // count - 1
        NSLog(@"count - 1");

        [object release]; // count - 0; Вызывается dealloc

    }
    return 0;
}

```

MRC – manual reference counter

```objc
@interface Person : NSObject
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (assign, readonly) NSString *fullName;
@end

@implementation Person
// ...
- (void)dealloc
    [_firstName release];
    [_lastName release];
    [super dealloc];
}
@end

```

ARC – automatic reference counter



ARC – automatic reference counter

После того, как программисты поняли, что можно придумать механизм, который сам за программиста считает ссылки — мир в iOS поменялся. Больше не нужно было считать ссылки и следить за ними. За нас это делает ARC автоматически. Он сам понимает куда и зачем что вставлять и когда удалять.

Что изменилось?

1. (release/retain — нельзя вызывать) dealloc — работает частично
2. properties change — weak/strong

У property появились модификаторы:

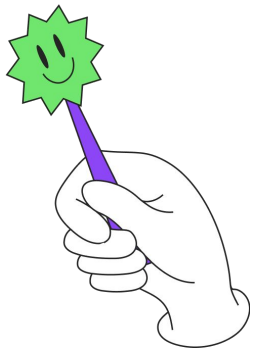
1. strong — аналог retain
2. weak — аналог assign. В @property при освобождении ставится nil и не крэшит приложение при обращении

ARC – automatic reference counter

Но есть и минусы, с которыми не справляется ARC:

Retain cycle — это когда объем выделенного пространства в памяти не может быть освобожден из-за циклов сохранения. Поскольку Objective – C / Swift использует автоматический подсчет ссылок (ARC), цикл сохранения происходит, когда два или более объекта содержат сильные ссылки друг на друга.

В результате эти объекты сохраняют друг друга в памяти, потому что их счетчик сохранения никогда не уменьшится до 0, что предотвратит вызов функции deinit и освобождение памяти



💡 Решение банальное — сделать одну из ссылок слабой.

ARC vs MRC

Code compiled without ARC (`-fno-objc-arc`)

```
NSScanner *scanner = [[NSScanner alloc] initWithString:string];
[scanner scanInt:&value1];
[scanner release]; // bad: premature release of object
...
[scanner scanInt:&value2]; // undefined behavior; potential crash
```

Code compiled with ARC (`-f-objc-arc`)

```
NSScanner *scanner = [[NSScanner alloc] initWithString:string];
[scanner scanInt:&value1];
...
// ARC guarantees the object is retained for as long as it's used
[scanner scanInt:&value2];
```

ARC & Retain Cycle

```objc
@class Department;

@interface Person:NSObject
@property (strong,nonatomic)Department * department;
@end

@implementation Person
-(void)dealloc{
    NSLog(@"dealloc person");
}

@end
@interface Department: NSObject
@property (strong,nonatomic)Person * person;
@end

@implementation Department
-(void)dealloc{
    NSLog(@"dealloc Department");
}
@end
```

ARC & Retain Cycle

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Person * person = [[Person alloc] init];
    Department * department = [[Department alloc] init];
    person.department = department;
    department.person = person;
}
```

ARC & Retain Cycle + weak

```objc
@class Department;

@interface Person:NSObject
@property (weak,nonatomic)Department * department;
@end

@implementation Person
-(void)dealloc{
    NSLog(@"dealloc person");
}

@end
@interface Department: NSObject
@property (strong,nonatomic)Person * person;
@end

@implementation Department
-(void)dealloc{
    NSLog(@"dealloc Department");
}
@end
```

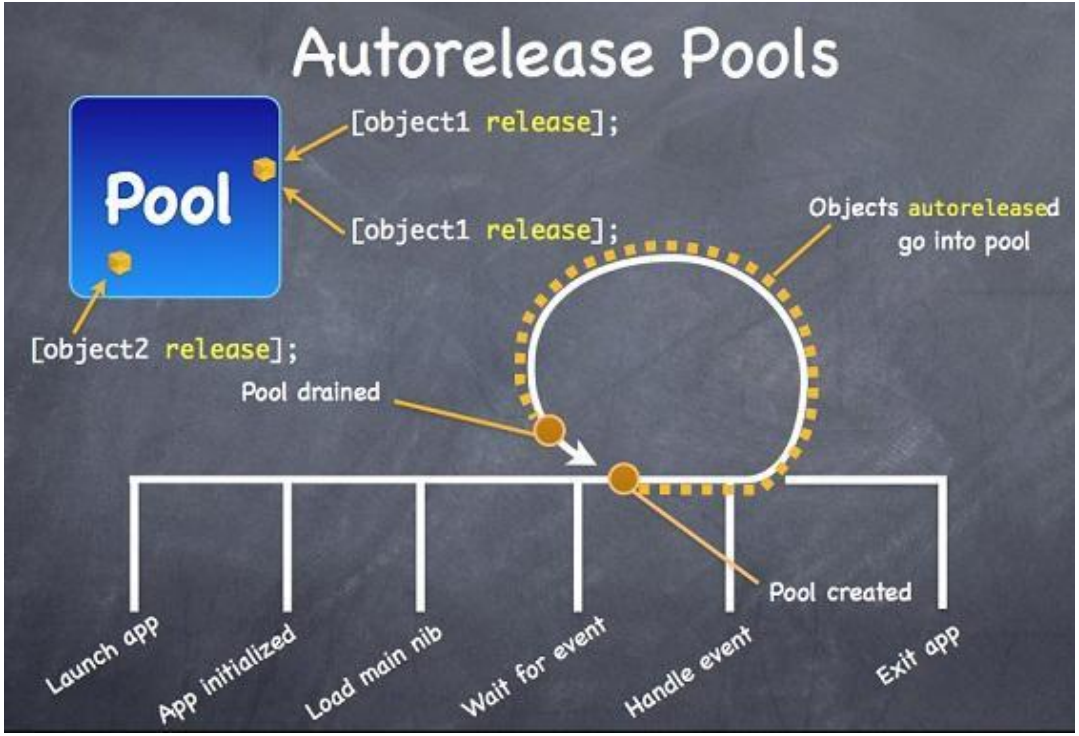
Autoreleasepool

Пул авто освобождения хранит объекты, которым отправляется сообщение об освобождении, когда сам пул опустошается.

В среде с подсчетом ссылок (в отличие от среды, в которой используется сборка мусора) объект NSAutoreleasePool содержит объекты, получившие сообщение об автоматическом освобождении, и при очистке отправляет сообщение об освобождении каждому из этих объектов.

Таким образом, отправка автоосвобождения вместо освобождения объекту продлевает время жизни этого объекта, по крайней мере, до тех пор, пока сам пул не будет опустошен (это может быть дольше, если объект впоследствии сохраняется). Объект может быть помещен в один и тот же пул несколько раз, и в этом случае он получает сообщение об освобождении каждый раз, когда он был помещен в пул.

Autoreleasepool



Important

If you use Automatic Reference Counting (ARC), you cannot use autorelease pools directly. Instead, you use `@autoreleasepool` blocks. For example, in place of:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
// Code benefitting from a local autorelease pool.
[pool release];
```

you would write:

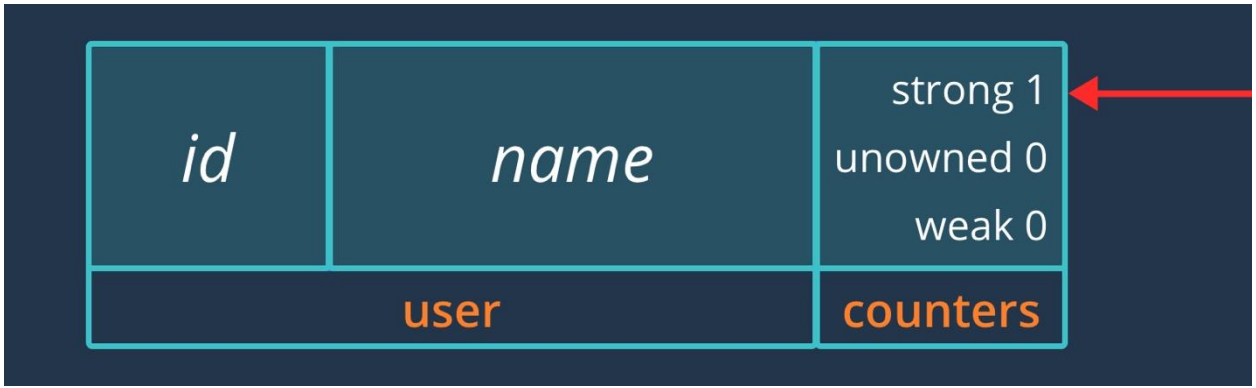
```
@autoreleasepool {
    // Code benefitting from a local autorelease pool.
}
```

`@autoreleasepool` blocks are more efficient than using an instance of `NSAutoreleasePool` directly; you can also use them even if you do not use ARC.

Side Table only (Swift)

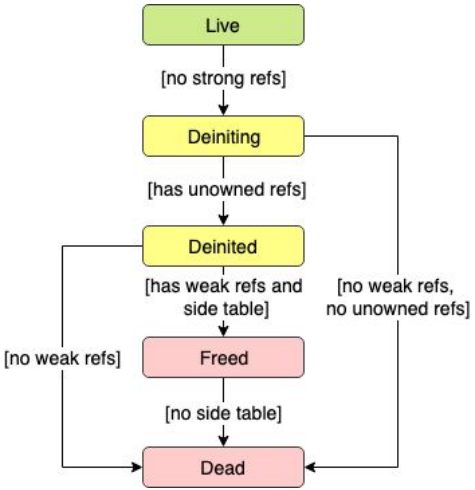
Side tables — это механизм для реализации слабых ссылок Swift.

Обычно объекты не имеют слабых ссылок, поэтому резервировать место для подсчета слабых ссылок в каждом объекте нецелесообразно. Эта информация хранится извне в дополнительных таблицах, поэтому ее можно выделить только тогда, когда это действительно необходимо.



Side Table only (Swift)

```
class HeapObjectSideTableEntry {
    std::atomic<HeapObject*> object;
    SideTableRefCounts refCounts; // Operations to increment and decrement
reference counts
}
```



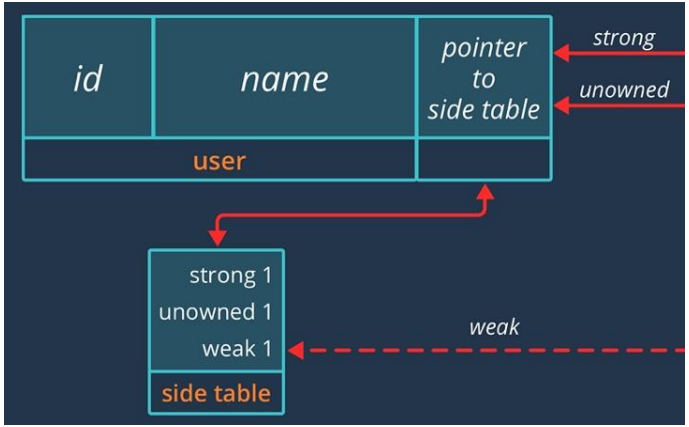
Side Table only (Swift)

**Объект изначально не имеет side table,
и он создается автоматически, когда:**

1. На объект указывает первая слабая ссылка.
2. Переполнение счетчика strong или unowned ссылок.

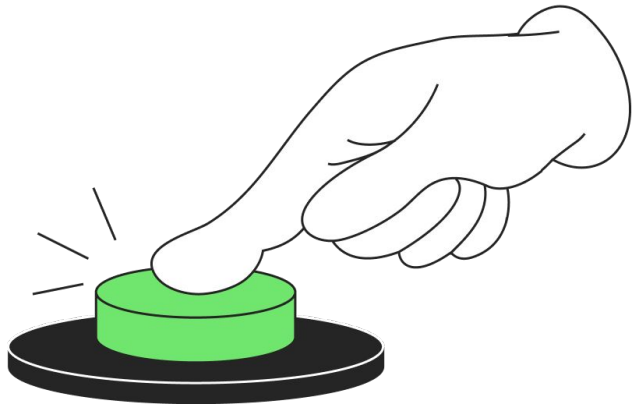
И объект, и боковая таблица имеют указатель друг на друга.

Получение side table — это односторонняя операция.
Еще одна вещь, на которую следует обратить внимание, это то, что слабые ссылки теперь указывают непосредственно на боковую таблицу, в то время как сильные и бесхозные ссылки по-прежнему указывают непосредственно на объект. Это позволяет полностью освободить память объекта.



Атрибуты свойств

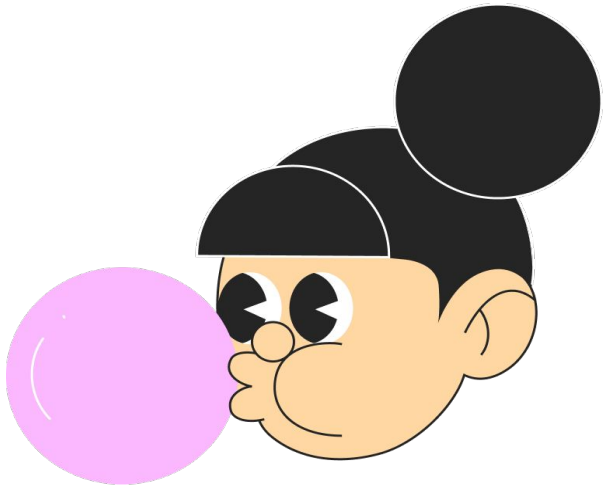
1. Атрибуты доступности (readonly/readwrite),
2. Атрибуты владения (retain/strong/copy/assign/unsafe_unretained/weak),
3. Атрибут атомарности (atomic/nonatomic).
4. Nullability атрибут (null_unspecified/null_resettable/nullable/nonnull) — появился в xcode 6.3



Атрибуты доступности

- **readwrite** — указывает, что свойство доступно и на чтение, и на запись, то есть будут сгенерированы и сеттер, и геттер. Это значение задается всем свойствам по умолчанию, если не задано другое значение.

- **readonly** — указывает, что свойство доступно только для чтения. Это значение стоит применять в случаях, когда изменение свойства «снаружи» во время выполнения объектом своей задачи нежелательно, либо когда значение свойства не хранится ни в какой переменной, а генерируется исходя из значений других свойств.



Атрибуты владения

1. **retain (соответствующая переменная должна быть с атрибутом __strong)** — это значение показывает, что в сгенерированном сеттере счетчик ссылок на присваиваемый объект будет увеличен, а у объекта, на который свойство ссылалось до этого, — счетчик ссылок будет уменьшен.

2. **strong (соответствующая переменная должна быть с атрибутом __strong)** — то значение аналогично retain, но применяется только при включенном автоматическом подсчете ссылок. При использовании ARC это значение используется по умолчанию. Используйте strong во всех случаях, не подходящих для weak и copy, и все будет хорошо.

3. **copy (соответствующая переменная должна быть с атрибутом __strong)** — при таком значении атрибута владения в сгенерированном сеттере соответствующей переменной экземпляра присваивается значение, возвращаемое сообщением copy, отправленным присваиваемому объекту.

Атрибуты владения

4. **weak (соответствующая переменная должна быть с атрибутом __weak)** — это значение аналогично assign и unsafe_unretained. Разница в том, что особая уличная магия позволяет переменным с таким значением атрибута владения менять свое значение на nil, когда объект, на который указывала переменная, уничтожается.

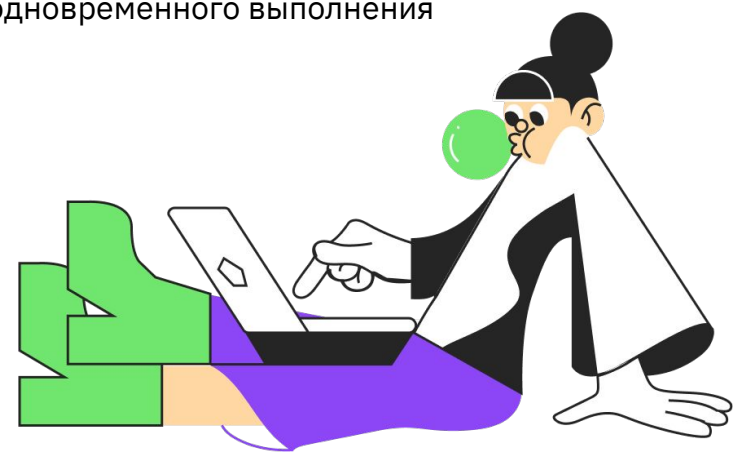
5. **unsafe_unretained (соответствующая переменная должна быть с атрибутом __unsafe_unretained)** — свойство с таким типом владения просто сохраняет адрес присвоенного ему объекта. Методы доступа к такому свойству никак не влияют на счетчик ссылок объекта. Он может удалиться, и тогда обращение к такому свойству приведет к крэшу (потому и unsafe).

6. **assign (соответствующая переменная должна быть с атрибутом __unsafe_unretained,** но так как атрибуты владения есть только у типов попадающих под ARC, с которыми лучше использовать strong или weak, это значение вам вряд ли понадобится) — просто присвоение адреса.

Атрибуты атомарности

- **atomic — это дефолтное значение для данного атрибута.** Оно означает, что акцессор и мутатор будут сгенерированы таким образом, что при обращении к ним одновременно из разных потоков, они не будут выполняться одновременно (то есть все равно сперва один поток сделает свое дело — задаст или получит значение, и только после этого другой поток сможет заняться тем же).

- **nonatomic — значение противоположное atomic** — у свойств с таким значением атрибута атомарности методы доступа не обременены защитой от одновременного выполнения в разных потоках, поэтому выполняются быстрее.



Атрибуты nullability

💡 Этот атрибут никак не влияет на генерируемые методы доступа. Он предназначен для того, чтобы обозначить, может ли данное свойство принимать значение nil или NULL.

1. **null_unspecified** — используется по умолчанию и ничего не говорит о том, может ли свойство принимать значение nil/NULL или нет .

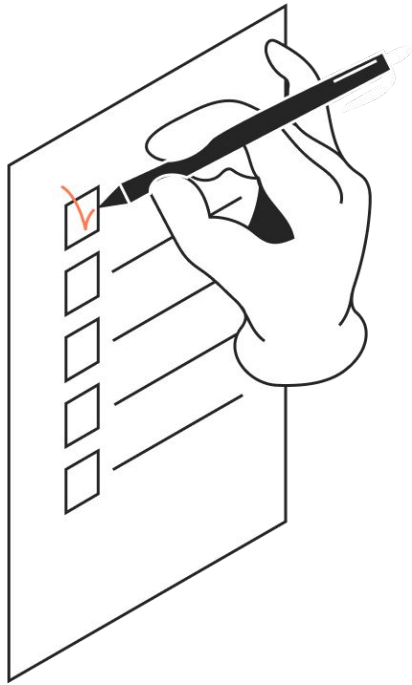
2. **null_resettable** — это значение свидетельствует о том, что геттер такого свойства никогда не вернет nil/NULL в связи с тем, что при задании такого значения, на самом деле свойству будет присвоено некое дефолтное.

3. **nonnull** — это значение свидетельствует о том, что свойство, помеченное таким атрибутом, не будет принимать значение nil/NULL.

4. **nullable** — это значение свидетельствует о том, что свойство может иметь значение nil/NULL.

Что почитать?

- Стивен Кочан. «Программирование на Objective-C».
- Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».
- https://www.objc.io/
- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/



Вопросы?

Вопросы?

Вопросы?