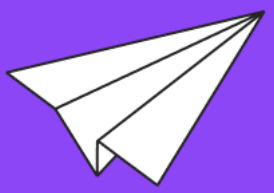
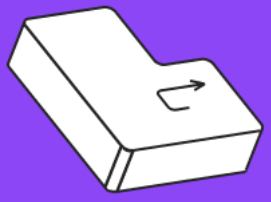
GeekBrains



Лекция 3. Работа с памятью в Objective – C, ARC и MRC

Рассмотрим работу памяти в Objective – C. Изучим, как и ручной подсчет ссылок (MRC), так и автоматический (ARC). Рассмотрим отличия от других языков программирования. Также, затронем понятие Side Table.



Оглавление

Основы работы с памятью

Управление памятью приложения — это процесс выделения памяти во время выполнения вашей программы, ее использования и освобождения, когда вы закончите с ней работать. Хорошо написанная программа использует как можно меньше памяти. В Objective – C – это также можно рассматривать как способ распределения владения ограниченными ресурсами памяти между множеством фрагментов данных и кода. Когда вы закончите работу с этой лекций, у вас будут знания, необходимые для управления памятью вашего приложения путем явного управления жизненным циклом объектов и их освобождения, когда они больше не нужны. Хотя управление памятью обычно рассматривается на уровне отдельного объекта, на самом деле вашей целью является управление графами объектов.

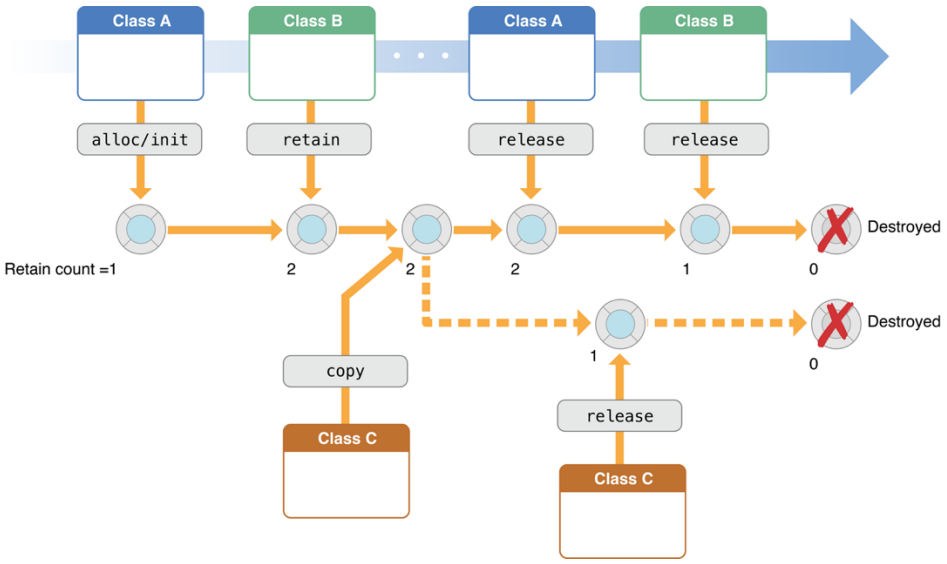
Необходимо убедиться, что у вас в памяти не больше объектов, чем вам действительно нужно.

Управление памятью – одна из наиболее сложных задач при программировании как на Objective-C, так и на других языках. Операционная система для работы программы может выделить ограниченные ресурсы: это касается и количества оперативной памяти, и открытых файлов, и сетевого подключения.

При работе с компьютером пользователь открывает программы, а при выполнении определенных задач закрывает их. Так он освобождает ресурсы системы. Но если пользователь не хочет или забывает сделать это, а разработчики никак не продумали работу с памятью, то программа постоянно открывает необходимые файлы для работы и не закрывает их после использования. Ресурсы, выделенные системой для программы, будут исчерпаны, и она не сможет далее корректно функционировать. Еще осторожнее относиться к управлению памятью следует на мобильных устройствах, так как ресурсы более ограничены.

Далеко не все программы используют сетевое подключение, но работать с памятью придется в любой программе. Ошибки памяти – одни из самых сложных задач при работе с C-подобными языками. Неосвобожденные ресурсы, выделенные для программы, приводят к утечке памяти.

Каждый объект в программе имеет свой жизненный цикл. При создании ему выделяется память, а при уничтожении – она должна освобождаться.



Введение

Objective-C предоставляет два метода управления памятью приложения.

1. MRC «Ручное сохранение-освобождение (retain / release)», в таком случае вы явно управляете памятью, отслеживая объекты, которыми владеете. Это реализуется с помощью модели, известной как подсчет ссылок, которую класс Foundation NSObject предоставляет в сочетании со средой выполнения (RunTime).

1. ARC (Автоматический подсчет ссылок), система использует ту же систему подсчета ссылок, что и MRC, но она вставляет соответствующие вызовы методов управления памятью во время компиляции. *Настоятельно рекомендуется использовать ARC для новых проектов.* Если вы используете ARC, обычно нет необходимости разбираться в базовой реализации, которая порой очень непростая, хотя в некоторых ситуациях это может быть полезно.

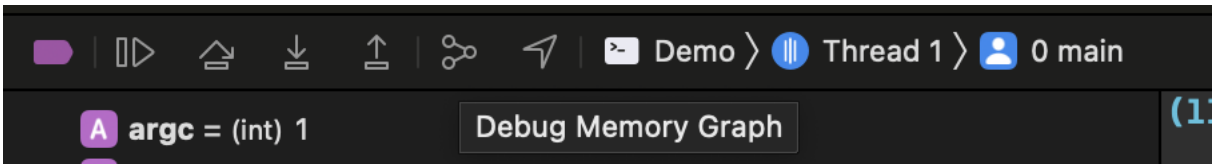
Существует два основных вида проблем, возникающих в результате неправильного управления памятью:

1. Освобождение или перезапись данных, которые все еще используются. Это вызывает повреждение памяти и обычно приводит к сбою приложения или, что еще хуже, к повреждению пользовательских данных.

1. Отсутствие освобождения данных, которые больше не используются, приводит к утечке памяти. Утечка памяти — это когда выделенная память не освобождается, даже если она больше никогда не используется. Утечки заставляют ваше приложение использовать постоянно увеличивающийся объем памяти, что, в свою очередь, может привести к снижению производительности системы или прекращению работы вашего приложения.

Однако думать об управлении памятью с точки зрения подсчета ссылок часто бывает контрпродуктивно, потому что вы склонны рассматривать управление памятью с точки зрения деталей реализации, а не с точки зрения ваших реальных целей. Вместо этого вы должны думать об управлении памятью с точки зрения владения объектами и графов объектов. Также есть технология отложенной очистки, а именно – блоки пула (@autorelease) автоматического освобождения, которые предоставляют механизм, с помощью которого вы можете отправить объекту «отложенное» сообщение об освобождение. Это полезно в ситуациях, когда вы хотите отказаться от владения объектом, но хотите избежать возможности его немедленного освобождения (например, когда вы возвращаете объект из метода).

Бывают случаи, когда вы можете использовать свои собственные блоки пула автоматического освобождения.
Чтобы выявить проблемы с вашим кодом во время компиляции, вы можете использовать статический анализатор Clang, встроенный в Xcode – Debug Memory Graph.



Основные правила управлению памятью

Модель управления памятью основана на владении объектами. Любой объект может иметь одного или нескольких владельцев. Пока у объекта есть хотя бы один владелец, он продолжает существовать. Если у объекта нет владельцев, исполняющая система автоматически уничтожает его. Чтобы было ясно, когда вы владеете объектом, а когда нет, Cocoa API устанавливает следующую политику:

Вы владеете любым созданным вами объектом (вы создаете объект с помощью метода, имя которого начинается с «alloc», «new», «copy» или «mutableCopy» (например, alloc, newObject или mutableCopy)).

Вы можете стать владельцем объекта, используя retain; (обычно гарантируется, что полученный объект останется действительным в методе, в котором он был получен, и этот метод также может безопасно вернуть объект вызывающему объекту.)

Вы используете retain в двух ситуациях:

1. В реализации метода доступа или метода инициализации, чтобы стать владельцем объекта, который вы хотите сохранить как значение свойства;
2. Для предотвращения признания объекта недействительным в качестве побочного эффекта какой-либо другой операции.

Когда он вам больше не нужен, вы должны отказаться от права собственности на объект, которым владеете. (вы отказываетесь от права собственности на объект, отправляя ему сообщение об освобождении(release) или сообщение об отложенном(autorelease) автоматическом освобождении.
Вы не должны отказываться от права собственности на объект, которым вы не владеете. Чтобы проиллюстрировать данный подход, рассмотрим следующий фрагмент кода:

```
{
    Person *aPerson = [[Person alloc] init];
    // ...
    NSString *name = aPerson.fullName;
    // ...
    [aPerson release];
}
```

Объект Person создается с помощью метода alloc, поэтому впоследствии ему отправляется сообщение об освобождении(release), когда он больше не нужен. Имя человека не извлекается ни одним из методов владения, поэтому ему не отправляется сообщение об освобождении. Однако обратите внимание, что в примере используется освобождение(release), а не отложенное освобождение(autorelease). Используйте autorelease, когда вам нужно отправить сообщение об отложенном выпуске — обычно при возврате объекта из метода.

Например, вы можете реализовать метод fullName следующим образом:

```
- (NSString *)fullName {
    NSString *string = [[[NSString alloc] initWithFormat:@"%@ %@",
                                        self.firstName, self.lastName] autorelease];
    return string;
}
```

Вы владеете строкой, возвращаемой alloc. Чтобы соблюдать правила управления памятью, вы должны отказаться от владения строкой до того, как потеряете ссылку на нее. Однако, если вы используете release, строка будет освобождена до того, как будет возвращена (и метод вернет недопустимый объект). Используя autorelease, вы показываете, что хотите отказаться от владения, но разрешаете вызывающему методу использовать возвращаемую строку до ее освобождения.

Вы также можете реализовать метод fullName следующим образом:

```
- (NSString *)fullName {
    NSString *string = [NSString stringWithFormat:@"%@ %@",
                                    self.firstName, self.lastName];
    return string;
}
```

Следуя основным правилам, вы не являетесь владельцем строки, возвращаемой stringWithFormat:, поэтому вы можете безопасно возвращать строку из метода. Напротив, следующая реализация неверна:

```
- (NSString *)fullName {
    NSString *string = [[NSString alloc] initWithFormat:@"%@ %@",
                                        self.firstName, self.lastName];
    return string;
}
```

Согласно соглашению об именах, нет ничего, что указывало бы на то, что возвращаемая строка принадлежит вызывающему объекту метода fullName. Таким образом, у вызывающей стороны нет причин освобождать возвращаемую строку, и, таким образом, произойдет утечка.

Метод dealloc. Удаление объекта из памяти

Класс NSObject определяет метод dealloc, который вызывается автоматически, когда у объекта нет владельцев и его память освобождается — в терминологии Cocoa он «освобождается». Роль метода dealloc состоит в том, чтобы освободить собственную память объекта и избавиться от любых ресурсов, которые он содержит, включая владение любыми переменными экземпляра объекта.

В следующем примере показано, как можно реализовать метод dealloc для класса Person:

```
@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

```

🔥 **Важно:** никогда не вызывайте метод dealloc другого объекта напрямую. Вы должны вызвать реализацию суперкласса в конце своей реализации. Не следует привязывать управление системными ресурсами к времени жизни объекта; Не используйте dealloc для управления ограниченными ресурсами. Когда приложение завершает работу, объектам не может быть отправлено сообщение об освобождении. Поскольку память процесса автоматически очищается при выходе, более эффективно просто позволить операционной системе очистить ресурсы, чем вызывать все методы управления памятью.

Практическое управление памятью.

Все основные концепции описаны выше, но для просты, есть несколько практических шагов, которые вы можете предпринять, чтобы упростить управление памятью и помочь обеспечить надежность и надежность вашей программы, в то же время сведя к минимуму ее требования к ресурсам.

Используйте методы доступа, чтобы упростить управление памятью. Если в вашем классе есть свойство, являющееся объектом, вы должны убедиться, что любой объект, установленный в качестве значения, не освобождается во время его использования. Поэтому вы должны заявить право собственности на объект, когда он установлен. Вы также должны убедиться, что затем отказываетесь от права собственности на любую текущую ценность. Иногда это может показаться утомительным или педантичным, но, если вы постоянно используете методы доступа, вероятность возникновения проблем с управлением памятью значительно снижается. Если вы используете сохранение и освобождение для переменных экземпляра в своем коде, вы почти наверняка делаете что-то не так.

Рассмотрим объект Counter, счетчик которого вы хотите установить.

```
1 @interface Counter: NSObject
2 @property (nonatomic, retain) NSNumber *count;
3 @end;
4
```

Свойство объявляет два метода доступа. Как правило, вы должны попросить компилятор синтезировать методы; однако поучительно посмотреть, как они могут быть реализованы. В методе доступа «get» вы просто возвращаете синтезированную переменную экземпляра, поэтому нет необходимости сохранять или освобождать:

```
1 - (NSNumber *)count {
2     return _count;
3 }
4
```

В методе «set», если все остальные играют по одним и тем же правилам, вы должны предположить, что новый счет может быть удален в любое время, поэтому вы должны стать владельцем объекта, отправив ему сообщение о сохранении, чтобы гарантировать, что он не будет. Здесь вы также должны отказаться от владения старым объектом счетчика, отправив ему сообщение об освобождении (release). (Отправка сообщения в nil разрешена в Objective-C, поэтому реализация все равно будет работать, если _count еще не был установлен.

```objc
- (void)setCount:(NSNumber *)newCount {
    [newCount retain];
    [_count release];
    // Make the new assignment.
    _count = newCount;
}

```

Используйте методы доступа для установки значений свойств
Предположим, вы хотите реализовать метод сброса счетчика. У вас есть несколько вариантов. Первая реализация создает экземпляр NSNumber с помощью alloc, поэтому вы уравновешиваете это освобождением (release).

```objc
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [self setCount:zero];
    [zero release];
}

```

Второй использует удобный конструктор для создания нового объекта NSNumber. Поэтому нет необходимости сохранять или освобождать сообщения.

```objc
- (void)reset {
    NSNumber *zero = [NSNumber numberWithInteger:0];
    [self setCount:zero];
}

```

Обратите внимание, что оба используют метод доступа set.
Следующее почти наверняка будет работать правильно для простых случаев, но как бы заманчиво ни было отказаться от методов доступа, это почти наверняка приведет к ошибке на каком-то этапе (например, когда вы забудете сохранить или освободить, или если семантика управления памятью для изменения переменной экземпляра).

```objc
- (void)reset {
    NSNumber *zero = [[NSNumber alloc] initWithInteger:0];
    [_count release];
    _count = zero;
}

```

Методы доступа

Не используйте методы доступа в методах инициализатора и деинициализаторах. Также обратите внимание, что, если вы используете наблюдение за ключом и

значением, изменение переменной таким образом не совместимо с KVO. Не используйте методы доступа в методах инициализатора, где вы не должны использовать методы доступа для установки переменной экземпляра, являются методы инициализации и деалок. Чтобы инициализировать объект-счетчик числовым объектом, представляющим ноль, вы можете реализовать метод init следующим образом:

```
- init {
    self = [super init];
    if (self) {
        _count = [[NSNumber alloc] initWithInteger:0];
    }
    return self;
}

```

Чтобы разрешить инициализацию счетчика со значением, отличным от нуля, вы можете реализовать метод initWithCount: следующим образом:

```
- initWithCount:(NSNumber *)startingCount {
    self = [super init];
    if (self) {
        _count = [startingCount copy];
    }
    return self;
}

```

Поскольку класс Counter имеет переменную экземпляра объекта, вы также должны реализовать метод Dealloc. Он должен отказаться от владения любыми переменными экземпляра, отправив им сообщение об освобождении, и, в конечном итоге, он должен вызвать реализацию super:

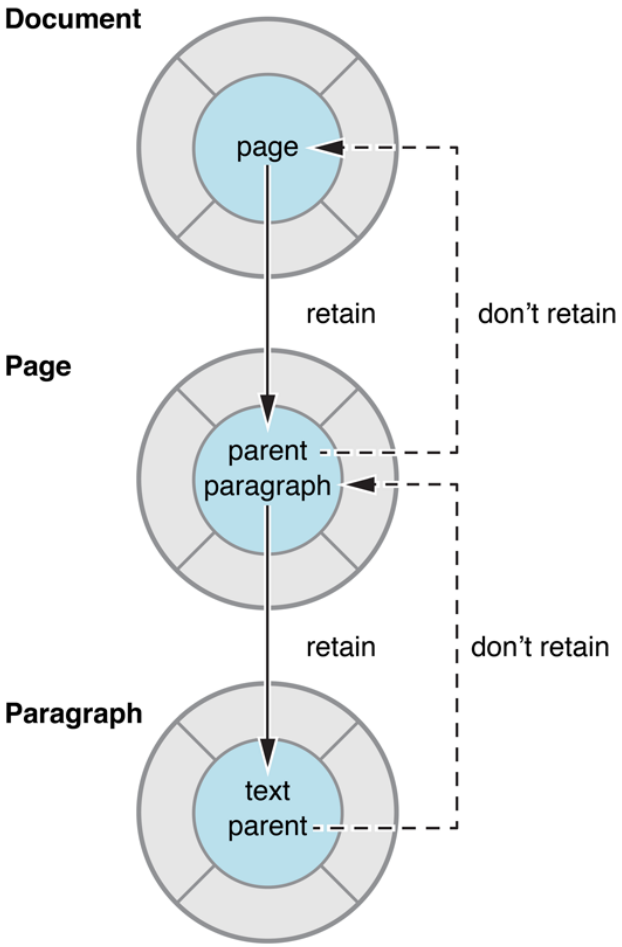
```
- (void)dealloc {
    [_count release];
    [super dealloc];
}

```

Retain Cycle

Сохранение объекта создает сильную ссылку на этот экземпляр. Объект не может быть освобожден до тех пор, пока не будут освобождены все его сильные ссылки. Таким образом, может возникнуть проблема, известная как цикл сильных ссылок, если два объекта могут иметь циклические ссылки, то есть они имеют сильную ссылку друг на друга (либо напрямую, либо через цепочку других объектов, каждый

из которых имеет сильную ссылку на следующий объект). Объект Document имеет ссылку на объект Page для каждой страницы документа. Каждый объект Page имеет свойство, которое отслеживает, в каком документе он находится. Если объект Document имеет строгую ссылку на объект Page, а объект Page имеет строгую ссылку на объект Document, ни один объект не может быть освобожден. Счетчик ссылок документа не может стать равным нулю, пока объект Page не будет освобожден. Чтобы решить проблему цикла сильных ссылок используют слабые(weak) или бесхозные(unowned) ссылки.

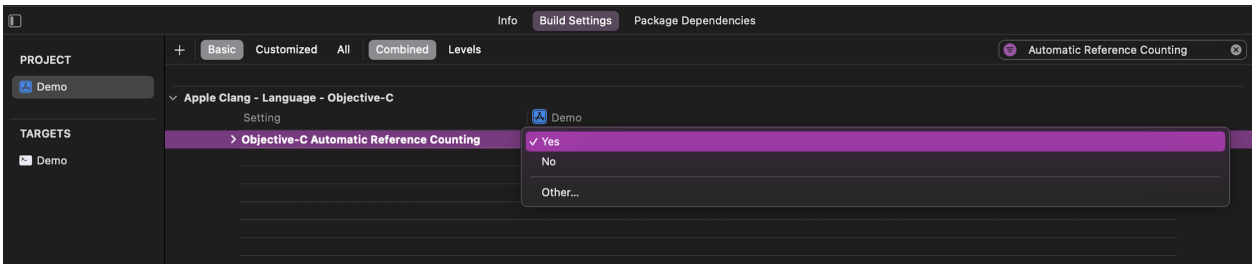


ARC

Итак, в итоге, чтобы упростить работу с памятью, компания Apple предложила решение – автоматический подсчет ссылок (**automatic reference counting – ARC**). ARC автоматически отслеживает объекты и определяет, какие из них уничтожить. При написании программы можно не задумываться об управлении памятью, а при

компиляции сообщения **retain** и **release** будут проставлены автоматически. Работа с памятью перекладывается с разработчика на компилятор. ARC включена по умолчанию.

Чтобы отключить или включить ARC, необходимо в настройках проекта перейти в раздел **Build Settings**. После – воспользоваться поиском и ввести «Automatic Reference Counting». Напротив найденного элемента поставить значение.



Для применения ARC необходимо выполнять три важных условия:

1. Однозначно идентифицировать объект, которым необходимо управлять;
2. Указать, как именно управлять объектом;
3. Иметь надежный механизм передачи владения объектом.

Рассмотрим пример:

```
1 NSString **string;
2 string = malloc(10 * sizeof(NSString *));
3
```

В коде создается С-массив, который указывает на 10 строк. ARC не может управлять С-массивами, так как они не являются хранимыми. Следовательно, надо иметь возможность увеличить и уменьшить счетчик ссылок на объект, который является производным от класса NSObject. С-массив не относится к таким объектам. А при передаче объекта программа должна иметь возможность передавать владение объектом.

Ссылки на объекты

В соответствии с жизненным циклом объекта необходимо определиться с моментом, когда объект становится ненужным и подлежит уничтожению. Для этого в Objective-C применяется подсчет ссылок. У каждого объекта существует связанная целочисленная переменная, которая указывается на количество ссылок на объект (счетчик ссылок). При работе с объектом его счетчик ссылок увеличивается, а при завершении работы – уменьшается. Когда он становится

нулевым, объект уничтожается, а выделенная для него память освобождается и возвращается системе.
При вызове методов **alloc** или **new** счетчик ссылок становится равным 1. Чтобы увеличить счетчик, необходимо послать сообщение **retain**, а для уменьшения счетчика – **release**.

При уничтожении объекта вызывается метод **dealloc**. Это происходит спустя некоторое время после того, как счетчик ссылок обнуляется и система получает об этом уведомление. **Dealloc** полностью уничтожает объект, освобождая при этом используемую память.
Создадим объект и выведем в консоль отметки о его создании и уничтожении:

```
@implementation Object

- (instancetype)init
{
    self = [super init];
    if (self) {
        NSLog(@"init: счетчик ссылок - 1");
    }
    return self;
}

- (void)dealloc {
    NSLog(@"dealloc");
    [super dealloc];
}

@end

```

Теперь создадим объект и попробуем несколько раз вызвать **retain** и **release** (при выключенном ARC):

```
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;
}

```

После выполнения программы в консоли можно будет увидеть результат:

```
init: счетчик ссылок - 1
count - 2
count - 3
count - 2
count - 1
dealloc

```

Владение объектом

На первый взгляд это кажется достаточно простым: когда объект используется, нужно увеличить, а когда работа с ним завершается – уменьшить. Но необходимо определить концепцию владения объектом. Владелец отвечает за удаление объекта и освобождение памяти. В предыдущем примере владельцем объекта **Object** является функция **main()**. Затруднения могут возникнуть, когда у объекта несколько владельцев. Рассмотрим пример, где добавим дополнительный объект:

```
@interface AnotherObject : NSObject

@property (nonatomic, strong) Object *object;

@end

```

Теперь в функции **main()** создадим оба объекта и передадим первый во второй:

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

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

        [anotherObject setObject:object];

    }
    return 0;
}

```

Теперь функция **main()** не может однозначно владеть первым объектом и отвечать за его уничтожение. Но и второй объект также не может уничтожить первый, ведь он может использоваться в функции **main()** в дальнейшем.

В таких случаях применяется захват объекта, суть которого – в увеличении счетчика объекта при присваивании. Это обозначает количество владельцев у объекта. Соответственно, первый объект передается во второй, и в сеттере увеличивается счетчик ссылок. Функция **main()** сможет освободиться от владения первым объектом, а второй объект будет ответственен за его уничтожение.

```objc
- (void)setObject:(Object *)object {
    [object retain];     // Увеличиваем счетчик ссылок
    [object release];    // Освобождаем функцию main от владения
    _object = object;
}

```

Так **AnotherObject** станет владельцем объекта и будет отвечать за его уничтожение и освобождение памяти.

Автоматическое освобождение

В среде Cocoa существует концепция пула автоматического освобождения (мы встречались с ним в функции **main**). Это коллекция объектов, которая автоматически выполняет их освобождение.
У класса **NSObject** существует метод **autorelease**, который планирует отправление сообщения **release** в будущем. При вызове этого метода объект добавляется к объекту класса **NSAutoreleasePool**. При уничтожении этого пула всем объектам отправляется **release**. Всем объектам, которые необходимо освободить в пуле, следует отправить сообщение **autorelease**.
Среди видов пула автоматического освобождения выделяют ключевое слово **@autorelesapool** и объект класса **NSAutoreleasePool**. Именно первая конструкция встречается в функции **main**:

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

    }
    return 0;
}

```

Все, что содержится в теле **@autoreleasepool**, помещается в пул и будет освобождено по его окончании. Второй способ – использовать объект класса **NSAutoreleasePool**:

```objc
    NSAutoreleasePool *pool;
    pool = [NSAutoreleasePool new];

    NSString *string = [[NSString alloc] init];

    [string autorelease];

    [pool release];

```

При создании пула он автоматически становится активным, и все объекты, которым будет отправлено сообщение **autorelease**, будут добавлены именно в него (вплоть до вызова у пула метода **release**).

Оба способа эффективны, но применение ключевого слова – более элегантно и быстродейственно. Это связано с тем, что система лучше знает, как создавать и уничтожать пулы.

Правила работы с памятью

- При создании объекта с использованием методов **alloc**, **new** или **copy** его освобождение остается ответственностью разработчика. Он обязан послать сообщение **release** или **autorelease** по завершении работы, чтобы избавиться от ненужных объектов и освободить ресурсы системы.
- Если захватывается объект и его счетчик ссылок равен 1, а также ему посылается сообщение **autorelease**, то более дополнительных действий не требуется;
- Если объект был захвачен, то по завершении работы с ним необходимо его освободить. Количество сообщений **retain** и **release** должно совпадать.

Временные объекты

У некоторых объектов существуют конструкторы, которые позволяют не задумываться о **release**. Это методы, которые не применяют **alloc**, **new** или **copy** при создании. Например, у объекта класса **NSArray** существует следующий конструктор:

```
1 NSArray *array = [NSArray array]
```

Эта реализация позволяет не задумываться об освобождении таких объектов. Они автоматически помещаются в пул освобождения при условии, что счетчик ссылок не будет увеличен умышленно (например, при захвате объекта).

Swift – Side Table

Side Table — это разумное улучшение системы управления ссылками, впервые представленной в Swift 4. Давайте подробнее рассмотрим эту концепцию и какие проблемы она решает при существующем подходе.

1. Сильная ссылка (Strong reference) прочно удерживает экземпляр и не позволяет освободить его, пока сохраняется эта сильная ссылка, увеличивает счетчик ссылок на 1.
2. Слабая ссылка (Weak reference) — это ссылка, которая не удерживает сильный контроль над экземпляром, на который ссылается, и поэтому не мешает ARC избавиться от экземпляра, на который ссылается, не увеличивает счетчик ссылок.
3. Бесхозная ссылка (Unowned reference), как слабая не держит сильную хватку. Доступ к бесхозным ссылкам с нулевым счетчиком приводит к ошибкам времени выполнения. Используйте его, когда другой экземпляр имеет такой же или более длительный срок службы, не увеличивает счетчик ссылок.

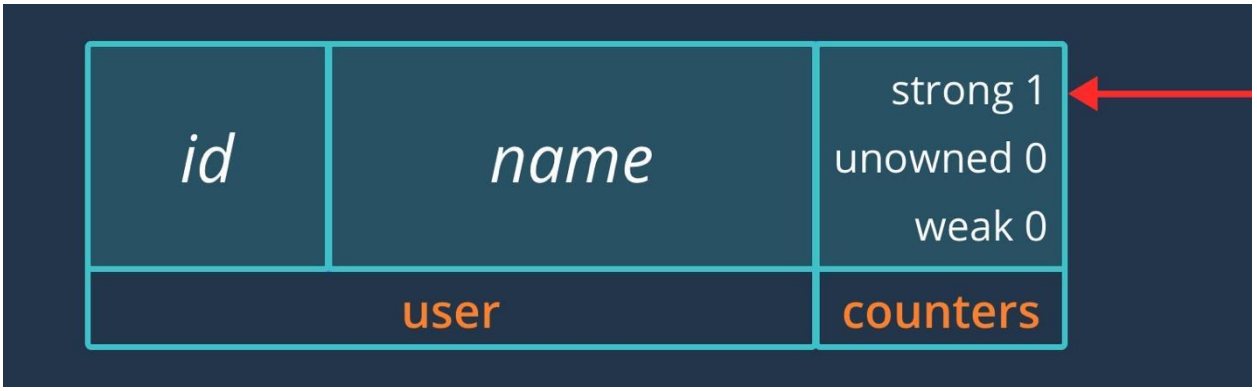
Также стоит отметить, что объекты в Swift концептуально имеют несколько счетчиков: для сильных, слабых и бесхозных ссылок. Эти счетчики хранятся либо в строке сразу после isa, либо в боковой таблице (Side Table), что будет объяснено чуть позже. Кроме того, слабые и бесхозные ссылки имеют дополнительный +1 от сильных ссылок. Дополнительное значение уменьшается после освобождения и деинициализации объекта соответственно. Для простоты материала я буду использовать 0 в качестве отправной точки для обоих типов ссылок.

Слабые ссылки до Swift 4

Чтобы продемонстрировать, как работает старый подход, давайте сначала рассмотрим пример.

```
1 class User {
2     let id: Int
3     let email: String
4 }
```

Вот класс User с двумя свойствами. Следующая картинка представляет этот объект в памяти.



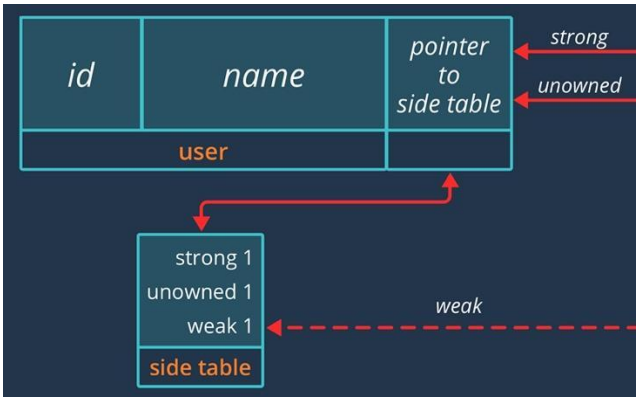
Класс, свойства и счетчики ссылок хранятся встроенными. Это обеспечивает несколько более быстрый доступ, чем хранение данных во внешнем блоке памяти.

Предположим, на пользователя ссылается одна слабая ссылка, и через некоторое время сильный счетчик обнуляется, в то время как слабый счетчик все еще отличен от нуля.

В этой ситуации объект предназначен для удаления из памяти с помощью автоматического подсчета ссылок. Несмотря на то, что это вызывает деинициализацию объекта, его память не освобождается. Слабый счетчик уменьшается, когда другой объект обращается к уничтоженному по слабой ссылке. Когда слабый счетчик достигает нуля, память окончательно освобождается. Это означает, что наш пользователь становится зомби – объектом, который может занимать память в течение длительного времени.

Понимание Swift Side Tables

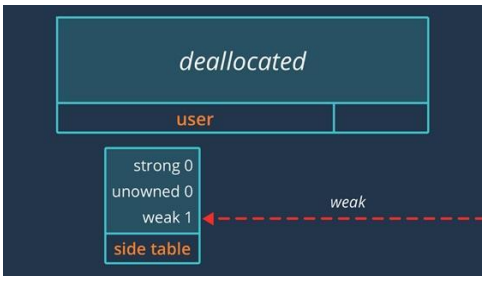
Side Table — это отдельный блок памяти, в котором хранится дополнительная информация об объекте. В настоящее время он хранит счетчики ссылок и флаги.



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.



Side Table вносит ощутимые улучшения в существующий подход. Во-первых, это позволяет вовремя полностью освобождать объекты, на которые указывают слабые ссылки. Более того, это может открыть двери для сохраненных свойств в расширениях в будущих выпусках swift.

Атрибуты свойства

Когда свойства начинают усложняться, это значит к ним присоединяются атрибуты. Это инструкции компилятора, которые влияют на автоматически сгенерированные методы доступа и бывают разных форм. Мы не используем никаких явных атрибутов прямо сейчас, но мы используем неявные атрибуты: нашему свойству имени присваиваются значения по умолчанию, которые влияют на то, как оно работает. Со временем эти значения по умолчанию немного изменились, но, по

моему опыту, большинство разработчиков просто пишут атрибуты для всего, даже если они совпадают со значениями по умолчанию.
Для начала поделим все атрибуты, которые есть у свойства, на группы:

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, позволяющие задать имя сеттера и геттера, рассматривать не будем — для них как таковых правил нет, за исключением тех, что предусматривает используемый вами Code Style. Явно или неявно, но атрибуты всех типов указываются у каждого свойства.

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. Например, есть у объекта User свойства firstName и lastName, и для удобства заданное readonly свойство fullName, значение которого генерируется исходя из значений первых двух свойств.

В случае, когда нежелательно изменение свойства «снаружи», оно, как правило, объявляется в интерфейсе класса как readonly, а потом переопределяется как readwrite в расширении класса (class extension), чтобы внутри класса также изменять значение не напрямую в переменной, а через сеттер.

Атрибуты владения

Это самый обширный тип атрибутов, во многом из-за сосуществования ручного и автоматического управления памятью. При включенном ARC у переменных, как и у свойств, есть атрибут владения, только в этом случае набор значений меньше, чем у свойств: __strong/__weak/__unsafe_unretained, и касается это только тех типов данных, которые подпадают под действие ARC. Поэтому при описании значений этого атрибута для свойств будем еще указывать, какое значение атрибута владения должно быть у соответствующей переменной экземпляра при

включенном ARC (если переменная создается автоматически — она сразу создается с нужным значением этого атрибута. Если же вы определяете переменную сами — нужно вручную задать ей правильное значение атрибута владения).

- **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. Это значение применимо при выключенном ARC для всех Objective-C классов во всех случаях, когда никакие другие значения не подходят. Это значение сохранилось со времен, когда ARC еще не было и, хотя ARC и не запрещает его использование, при включенном автоматическом подсчете ссылок лучше вместо него использовать strong.

```objc
//примерно вот такой сеттер будет сгенерирован для свойства
//с таким значением атрибута владения с отключенным ARC
-(void)setFoo(Foo *)foo {
  if (_foo ≠ foo) {
    Foo *oldValue = _foo;

    //увеличивается счетчик ссылок на новый объект и укзатель на него сохраняется в ivar
    _foo = [foo retain];

    //уменьшается счетчик ссылок на объект, на который раньше указывало свойство
    [oldValue release];
  }
}

```

- **strong** (соответствующая переменная должна быть с атрибутом __strong) — это 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.
- **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.

Использование этого значения атрибута владения накладывает некоторые ограничения на класс объекта:

1. Класс должен поддерживать протокол NSCopying,
2. Класс не должен быть изменяемым (mutable). У некоторых классов есть mutable-подкласс, например, NSString-NSMutableString. Если ваше свойство — экземпляр «мутабельного» класса, использование copy приведет к нежелательным последствиям, так как метод copy вернет экземпляр его «немутабельного» сородича. Например, вызов copy у экземпляра NSMutableString вернет экземпляр NSString.

```objc
@property (copy, nonatomic) NSMutableString *foo;
    ...
    //в сгенерированном сеттере будет примерно следующее
- (void)setFoo:(NSMutableString)foo {
    _foo = [foo copy]; //метод copy класса NSMutableString вернет объект типа NSString, так что после присвоения значения этому свойству, оно будет указывать на неизменяемую строку, и вызов методов, изменяющих строку, у этого объекта приведет к крэшу
}
```

- **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, что очень хорошо сказывается на устойчивости работы приложения (ибо, как известно, nil отвечает на любые сообщения, а значит никаких вам EXC_BAD_ACCESS при обращении к уже удаленному объекту). Это значение атрибута владения стоит использовать при включенном ARC для исключения retain cycle'ов для свойств, в которых хранится указатель на делегат объекта и им подобных. Это единственное значение атрибута владения, которое не поддерживается при выключенном ARC (как и при включенном ARC на iOS до версии 5.0).
- **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). Это значение использовалось вместо weak, когда уже появился ARC, но нужно было еще поддерживать iOS 4.3. Сейчас его использование можно оправдать разве что скоростью работы (есть сведения, что магия weak свойств требует значительного времени, хотя, конечно, невооруженным глазом при нынешней производительности этого не заметишь), поэтому, особенно на первых порах, использовать его не стоит.
- **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. Без ARC является дефолтным значением атрибута владения. Его стоит применять к свойствам типов, не попадающих под действие ARC (к ним относятся примитивные типы и так называемые необъектные типы (non-object types) вроде CFTypeRef). Без ARC он также используется вместо weak для

исключения retain cycle’ов для свойств, в которых хранится указатель на делегат объекта и им подобных.

Атрибут 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). Из-за особенностей реализации у свойств с таким значением атрибута атомарности нельзя переопределять только один из методов доступа (уж если переопределяете, то переопределяйте оба, и сами заморочьтесь с защитой от одновременного выполнения в разных потоках). Не стоит путать атомарность свойства с потокобезопасностью объекта. К примеру, если вы в одном потоке получаете значение имени и фамилии из свойств объекта, а в другом — изменяете эти же значения, вполне может получиться так, что значение имени вы получите старое, а фамилии — уже измененное. Соответственно, применять стоит для свойств объектов, доступ к которым может осуществляться из многих потоков одновременно, и нужно спастись от получения каких-нибудь невалидных значений, но при необходимости обеспечить полноценную потокобезопасность, одного этого может оказаться недостаточно. Кроме прочего стоит помнить, что методы доступа таких свойств работают медленнее, чем nonatomic, что, конечно, мелочи в масштабах вселенной, но все же копейка рубль бережет, поэтому там, где нет необходимости, лучше использовать nonatomic.
- **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. Это значение пригодно для большинства свойств, так как большинство объектов все-таки используются только в одном потоке, и нет смысла «обвешивать» их лишними фичами. В общем, для всех свойств, для которых не сможете объяснить, почему оно должно быть atomic, используйте nonatomic, и будет вам fast and easy and smart и просто праздник какой-то.

Nullability атрибут

Этот атрибут никак не влияет на генерируемые методы доступа. Он предназначен для того, чтобы обозначить, может ли данное свойство принимать значение nil или NULL. Xcode использует эту информацию при взаимодействии Swift-кода с вашим

классом. Кроме того, это значение используется для вывода предупреждений в случае, если ваш код делает что-то, противоречащее заявленному поведению. Например, если вы пытаетесь задать значение nil свойству, которое объявлено как nonnull. А самое главное, эта информация будет полезна тому, кто будет читать ваш код. На использование этого атрибута есть пара ограничений:

1. Его нельзя использовать для примитивных типов (им и не нужно, так как они в любом случае не принимают значений nil и NULL)
2. его нельзя использовать для многоуровневых указателей (например, id* или NSError**)

- **null_unspecified** — используется по умолчанию и ничего не говорит о том, может ли свойство принимать значение nil/NULL или нет. До появления этого атрибута именно так мы и воспринимали абсолютно все свойства. Это плохо в плане содержательности заголовков, поэтому использование этого значения не рекомендуется.
- **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. А так как генерируемые методы доступа от значения этого атрибута не зависят, вы сами должны будете либо переопределить сеттер так, чтобы при получении на вход nil/NULL он сохранял в ivar значение по умолчанию, либо переопределить геттер так, чтобы он возвращал дефолтное значение в случае, если соответствующая ivar == nil/NULL. Соответственно, если у вашего свойства есть дефолтное значение — объявляйте его как null_resettable.
- **nonnull** — это значение свидетельствует о том, что свойство, помеченное таким атрибутом, не будет принимать значение nil/NULL. На самом деле, вы все еще можете получить nil, если, к примеру, попробуете получить значение этого свойства у nil'а и просто потому, что Xcode не очень жестко следит за этим. Но это уже будет ваша ошибка и Xcode будет по мере сил указывать вам на нее в своих предупреждениях. Использовать стоит, если вы уверены, что значение данного свойства никогда не будет nil/NULL и хотите, чтобы IDE помогала вам следить за этим.
- **nullable** — это значение свидетельствует о том, что свойство может иметь значение nil/NULL. Оно так же, как и null_unspecified, ни к чему не обязывает, но все же ввиду его бОльшей определенности, среди этих двух правильнее использовать именно nullable. Таким образом, если вам не подходит ни null_resettable, ни nonnull — используйте nullable.

Для большего удобства вы можете изменить значение по умолчанию с

null_unspecified на nonnull для определенного блока кода. Для этого нужно поставить NS_ASSUME_NONNULL_BEGIN перед таким блоком и NS_ASSUME_NONNULL_END — после него.

```
//например вот такое объявление
@property (copy, nonatomic, nonnull) NSString *foo;
@property (copy, nonatomic, nonnull) NSString *str;
@property (strong, nonatomic, nullable) NSNumber *bar;
@property (copy, nonatomic, null_resettable) NSString *baz;

//аналогично следующему блоку
NS_ASSUME_NONNULL_BEGIN
@property (copy, nonatomic) NSString *foo;
@property (copy, nonatomic) NSString *str;
@property (strong, nonatomic, nullable) NSNumber *bar;
@property (copy, nonatomic, null_resettable) NSString *baz;
NS_ASSUME_NONNULL_END
```

Что можно почитать:

1. Side Table https://github.com/apple/swift/blob/d1c87f3c936c41418ee93320e42d523b3f51b6df/stdlib/public/SwiftShims/RefCount.h#L44
2. ARC - https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html#ID52
3. https://habr.com/ru/post/341014/
4. https://devsday.ru/blog/details/1781
5. Стивен Кочан. «Программирование на Objective-C»;
6. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».