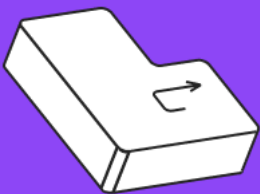




Знакомство с основами Objective-C

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



Оглавление

Концепции	2
Немного истории	3
Различия	4
Пространства имен	5
Синтаксические особенности	10
Типы данных	
Классификация базовых типов данных из языка C:	10
Создание переменных	13
Арифметические операции	14
Условные выражения и циклы	15
Приведение типов	23
Адреса и указатели	24
Что можно почитать еще?	28

Концепции

У любого желающего писать программы для продукции фирмы Apple в жизни наступает такой момент, когда ему приходится изучить новый язык программирования — Objective-C. На данной лекции мы собираемся углубиться в суть языка Objective – C и рассмотреть основные отличия Objective-C от Swift. В этом курсе вам предстоит рассмотреть все ключевые аспекты в разработке мобильных приложений на языке Objective – C, а именно основы синтаксиса языка, базовые паттерны в программирование, работе с многопоточной средой, запросах в сеть и Objective – C Runtime.

Немного истории

Objective-C возник в 80-х как модификация C в сторону Smalltalk. Причем модификация эта состояла в добавлении новых синтаксических конструкций и специальном препроцессоре для них (который, проходя по коду преобразовывают их в обычные вызовы функций C), а также библиотеке времени выполнения (Runtime). Таким образом, изначально Objective-C воспринимался как надстройка над C. В каком-то смысле это так и до сих пор: можно написать программу на чистом C, а после добавить к ней немного конструкций из Objective-C (при необходимости), или же наоборот, свободно пользоваться C в программах на Objective-C. Кроме того, все это касается и программ на C++. В 1988 NeXT (а в последствии Apple) лицензировала Objective-C и написала для него компилятор и стандартную библиотеку (по сути, SDK). В 1992 к усовершенствованию языка и компилятора подключились разработчики проекта GNU в рамках проекта OpenStep. С тех пор GCC поддерживает Objective-C. После покупки NeXT, Apple взяла их SDK (компилятор, библиотеки, IDE) за основу для своих дальнейших разработок. IDE для кода называли Xcode, а для GUI – Interface Builder. Фреймворк Cocoa для GUI разработок (и не только) на сегодня является наиболее значимой средой разработки программ на Objective-C. Самое важное, что нужно знать о Objective-C, это то, что это строгое надмножество C, языку, которому более 40 лет. Это означает, что действительный код C также является допустимым кодом Objective-C, и вы можете свободно смешивать и сочетать их. Вы даже можете использовать C++ с Objective-C, который обычно имеет прозвище Objective-C++, но это менее распространено. C и C++ несут с собой много багажа, и места, где встречаются C и Objective-C, немного грубоваты по краям, но это означает, что в Objective-C легко использовать код на основе C и C++.

Сразу очевидным недостатком являются заголовочные файлы: когда вы создаете класс в Objective-C, он состоит из YourClass.h (заголовочный файл) и YourClass.m (файл реализации). Первоначально «m» означало «сообщения», но сегодня большинство людей считают его файлом «Implementation». Ваш заголовочный файл описывает, что класс предоставляет внешнему миру: свойства, к которым можно получить доступ, и методы, которые можно вызвать. В вашем файле реализации вы пишете фактический код для этих методов.

Этого разделения между H и M нет в Swift, где весь класс или структура создается внутри одного файла. Но в Objective-C это важно: когда вы хотите использовать другой класс, компилятору достаточно прочитать H-файл, чтобы понять, как можно использовать этот класс. Это позволяет вам использовать компоненты с закрытым исходным кодом, такие как аналитическая библиотека Google: они предоставляют

вам файл `h`, который описывает, как работают их компоненты, и файл `«.a»`, который содержит их скомпилированный исходный код.

Вторым очевидным недостатком является препроцессор `C`. Препроцессор — это этап компиляции, который происходит до сборки кода `Objective-C`, что позволяет ему переписать ваш исходный код до его компиляции. Этого нет в `Swift`, и на то есть веская причина: основными причинами его использования являются файлы заголовков (которых нет в `Swift`) и создание макросов, которые представляют собой определения кода, которые заменяются при сборке вашего исходного кода. Например, вместо повторного написания `3.14159265359` вы можете создать макрос с именем `PI` и присвоить ему это значение — это немного похоже на константу в `Swift`, но, эти макросы могут делать гораздо больше.

Различия

Swift — гораздо более продвинутый язык, чем `Objective-C`, и поэтому имеет некоторые функции, которых просто нет в `Objective-C`.

В частности, `Objective-C` не поддерживает следующее:

- Вывод типа (Type inference).
- Перегрузка оператора (Operator overloading).
- Расширения протокола (Protocol extensions).
- Интерполяция строк (String interpolation).
- Пространства имен (Namespaces).
- Кортежи (Tuples).
- Опционалы (Optionals).
- Playgrounds.
- Конструкций `guard` and `defer`.
- Закрытых и полукоткрытых диапазонов.
- Перечисления со связанными значениями.

Структуры существуют в `Objective-C`, но используются гораздо реже, чем в `Swift`. `Objective-C`, как вы могли догадаться, учитывая его название, ориентирован на объекты.

Ранние версии `Swift` — от 1.0 до 2.2 — использовали почти те же соглашения об именах для методов и свойств, что и `Objective-C`. В `Swift 3.0` Apple представила Великое переименование `Cocoa`, которое включало переименование почти каждого

метода и свойства в «более Swifty», что приводило к тому, что почти каждый существующий проект ломался, пока он не был обновлен для использования новых соглашений об именах.

Вот что это означает в деталях:

1. В Objective-C первый параметр метода не имеет метки, поэтому метка для первого параметра обычно является частью имени метода. Например, `UIFont.preferredFont(forTextStyle:)` в Swift, а в Objective-C записывается как `[UIFont PreferredFontForTextStyle:]`.

2. Objective-C любит повторять названия вещей, чтобы сделать свои методы понятными. Это означает, что метод `append()` для `NSAttributedString` в Swift становится `appendAttributedString` в Objective-C, а компоненты (`separatedBy:`) строк становятся компонентами `SeparatedByString`.

3. Некоторые вещи, которые являются свойствами в Swift, являются методами в Objective-C. Например, `UIColor.blue` в Swift — это `[UIColor blueColor]` в Objective-C.

4. Многие имена классов должны иметь префикс «NS» перед ними в Objective-C. Например, `NSUserDefaults`, `NSFileManager`, `NSNotificationCenter` и `NSUUID`. В остальном они работают так же, как и их аналоги Swift.

Пространства имен

Отсутствие пространств имен в Objective-C поначалу может не иметь особого смысла, но оно имеет некоторое объяснение. Пространство имен — это способ сгруппировать функциональность в дискретные повторно используемые фрагменты. Когда вы распределяете пространство имен в своем коде, это гарантирует, что имена, которые вы используете для своих классов, не перекрываются с именами, которые использовали другие люди, потому что у вас есть дополнительный контекст. Например, вы можете создать класс с именем `Person` и не беспокоиться о том, что Apple создаст еще один класс с именем `Person`, потому что они не будут конфликтовать. Swift автоматически создает пространства имен в вашем коде, так что ваши классы автоматически помещаются внутри вашего модуля — `YourApp>YourClass`.

В Objective-C нет концепции пространств имен, а это означает, что все имена классов должны быть глобально уникальными. Это легче сказать, чем сделать: если вы используете пять библиотек, каждая из этих библиотек может использовать три другие библиотеки, и каждая библиотека может определять множество имен классов. Возможно, что и библиотека А, и библиотека В могут включать в себя

библиотеку C и, возможно, даже разные ее версии. Это создает множество проблем, и решение Apple простое и универсальное: используйте двух-, трех- или четырехбуквенные префиксы, чтобы сделать каждое имя класса уникальным. Подумайте об этом: UITableView, SKSpriteNode, MKMapView, XCTestCase — вы все время использовали этот префиксный подход, возможно, даже не осознавая, что он был разработан для устранения недостатка Objective-C.

Компилятор

У компилятора две задачи: преобразовать наш код на Objective-C в низкоуровневый код и проанализировать наш код, чтобы убедиться, что мы не сделали очевидных ошибок.

В наши дни Xcode поставляется с clang в качестве компилятора. Clang — это инструмент, который берет код Objective-C, анализирует его и преобразует в более низкоуровневое представление, напоминающее ассемблерный код: промежуточное представление — LLVM IR. LLVM — проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит. Состоит из набора компиляторов из языков высокого уровня, системы оптимизации, интерпретации и компиляции в машинный код. LLVM IR является низкоуровневым представлением кода и не зависит от операционной системы. LLVM берет инструкции и компилирует их в собственный байт-код IR(Intermediate Representation) для целевой платформы.

Польза LLVM заключается в том, что вы можете генерировать и запускать такие программные продукты на любой платформе, поддерживаемой LLVM. Например, если вы пишете свое приложение для iOS, оно автоматически запускается на двух очень разных архитектурах (Intel и ARM), и именно LLVM позаботится о переводе IR-кода в собственный байт-код для этих платформ.

При компиляции исходного файла компилятор проходит несколько этапов. Чтобы увидеть разные фазы, мы можем спросить clang, что он будет делать для компиляции файла hello.m :

```
% clang -ccc-print-phases hello.m
```

```
0: input, "hello.m", objective-c
```

```
1: preprocessor, {0}, objective-c-cpp-output
```

```
2: compiler, {1}, assembler
```

3: assembler, {2}, object

4: linker, {3}, image

5: bind-arch, "x86_64"

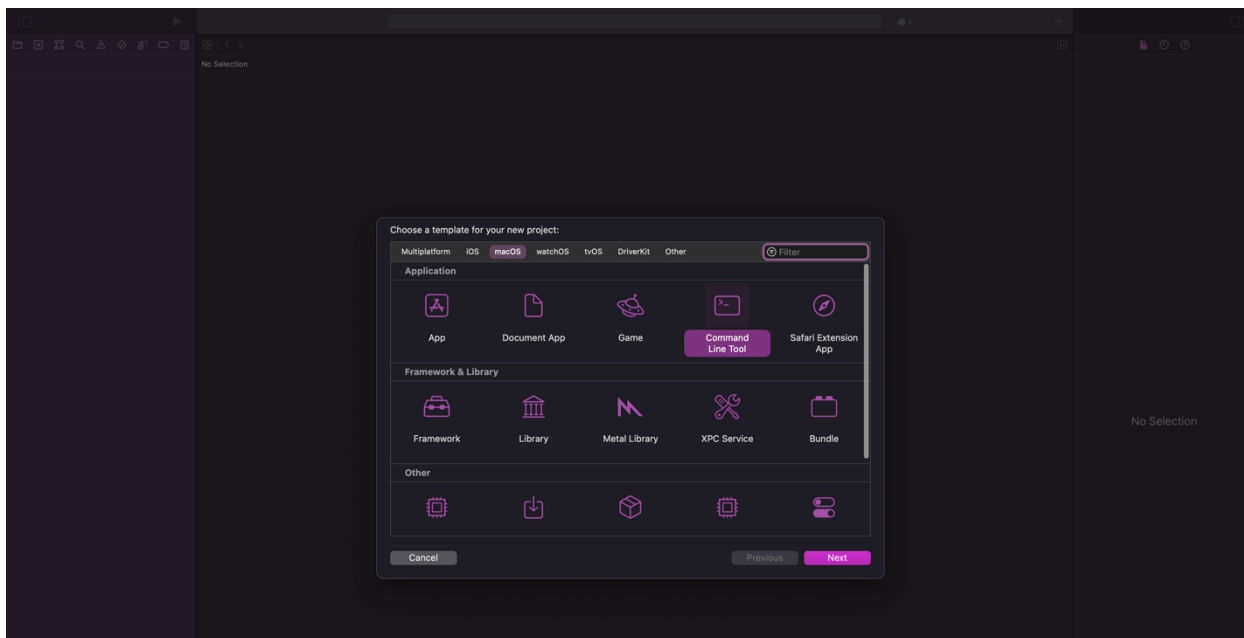
Итак, для разработки мобильных приложений мы будем использовать среду разработки – XCode. В данном ПО уже есть все готовые инструменты iOS SDK для разработки мобильных приложений и соответственно отдельно устанавливать ничего не придется. Данное ПО можно скачать с официального сайта компании Apple <https://xcodereleases.com/> или же через AppStore.

Основы синтаксиса Objective – C

Теперь рассмотрим некоторые базовые конструкции Objective-C:

1. Переменные;
2. Условные операторы;
3. Блоки переключения (switch) и циклы.

Создать новый проект в Xcode: перейдите в «Файл» > «Новый проект», затем выберите «macOS» > «Приложение» с левой стороны и «Инструмент командной строки».



При создании проекта из этого шаблона вам будет предоставлен только один файл: main.m. Внутри всего несколько строк кода, но даже они вводят несколько основных концепций. Вы должны увидеть что-то вроде этого:



```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"Hello World");
    }
    return 0;
}
```

Если у вас есть некоторый опыт работы с C, вы уже узнаете большую часть этого кода, как обычную точку входа для приложений командной строки. Но есть две части, которые уникальны для Objective-C: `@autoreleasepool` и `@"Hello, World!"`, разберемся с этими конструкциями дальше.

Рассмотрим пример вывода строки в лог консоли:

```
NSLog(@"Hello World");
```

Эта строка кода выведет строку «Hello World» на консоль. Функция **NSLog()** является аналогом **printf()** в языке C. **NSLog()** принимает строку как первый аргумент, а также может применять другие аргументы для вывода дополнительных элементов.

```
NSLog(@"Hello %@", @"World");
```

Результатом станет также «Hello World» в консоли. Таких дополнительных элементов может быть столько, сколько необходимо. Но для каждого их них в строке необходимо поставить специальный символ, соответствующий типу этого элемента (см. перечень ниже). С такими символами придется встречаться довольно часто, и по ходу курса они постепенно запомнятся.

Список спецификаторов формата NSString

Specifier	Description
%@	Objective-C object, printed as the string returned by <code>descriptionWithLocale:</code> if available, or <code>description</code> otherwise. Also works with <code>CTypeRef</code> objects, returning the result of the <code>CFCopyDescription</code> function.
%%	'%' character.
%d, %D	Signed 32-bit integer (<code>int</code>).
%u, %U	Unsigned 32-bit integer (<code>unsigned int</code>).
%x	Unsigned 32-bit integer (<code>unsigned int</code>), printed in hexadecimal using the digits 0-9 and lowercase a-f.
%X	Unsigned 32-bit integer (<code>unsigned int</code>), printed in hexadecimal using the digits 0-9 and uppercase A-F.
%o, %O	Unsigned 32-bit integer (<code>unsigned int</code>), printed in octal.
%f	64-bit floating-point number (<code>double</code>).
%e	64-bit floating-point number (<code>double</code>), printed in scientific notation using a lowercase e to introduce the exponent.
%E	64-bit floating-point number (<code>double</code>), printed in scientific notation using an uppercase E to introduce the exponent.
%g	64-bit floating-point number (<code>double</code>), printed in the style of %e if the exponent is less than -4 or greater than or equal to the precision, in the style of %f otherwise.
%G	64-bit floating-point number (<code>double</code>), printed in the style of %E if the exponent is less than -4 or greater than or equal to the precision, in the style of %f otherwise.
%c	8-bit unsigned character (<code>unsigned char</code>).
%C	16-bit UTF-16 code unit (<code>unichar</code>).
%s	Null-terminated array of 8-bit unsigned characters. Because the %s specifier causes the characters to be interpreted in the system default encoding, the results can be variable, especially with right-to-left languages. For example, with RTL, %s inserts direction markers when the characters are not strongly directional. For this reason, it's best to avoid %s and specify encodings explicitly.
%S	Null-terminated array of 16-bit UTF-16 code units.
%p	Void pointer (<code>void *</code>), printed in hexadecimal with the digits 0-9 and lowercase a-f, with a leading 0x.
%a	64-bit floating-point number (<code>double</code>), printed in scientific notation with a leading 0x and one hexadecimal digit before the decimal point using a lowercase p to introduce the exponent.
%A	64-bit floating-point number (<code>double</code>), printed in scientific notation with a leading 0X and one hexadecimal digit before the decimal point using an uppercase P to introduce the exponent.
%F	64-bit floating-point number (<code>double</code>), printed in decimal notation.

Вместо **NSLog** можно использовать вариант из языка C – **printf()**. Но **NSLog** предпочтительнее, так как позволяет выводить в лог консоли объекты Objective-C. Кроме этого, при вызове этой функции указывается дата и время, добавляется символ окончания строки – **\n**.

@autoreleasepool означает «Я собираюсь выделить много памяти; когда я закончу, пожалуйста, освободи его». Все, что находится внутри открывающей и закрывающей фигурных скобок, является частью этого пула, который сейчас представляет собой всю программу.

Стоит также кратко упомянуть код C, в частности то, как написана функция.

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

И вот что означает каждая вещь:

- `int`: эта функция возвращает целое число.
- `main`: Функция называется `main()`.
- `int argc`: первый параметр представляет собой целое число с именем `argc`.
- `const char * argv[]`: Второй параметр представляет собой массив строк с именем `argv`.

Эта функция `main()` с этим параметром является стандартным способом создания программ командной строки, и она будет автоматически вызываться при запуске

программы.

Еще несколько мелочей, прежде чем мы двинемся дальше.

Во-первых, обратите внимание, что `return` используется для возврата значения из функции, как и в Swift.

Во-вторых, каждое выражение должно заканчиваться точкой с запятой. В нашем коде это означает `NSLog()` и возвращает оба конца с точкой с запятой.

В-третьих, `//` — это комментарий, как и в Swift.

Синтаксические особенности

В этом курсе вам предстоит познакомиться с синтаксическими особенностями Objective-C, которые на первых порах могут быть непривычными. В примере вывода в консоль строки можно было заметить, что перед строкой стоит символ «@».

Как и в C, а конце строки ставится двоеточие, чтобы компилятор однозначно смог определить завершение строки кода.

Еще одна синтаксическая особенность Objective-C — отправка сообщений (обращение к методу объекта). Для этого в квадратных скобках указывается объект и через пробел — его метод:

```
[object method];
```

Типы данных

Классификация базовых типов данных из языка C:

Тип	Размер	Диапазон значений
char	1 байт	От -127 до 127
bool	1 байт	true, false
short int	2 байта	От -32767 до 32767

unsigned short int	2 байта	От 0 до 65535
int	4 байта	От -32767 до 32767
unsigned int	4 байта	От 0 до 65535
long int	4 байта	От -2147483647 до 2147483647
unsigned long int	4 байта	От 0 до 4294967295
float	4 байта	От $1E-37$ до $1E+37$ с точностью не менее 6 значащих десятичных цифр
double	8 байт	От $1E-37$ до $1E+37$ с точностью не менее 10 значащих десятичных цифр
long double	10 байт	От $1E-37$ до $1E+37$ с точностью не менее 10 значащих десятичных цифр

Пример использования примитивных типов данных

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int intValue = 10;
        char *charValue = "s";
        bool boolValue = false;
        float floatValue = 1.2;
        double doubleValue = 2.3;

        BOOL boolObjc = YES;
        NSInteger integer = 3;
        CGFloat cgFloat = 3.1;
        NSNumber *number = @1;
        NSString *string = @"Hello";

        NSLog(@"%d", intValue);           // 10
        NSLog(@"%s", charValue);          // s
        NSLog(@"%d", boolValue);          // 0
        NSLog(@"%f", floatValue);         // 1.200000
        NSLog(@"%f", doubleValue);        // 2.300000
        NSLog(@"%d", boolObjc);           // 1
        NSLog(@"%ld", (long)integer);      // 3
        NSLog(@"%f", cgFloat);             // 3.100000
        NSLog(@"%@", number);              // 1
        NSLog(@"%@", string);              // Hello
    }
    return 0;
}
```


Базовые типы Objective-C

Тип	Значение
BOOL	YES, NO
NSInteger	Целое число
CGFloat	Число с плавающей точкой
NSNumber	Объект, числового значения
NSString	Строка (@"Hello")
NSMutableString (Изменяемая версия NSString)	Строка

Создание переменных

Objective-C не поддерживает вывод типов (type inference), и, в отличие от Swift, почти все создается как переменная, как можно было заметить из примера выше. На практике это означает, что вам нужно сообщить Xcode тип каждой части данных, с которой вы хотите работать.

1. **BOOL** хранит значение истинности и может выражаться только двумя состояниями: YES и NO.
2. **NSInteger** «отвечает» за целые числа (похож на примитивный тип int).
3. **CGFloat** хранит числа с плавающей точкой. Этот тип часто применяется для отображения размера и положения объектов на экране устройства (но это не единственное его применение).
4. **NSNumber** представляет числа (как целые, так и с плавающей запятой) в виде объекта. Одним из многочисленных применений для этого типа является представление чисел в массиве. Objective-C позволяет хранить в массивах только объекты, так что для добавления числа необходимо преобразовать его. Рассмотрим создание **NSNumber**:



```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSInteger integer = 3;
        // Создание NSNumber из NSInteger
        NSNumber *integerNumber = [NSNumber numberWithInt:integer];
        // Создание NSNumber из BOOL
        NSNumber *boolNumber = [NSNumber numberWithBool:NO];
        // Создание NSNumber, используя литерал
        NSNumber *number = @1;
        // 3, 0, 1
        NSLog(@"%@, %@, %@", integerNumber, boolNumber, number);
    }
    return 0;
}
```

Таким образом в **NSNumber** преобразуются **NSInteger**, **BOOL**, примитивные типы. Также можно создавать объект, используя литерал.

NSString — это объект, который представляет строку. Рассмотрим пример создания строки:

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSNumber *number = @1;
        NSString *str = [NSString stringWithFormat:@"%d", number];
        NSString *anotherStr = @"Hello";

        NSLog(@"%@, %@", str, anotherStr);
    }
    return 0;
}
```

Объект **NSString** можно создать, используя различные типы данных, применяя спецификаторы формата. Вариант с использованием литерала тоже применим.

Арифметические операции

Над переменными можно совершать арифметические операции: сложение, вычитание, умножение, деление и получение остатка от деления.

Пример:

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int number1 = 10 + 15; // 25
        int number2 = 15 - 10; // 5
        int number3 = 10 * 15; // 150
        int number4 = 10 / 15; // 0
        int number5 = 10 % 2; // 0
    }
    return 0;
}
```

Также в Objective-C присутствуют операторы инкремента и декремента, позволяющие увеличить или уменьшить число на 1. Для этого применяется следующая конструкция:

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int a = 0;
        int b = 1;
        b--;
        a++;
        NSLog(@"%d, %d", a, b);
    }
    return 0;
}
```

В результате выполнения переменная *a* будет иметь значение 1, а переменная *b* = 0.

Для увеличения или уменьшения числа можно применять конструкции вида: +=, -=, *=, /=.

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int a = 0;
        int b = 1;
        b -= 2;
        a += 3;
        NSLog(@"%d, %d", a, b);
    }
    return 0;
}
```

В результате переменная *a* будет равна 3, а переменная *b* примет значение -1.

Условные выражения и циклы

- **Оператор if**

Условные операторы в основном работают так же, как и в Swift, хотя вы всегда должны заключать свои условия в скобки. Эти круглые скобки, как и точка с запятой в конце строки, часто случайно пропускаются, когда вы переходите из Swift, но Xcode откажется компилироваться, пока вы это не исправите.

Пример условного оператора:

```
int i = 10;
if (i == 10) {
    NSLog(@"Привет, мир!");
}
```

Однако в Objective-C есть один нюанс, который одновременно вводит целый новый класс ошибок и экономит пару нажатий клавиш: если содержимое вашего условного оператора — это всего лишь одно выражение, вы можете опустить фигурные скобки. Например, эти два фрагмента кода делают одно и то же:

```
if (i == 10) {
    NSLog(@"Привет, мир!");
} else {
    NSLog(@"До свидания!");
}

if (i == 10)
    NSLog(@"Привет, мир!");
else {
    NSLog(@"До свидания!");
}
```

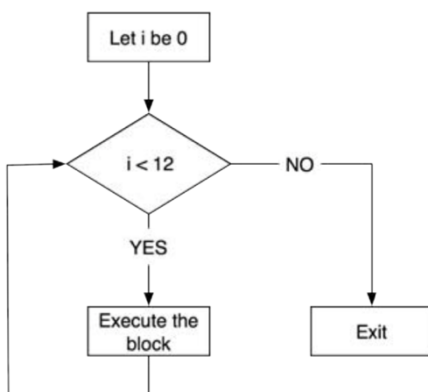
Поскольку вы только начинаете изучать Objective-C, я бы посоветовал вам воздержаться от последнего варианта. Если вы отчаянно хотите избежать фигурных скобок, по крайней мере напишите оператор if в одной строке, например:

```
if (i == 10) NSLog(@"Hello, World!");
```

Преимущества этого синтаксиса в лучшем случае сомнительны.

• Цикл While

Наше знакомство с циклами начнется с цикла while. Конструкция while отдаленно напоминает конструкцию if. Она тоже состоит из выражения и блока кода, заключенного в фигурные скобки. В конструкции if в случае истинности выражения блок кода выполняется только один раз. В конструкции while блок выполняется снова и снова, пока выражение не станет равно false.



Блок схема цикла While



```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i = 0;
    while (i < 12) {
        NSLog("%d. Aaron is Cool\n", i);
        i++;
    }
    return 0;
}
```

Пример цикла while

Результат:

- 0. Aaron is Cool
- 1. Aaron is Cool
- 2. Aaron is Cool
- 3. Aaron is Cool
- 4. Aaron is Cool
- 5. Aaron is Cool
- 6. Aaron is Cool
- 7. Aaron is Cool
- 8. Aaron is Cool
- 9. Aaron is Cool
- 10. Aaron is Cool
- 11. Aaron is Cool

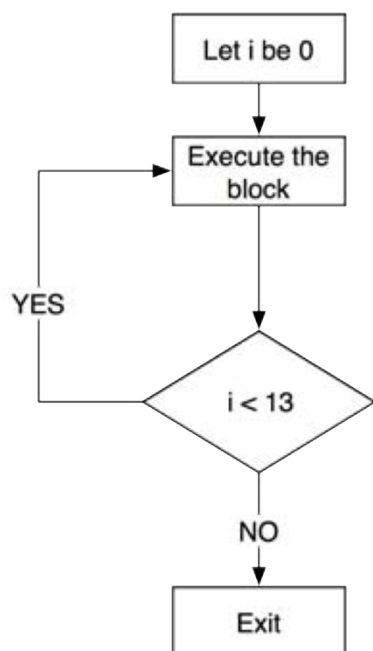
- **Цикл do-while**

Опытные разработчики стараются не использовать цикл do-while, но для полноты картины следует упомянуть и его. Цикл do-while не проверяет выражение, пока блок не будет выполнен. Таким образом, блок всегда будет выполнен хотя бы один раз. Если переписать исходную программу так, чтобы в ней использовался цикл do - while, она будет выглядеть так:

```
{  
    int i = 0;  
    do {  
        printf("%d. Aaron is Cool\n", i);  
        i++;  
    } while (i < 13);  
    return 0;  
}
```

Обратите внимание на завершающий символ «;». Дело в том, что, в отличие от других циклов, цикл do-while представляет собой одну длинную команду:

do { что – то делаем } while (пока условие остается истинным); А вот как выглядит блок-схема цикла do-while:



- **Цикл For**

Objective-C имеет полный набор параметров цикла, включая цикл for в стиле C, который устарел в Swift 2.2. Начнем с наиболее распространенного типа цикла, известного как быстрое перечисление:

```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    NSArray *names = @"Laura", @"Janet", @"Kim";
    for (NSString *name in names) {
        NSLog(@"Hello, %@", name);
    }
    return 0;
}
```

Этот фрагмент кода создает массив имен, затем перебирает каждое из них и печатает приветствие. Синтаксис NSLog() может показаться на первый взгляд особенно странным, но это результат того, что в Objective-C нет интерполяции строк. NSLog() является функцией с переменным числом аргументов и объединяет строку в своем первом параметре со значениями второго и последующих параметров.

Вы можете использовать while точно так же, как в Swift, а do/while идентично конструкции Repeat/while в Swift.

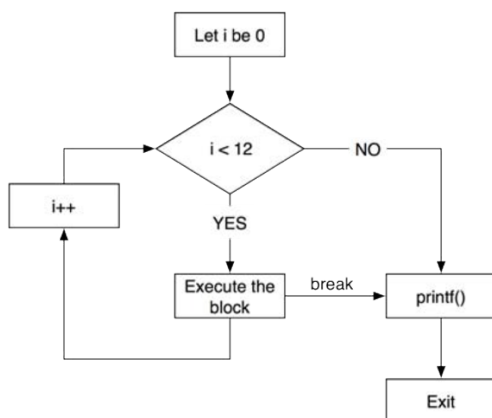
```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    for (int i = 1; i <= 5; ++i) {
        NSLog(@"%d * %d равно %d", i, i, i * i);
    }
    return 0;
}
```

Как и в случае с условиями, вы можете опускать фигурные скобки в циклах, если тело цикла содержит только один оператор. Это имеет те же сомнительные затраты/выгоды, поэтому используйте его с осторожностью.

Инструкция break

Иногда бывает нужно прервать выполнение цикла изнутри. Предположим, мы хотим перебрать все положительные числа от 0 до 11 в поисках такого числа x , для которого выполняется условие $x + 90 = x^2$. План действий: перебираем целые числа от 0 до 11 и при обнаружении нужного числа прерываем выполнение цикла.

```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i;
    for (i = 0; i < 12; i++) {
        printf("Checking i = %d\n", i);
        if (i + 90 == i * i) {
            break;
        }
    }
    printf("The answer is %d.\n", i);
    return 0;
}
```



Результат:

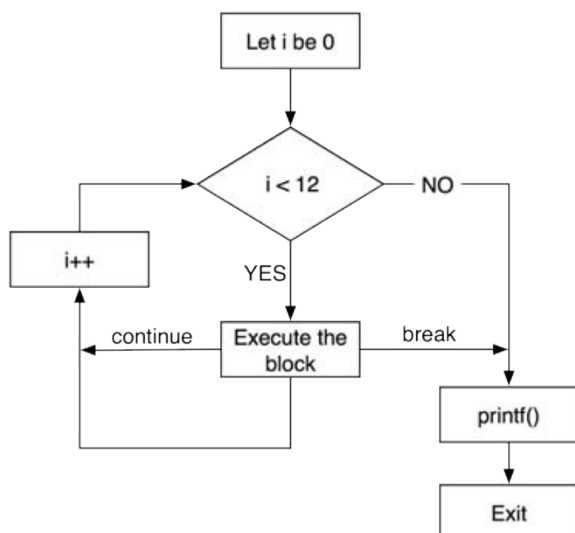
Checking i = 0
Checking i = 1
Checking i = 2
Checking i = 3
Checking i = 4
Checking i = 5
Checking i = 6
Checking i = 7
Checking i = 8
Checking i = 9
Checking i = 10

The answer is 10.

Инструкция continue

Иногда во время выполнения блока в цикле нужно сказать программе. «А теперь пропусти все, что осталось выполнить в блоке, и начини следующий проход». Эта задача решается командой `continue`. Допустим, вы твердо уверены в том, что для чисел, кратных 3, условие никогда не выполняется. Как избежать напрасной потери времени на их проверку?

```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i;
    for (i = 0; i < 12; i++) {
        if (i % 3 == 0) {
            continue;
        }
        printf("Checking i = %d\n", i);
        if (i + 90 == i * i) {
            break;
        }
    }
    printf("The answer is %d.\n", i);
    return 0;
}
```



Результат:

Checking i = 1


Checking i = 2

Checking i = 4
Checking i = 5
Checking i = 7
Checking i = 8
Checking i = 10
The answer is 10.

- **Switch case оператор**

В Objective-C менее мощен, чем Swift, поэтому вам нужно проделать больше работы самостоятельно, а во-вторых, операторы case имеют неявные отказы. Это противоположность Swift и означает, что вы почти всегда хотите написать break в конце блоков case, чтобы избежать провала.

Базовый пример:



```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i = 20;
    switch (i) {
        case 20:
            NSLog(@"It's 20!");
            break;
        case 40:
            NSLog(@"It's 40!");
            break;
        case 60:
            NSLog(@"It's 60!");
            break;
        default:
            NSLog(@"It's something else.");
    }
    return 0;
}
```

Обратите внимание на круглые скобки для переключателя (i). Запустите это сейчас, и вы должны увидеть «It's 20!» распечатывается, но попробуйте убрать разрыв; операторы, и вы поймете, что я имею в виду под неявным провалом: он напечатает «It's 20!» затем «Это 40!», «Это 60!» и «Это что-то другое». один за другим. В Objective-C отсутствие break эквивалентно добавлению fallthrough в Swift. В Objective-C есть поддержка сопоставления с образцом, но она ограничена диапазоном: вы пишете одно число, затем ... с пробелом с обеих сторон, затем другое число, например:

```

#include <stdio.h>
int main(int argc, const char * argv[]) {
    switch (i) {
        case 1 ... 30:
            NSLog(@"It's between 1 and 30!");
            break;
        default:
            NSLog(@"It's something else.");
    }
    return 0;
}

```

Приведение типов

Чтобы получить переменную определенного типа из другого, можно воспользоваться приведением типов.

Допустим, что нам дано десятичное число, но необходимо, чтобы результат был в виде целого числа. Необходимо провести преобразование:

```

#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        double value = 1.2;
        int number = (int)value;
        NSLog(@"%d", number);
    }
    return 0;
}

```

В результате переменная ***number*** будет включать в себя целое число со значением 1.

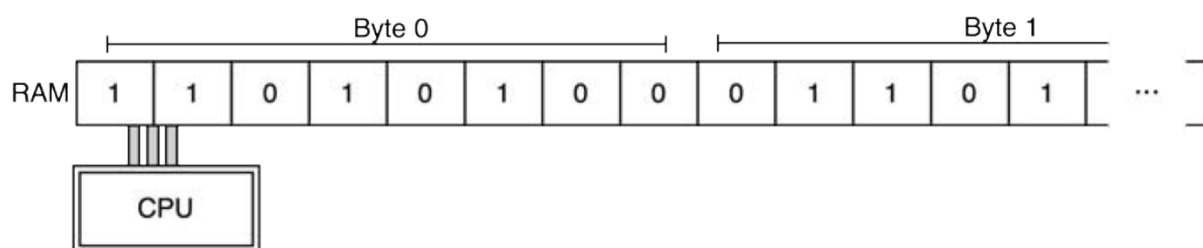
В Objective-C существует и автоматическое приведение типов для любых числовых типов данных, кроме объектов. При присваивании значение иного типа будет приведено к типу переменной.

При сложении двух чисел разных типов их сумма будет соответствовать типу одного из них. Например, при сложении числа с плавающей точкой и целого результат

будет приведет к числу с плавающей точкой. Математические операторы для приводимых типов и объектов применять нельзя.

Адреса и указатели

В сущности, ваш компьютер состоит из процессора и оперативной памяти — огромного набора “переключателей”, которые могут включаться и выключаться процессором. Условно каждый переключатель хранит 1 бит информации. Значение 1 обычно представляет «включенное» состояние, а значение 0 — «выключенное». Восемь “переключателей” образуют один *байт* информации. Процессор может получить состояние “переключателей”, выполнить операции с битами и сохранить результат в другом наборе “переключателей”. Например, процессор может прочитать байт из одного места, прочитать другой байт из другого места, сложить прочитанные значения и сохранить полученный байт в третьем месте.



Все ячейки памяти пронумерованы. Номер байта обычно называется его *адресом*. и когда речь идет о 32-разрядных или 64-разрядных процессорах, имеется в виду величина адресов, используемых этими процессорами. 64-разрядный процессор способен работать с памятью намного, намного большего объема, чем 32-разрядный.

- **Получение адресов**

Для получения адреса переменной используется оператор & амперсанд:

```
#include <stdio.h>
int main(int argc, const char * argv[]) {
    int i = 17;
    printf("i stores its value at %p\n", &i);
    return 0;
}
```

Обратите внимание на заполнитель %p — он используется при выводе адресов памяти. Постройте и запустите программу результат будет выглядеть примерно так:

`i` stores its value at `0xbffff738`, впрочем, ваш компьютер может разместить значение `i` по совершенно иному адресу. Адреса памяти почти всегда выводятся в шестнадцатеричном формате. На компьютере все данные хранятся в памяти, а следовательно, имеют адрес. Например, функция тоже хранится в памяти, начиная с некоторого адреса. Для вывода начального адреса функции достаточно указать ее имя:

```
int main(int argc, const char * argv[]) {
    int i = 17;
    printf("i stores its value at %p\n", &i);
    printf("this function starts at %p\n", main);
    return 0;
}
```

• Хранение адресов в указателях

А если адрес потребуется сохранить у переменной? Конечно, можно воспользоваться без знаковой целочисленной переменной подходящего размера, но если тип переменной будет указан более точно, компилятор поможет вам и укажет на возможные ошибки в программе. Например, если в переменной с именем `ptr` должен храниться адрес, по которому содержится значение типа `float`, ее объявление может выглядеть так:

```
float *ptr;
```

В этой строке мы сообщаем, что переменная `ptr` является указателем на `float`. В ней не хранится значение типа `float`: она содержит адрес, по которому может храниться значение `float`.

Объявите новую переменную `addressOfI`, содержащую указатель на тип `int`. Присвойте ей адрес `i`.

```
int main(int argc, const char * argv[]) {
    int i = 17;
    int *addressOfI = &i;

    printf("i stores its value at %p\n", addressOfI);
    printf("this function starts at %p\n", main); return 0;
}
```

Постройте и запустите программу. В ее повелении ровным счетом ничего не изменилось. Сейчас для простоты мы используем целые числа. Но если вас интересует, для чего нужны указатели, я вас отлично понимаю. Передать целое значение, присвоенное переменной, ничуть не сложнее, чем ее адрес. Однако скоро бы начнете работать с данными, которые намного сложнее и больше отдельных целых чисел. Именно этим так удобны адреса: мы не всегда можем передать копию данных, с которыми работаем, но зато всегда можем передать *адрес*, по которому эти данные начинаются в памяти. А при наличии адреса обратиться к данным уже несложно.

- **Обращение к данным по адресу**

Если вам известен адрес данных, для обращения к самим данным можно воспользоваться оператором *. Следующая программа выводит на консоль значение целочисленной переменной, хранящейся по адресу addressOfI:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI); printf("this
function starts at %p\n", main);
    printf("the int stored at addressOfI is %d\n", *addressOfI);
return 0;
}
```

Обратите внимание: звездочка (*) здесь используется двумя разными способами.

Во-первых, переменная addressOfI объявляется с типом int*. Иначе говоря, объявляемая переменная является указателем на область памяти, в которой может храниться значение int.

Во-вторых, звездочка используется при чтении значения int, которое хранится по адресу, находящемуся addressOfI. Указатели также иногда называются *ссылками*, а использование указателя для чтения данных по адресу — *разыменованием* (dereferencing) указателя.

Оператор * также может использоваться в левой части команды присваивания для сохранения данных по указанному адресу:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89; printf("Now i is %d\n", i); return 0;
}
```

- **Размер в байтах**

Итак, теперь мы знаем, что все данные хранятся в памяти, и умеем получать адрес, начиная с которого размещаются данные. Но как определить, сколько байт занимает тип данных?

Для определения размера типа данных используется функция `sizeof()`.

Пример:

```
int main(int argc, const char * argv[]) {
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89;
    printf("Now i is %d\n", i);
    printf("An int is %zu bytes\n", sizeof(int));
    printf("A pointer is %zu bytes\n", sizeof(int *));
    return 0;
}
```

В вызовах `printf()` встречается новый заполнитель `%zu`. Функция `sizeof()` возвращает значение типа `size_t`, для вывода которого следует использовать относительно редкий заполнитель `%zu`.

Запустите программу. Если размер указателя составляет 4 байта, ваша программа выполняется в 32-разрядном режиме. Если же указатель занимает 8 байт, программа выполняется в 64-разрядном режиме.

В аргументе функции `sizeof()` также может передаваться переменная, так что приведенная выше программа может быть записана в таком виде:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89;
    printf("Now i is %d\n", i);
    printf("An int is %zu bytes\n", sizeof(i));
    printf("A pointer is %zu bytes\n", sizeof(addressOfI)); return
0;
}
```

- **NULL**

Иногда в программе бывает нужно создать указатель «на ничто» - то есть переменную для хранения адреса, которая содержит значение, однозначно показывающее, что этой переменной еще не было присвоено определенное значение, для этой цели используется значение NULL:

```
float *myPointer;
// Сейчас переменной myPointer присваивается значение NULL,
// позднее в ней будет сохранен реальный указатель.
myPointer = NULL;
Что такое NULL? Вспомните, что адрес - всего лишь число.
Обозначению NULL соответствует ноль. Это очень удобно в
конструкциях вида:
float *myPointer;
// Переменной myPointer было присвоено значение?
if (!myPointer) {
    // Значение myPointer отлично от NULL
    ... Работаем с данными, на которые ссылается myPointer ...
} else {
    // Значение myPointer равно NULL
}
```

Позднее, когда речь пойдет об указателях на объекты, вместо NULL будет использоваться обозначение nil. Эти обозначения эквивалентны, но программисты Objective-C используют nil для обозначения адресов, по которым не хранятся объекты.

Что можно почитать еще?

1. <https://llvm.org/docs/LangRef.html>
2. Стивен Кочан. «Программирование на Objective-C».
3. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».
4. <https://www.objc.io/>
5. <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/>