Урок 5

Работа с памятью ARC. Атрибуты свойств

Обзор работы с памятью и ARC. Рассмотрение атрибутов свойств.

Работа с памятью

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.

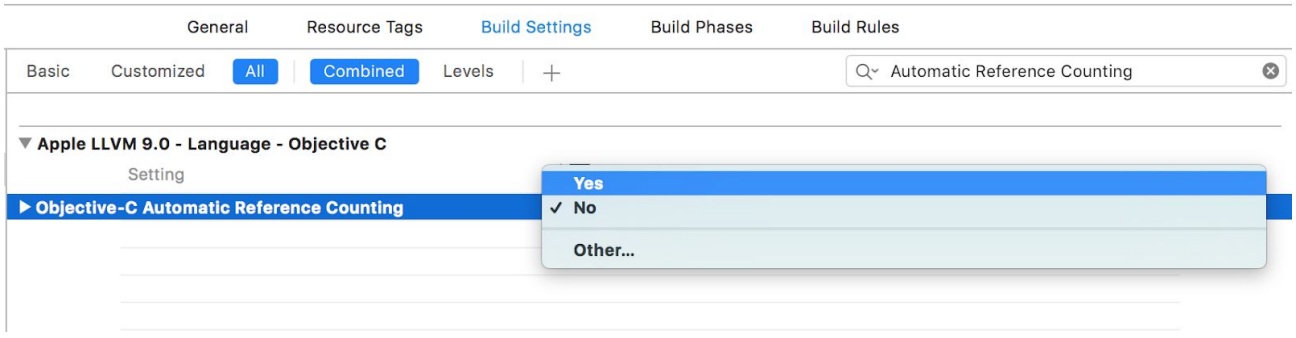
Далеко не все программы используют сетевое подключение, но работать с памятью придется в любой программе. Ошибки памяти – одни из самых сложных задач при работе с C-подобными языками. Неосвобожденные ресурсы, выделенные для программы, приводят к утечке памяти.

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

ARC

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:

```
NSString **string;
string = malloc(10 * sizeof(NSString *));
```

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.

```
- (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**:

```
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**:

```
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:

```
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).

Атрибуты свойств

Ранее мы рассмотрели свойства объектов, теперь – изучим их атрибуты. В объектно-ориентированном программировании правилом хорошего тона является применение геттеров и сеттеров.

Для свойств существуют следующие атрибуты:

1. Доступности;
2. Владения;
3. Атомарности;
4. Nullability.

Атрибуты доступности

Атрибуты доступности позволяют установить возможности изменения свойства.

Виды атрибутов доступности:

- **readwrite** – указывает, что свойство возможно использовать для чтения и записи. Этот атрибут устанавливается всем свойствам по умолчанию, если не указано иного.
- **readonly** – указывает, что свойство доступно только для чтения. Позволяет ограничить изменение значения.

Атрибуты владения

Атрибуты владения отвечают за определение типа ссылки на объект.

Виды атрибутов владения:

- **retain** – показывает, что счетчик ссылок на присваиваемый объект увеличен, а у объекта, на который свойство ссылалось до этого, счетчик будет уменьшен. Применяется для всех классов Objective-C, когда иные значения не подходят. **Retain** было определено до появления ARC. Несмотря на то, что ARC допускает его использование, лучше применять вместо него **strong**.

Пример при выключенном ARC:

```
- (void)setValue:(NSString *)value {
    if (_value != value) {
        NSString *oldValue = _value;

        _value = [value retain];    // Счетчик ссылок на новый объект
увеличивается

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

- **strong** – это значение схоже с **retain**, но применяется только при включенном ARC. По умолчанию устанавливается всем свойствам, в которых не указано иного. **Strong** применяется во всех случаях, которые не подходят для значений **weak** и **copy**;
- **copy** – при нем переменной присваивается значение, которое возвращается методом **copy**. Для использования этого атрибута необходимо, чтобы присваиваемый объект поддерживал протокол **NSCopying** и был неизменяемым;

- **weak** – это значение сравнимо с **assign** и **unsafe_unretained**. Переменные с этим атрибутом изменяют свое значение на **nil** – при условии, что объект был уничтожен;
- **unsafe_unretained** – это значение сохраняет адрес объекта, который был присвоен переменной. Методы доступа никак не влияют на счетчик ссылок объекта;
- **assign** – это значение сохраняет адрес, как и **unsafe_unretained**. Применяется для типов, которые не попадают под действие ARC: примитивных и необъектных.

Атрибуты атомарности

Атрибуты атомарности бывают двух типов: **atomic** и **nonatomic**.

- **Atomic** – значение, которое устанавливается свойствам по умолчанию. При нем геттер и сеттер не смогут выполняться одновременно при обращении к ним из разных потоков. Сначала один поток обработает необходимый геттер или сеттер, и только после этого к ним сможет обратиться другой. При переопределении сеттера необходимо также переопределить геттер, и наоборот. **Atomic** обязует переопределять оба метода. Без конкретной необходимости лучше применять **nonatomic**, так как методы доступа при **atomic** работают медленнее;
- **Nonatomic** – полная противоположность **atomic**. У свойств, которые имеют данный атрибут, отсутствует защита от одновременного выполнения в разных потоках. Это позволяет обрабатывать их быстрее.

Nullability

Данный атрибут обозначает, может ли свойство принимать значение **nil**. Если установлено некорректное значение, он предупредит разработчика. Этот атрибут нельзя использовать для примитивных типов, так как им не может установиться значение nil. Он также не подходит для многоуровневых указателей, таких как **id*** или **NSError****.

- **null_unspecified** – применяется для всех свойств по умолчанию и не сообщает о возможности свойства принимать **nil**;
- **null_resettable** гарантирует, что геттер свойства никогда не вернет значение **nil**. По умолчанию будет присвоено некоторое дефолтное значение. Для данного атрибута будет необходимо переопределить сеттер так, чтобы было невозможно установить значение **nil**;
- **nonnull** указывает на то, что свойство не может принимать **nil**. Применяется для переменных, которым нежелательно хранить это значение;
- **nullable** указывает на то, что свойство может иметь значение nil.

Практика

Создание программы «Автомобиль» с использованием ручного управления памятью

Чтобы создать приложение «Автомобиль», используя ручное управление, необходимо выключить ARC. Автоматический пул использовать тоже не будем.

Пусть основными компонентами нашего автомобиля будут колеса и двигатель. Создадим объект **Wheel**:

```
// Wheel.h

@interface Wheel : NSObject

- (instancetype) initWithNumber:(NSNumber *)number;
@property (nonatomic, strong) NSNumber *number;
@end

// Wheel.m
@implementation Wheel

- (instancetype)initWithNumber:(NSNumber *)number {
    self = [super init];
    if (self) {
        [number retain];
        [number release];
        _number = number;
        NSLog(@"Create Wheel %@", number);
    }
    return self;
}

- (void)dealloc {
    NSLog(@"Dealloc Wheel - %@", _number);
    [_number release];
    [super dealloc];
}

@end
```

Для объекта **Wheel** реализован собственный конструктор, который принимает идентификационный номер колеса. Еще он имеет свойство, которое хранит этот номер.

При инициализации осуществляется захват для объекта **number,** и объект класса **Wheel** становится его владельцем.

При вызове метода **dealloc** выводится сообщение об уничтожении объекта и освобождается объект **number**.

Далее необходимо создать объект двигателя (Engine):

```
// Engine.h

@interface Engine : NSObject

- (instancetype) initWithModel:(NSString *)model;

@property (nonatomic, strong) NSString *model;

@end

// Engine.m

@implementation Engine

- (instancetype)initWithModel:(NSString *)model {
    self = [super init];
    if (self) {
        [model retain];
        [model release];
        _model = model;
        NSLog(@"Model engine - %@", model);
    }
    return self;
}

- (void)dealloc {
    NSLog(@"Dealloc Enging - %@", _model);
    [_model release];
    [super dealloc];
}

@end
```

Как и у колеса, у двигателя существует собственный конструктор, который принимает на вход название модели и сохраняет в свойство **model**.

При инициализации захватывается название модели и присваивается свойству **model**. При вызове метода **dealloc** сообщение об этом выводится в консоль, и объект model освобождается.

После создания основных компонентов можно реализовать сам класс автомобиля:

```
// Car.h

@interface Car : NSObject

- (void)configWithEngine:(Engine *)engine andWheels:(NSArray *)wheels;

@property (nonatomic, strong) Engine *engine;
@property (nonatomic, strong) NSArray *wheels;

@end


// Car.m

@implementation Car

- (instancetype)init
{
    self = [super init];
    if (self) {
        NSLog(@"Create car");
    }
    return self;
}
- (void)configWithEngine:(Engine *)engine andWheels:(NSArray *)wheels {
    [engine retain];
    [engine release];
    _engine = engine;
    NSLog(@"Add engine for car");

    [wheels retain];
    [wheels release];
    _wheels = wheels;
    for (Wheel *wheel in wheels) {
        NSLog(@"Add wheel (%@) for car", wheel.number);
    }
}
- (void)remove {
    NSLog(@"Remove engine and wheels from car");
    for (Wheel *wheel in _wheels) {
        [wheel release];
    }
    [_wheels release];
    [_engine release];
}
- (void)dealloc {
    [self remove];
    NSLog(@"Dealloc car");
    [super dealloc];
}
@end
```

Как видно из реализации, объект автомобиля имеет метод для конфигурирования на основе двигателя и массива колес, а также имеет свойства для их хранения.

При инициализации выводится сообщение об этом.

При конфигурации захватываются объект двигателя и массив колес. Автомобиль становится их владельцем и может их освободить при необходимости. В консоль также выводится сообщение о добавлении необходимых компонентов в автомобиль.

Добавлен метод для удаления компонентов и их освобождения из памяти. Он вызывается в методе **dealloc**, чтобы у приложения не было утечек памяти.

Реализация **main**:

```
#import "Wheel.h"
#import "Engine.h"
#import "Car.h"

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

    Car *car = [[Car alloc] init];

    // Создание первого колеса
    Wheel *wheel1 = [[Wheel alloc] initWithNumber:@1];
    // Создание второго колеса
    Wheel *wheel2 = [[Wheel alloc] initWithNumber:@2];
    // Создание третьего колеса
    Wheel *wheel3 = [[Wheel alloc] initWithNumber:@3];
   // Создание четвертого колеса
    Wheel *wheel4 = [[Wheel alloc] initWithNumber:@4];

    // Создание массива колес
    NSArray *wheels = [[NSArray alloc] initWithObjects:wheel1, wheel2, wheel3,
wheel4, nil];
    // Создание двигателя
    Engine *engine = [[Engine alloc] initWithModel:@"Engine 1x"];

    // Конфигурация автомобиля с созданными компонентами
    [car configWithEngine:engine andWheels:wheels];

    // Освобождение автомобиля и удаление компонентов
    [car release];

    return 0;
}
```

Результат выполнения программы в консоли:

```
Create car
Create Wheel 1
Create Wheel 2
Create Wheel 3
Create Wheel 4
Model engine - Engine 1x
Add engine for car
Add wheel (1) for car
Add wheel (2) for car
Add wheel (3) for car
Add wheel (4) for car
Remove engine and wheels from car
Dealloc Wheel - 1
Dealloc Wheel - 2
Dealloc Wheel - 3
Dealloc Wheel - 4
Dealloc Enging - Engine 1x
Dealloc car
```

Так мы создали объекты автомобиля, четырех колес и двигателя. На их основе был сконфигурирован автомобиль, а при вызове **release** у него были освобождены все объекты, которые использовались без применения ARC и пула автоматического освобождения.

Практическое задание

1. Изменить созданный калькулятор из предыдущих уроков: внедрить ручное управление памятью.
2. Смоделировать и разработать программу «Стая птиц» (на основе практического задания) с применением ручного управления памятью.
3. *Улучшить созданную программу из задания 2: применить пул автоматического освобождения.

Дополнительные материалы

1. https://habrahabr.ru/post/265175/;
2. Стивен Кочан. «Программирование на Objective-C»;
3. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://habrahabr.ru/post/265175;
2. Стивен Кочан. «Программирование на Objective-C»;
3. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».