Глава 9

Управление памятью

Следующая тема — это управление памятью с использованием языка Objective-C и среды Cocoa (правда, интересно?). Управление памятью — это только часть общей проблемы, именуемой *управление ресурсами*. Каждая компьютерная система может предоставить программе для выполнения только весьма ограниченные ресурсы, которые включают память, открытые файлы, сетевые соединения и т.п. Если вы используете ресурсы, например, путем открытия файла, то после работы должны прибраться (в данном случае закрыть файл). Если вы будете оставлять файлы открытыми и никогда не будете их закрывать, то в конечном счете исчерпаете этот ресурс и не сможете продолжать работу. Подумайте о библиотеках — если бы каждый брал в них книги и никогда не возвращал их, надолго ли хватило бы этих библиотек?

Конечно, когда программа завершает работу, операционная система освобождает выделенные ей ресурсы. Но пока ваша программа работает и использует ресурсы, а вы пренебрегаете их освобождением, они в конце концов закончатся и программа завершится аварийно. По мере развития операционных систем момент исчерпания ресурсов становится все более неопределенным.

Не каждая программа нуждается в сетевых подключениях, но нет такой программы, которая не использовала бы память. Ошибки, связанные с памятью, — сущее наказание для программистов, работающих с C-образными языками программирования. Нашим коллегам, работающим на Java и языках сценариев, гораздо проще: управление памятью у них автоматизировано; можно сказать, что родители прибирают их комнаты. Мы же сами выделяем память, когда она нужна, и освобождаем, когда выполнили с ней все действия. Если мы выделяем память, но не освобождаем ее — получаем *утечку памяти*: потребление памяти программой растет и растет, и в конечном итоге программа аварийно завершается. С другой стороны, надо быть не менее осторожным и не использовать уже освобожденную память, так как можно воспользоваться утратившими актуальность данными и получить самые разнообразные и очень трудно обнаруживаемые ошибки.

ПРИМЕЧАНИЕ. Управление памятью — сложная задача. Решение среды Cocoa скорее элегантно, но требует умственных усилий, чтобы разобраться в нем. Даже программисты с большим стажем, впервые столкнувшись с данным материалом, испытывают определенные затруднения, так что не беспокойтесь, если после прочтения данной главы ваша голова будет некоторое время кружиться…

Жизненный цикл объекта

Подобно любому живому существу в реальном мире, объекты в программе имеют свой жизненный цикл. Они рождаются (при помощи методов `alloc` или `new`), живут (получают сообщения и реагируют на них), заводят друзей (путем композиции и аргументов методов) и в конце концов умирают (освобождают память). Когда это происходит, их тела (память) используются новым поколением объектов.

Подсчет ссылок

Теперь, когда мы знаем о том, когда объект рождается, и немало поговорили о том, как он живет, т.е. как его использовать в программе, пора задаться вопросом: как узнать, что жизнь объекта подошла к концу? В среде Cocoa используется методика, известная как *подсчет ссылок* (reference counting). Каждый объект содержит связанную с ним целочисленную переменную, известную как *счетчик ссылок*. Когда некоторый фрагмент кода работает с объектом, он увеличивает его счетчик ссылок, говоря: "Я работаю с этим объектом". Когда работа завершена, код уменьшает счетчик ссылок, указывая, что ему объект больше не нужен. Когда счетчик становится равен нулю, это означает, что объект больше никому не нужен (бедняжка!), так что он уничтожается, а занятая им память возвращается системе для повторного использования.

Когда объект создается при помощи методов `alloc` или `new` или в ответ на сообщение `copy` (которое приводит к созданию копии объекта-получателя), счетчик объекта устанавливается равным 1. Для увеличения счетчика ссылок следует послать объекту сообщение `retain`, а для уменьшения — сообщение `release`.

Когда объект должен быть уничтожен из-за обнуления его счетчика ссылок, система выполнения программ на языке Objective-C автоматически отправляет объекту сообщение `dealloc`. Вы можете заместить метод `dealloc` в ваших объектах для освобождения всех ресурсов, которые могли захватить. Никогда не вызывайте метод `dealloc` непосредственно — положитесь в этом на систему выполнения программ на языке Objective-C, которая вызывает метод `dealloc`, когда наступает время убить ваш объект.

Чтобы узнать текущее состояние счетчика ссылок, пошлите объекту сообщение `retainCount`. Вот как выглядят сигнатуры упомянутых методов `retain`, `release` и `retainCount`:

```
- (id) retain;
- (void) release;
- (unsigned) retainCount;
```

Метод `retain` возвращает идентификатор `id`, так что вы можете встроить этот вызов в цепочку, сперва увеличивая счетчик ссылок, а затем запрашивая от объекта некоторые действия. Например, `[[car retain] setTire: tire atIndex: 2];` требует от `car` увеличить счетчик ссылок и выполнить действие `setTire:`.

Первый проект данной главы — `RetainCount1` — расположен в папке `09.01 RetainCount-1`. Эта программа создает объект (`RetainTracker`), который вызывает функцию `NSLog()` при инициализации и уничтожении.

```
@interface RetainTracker : NSObject
@end // RetainTracker

@implementation RetainTracker
- (id) init
{
 if (self = [super init]) {
     NSLog (@"init: счетчик равен %d.",
            [self retainCount]);
 }
 return (self);
} // init

- (void) dealloc
{
 NSLog (@"Вызван dealloc. Прощайте!");
 [super dealloc];
} // dealloc
@end // RetainTracker
```

Метод `init` следует стандартной идиоме Cocoa для инициализации объектов, которую мы рассмотрим в следующей главе. Как упоминалось ранее, сообщение `dealloc` посылается автоматически (и в результате вызывается метод `dealloc`), когда счетчик ссылок обнуляется. Наши версии `init` и `dealloc` используют функцию `NSLog()` для вывода сообщений, информирующих об их вызовах.

В функции `main()` создается новый объект `RetainTracker`, при этом описанные выше два метода вызываются неявно. После создания нового объекта `RetainTracker` для увеличения и уменьшения счетчика ссылок посылаются сообщения `retain` и `release`, причем в промежутках между вызовами значение счетчика отслеживается и выводится при помощи функции `NSLog()`.

```
int main (int argc, const char *argv[])
{
 RetainTracker *tracker = [RetainTracker new];
 // count: 1

 [tracker retain]; // count: 2
 NSLog (@"%d", [tracker retainCount]);
```

```
    [tracker retain]; // count: 3
    NSLog (@"%d", [tracker retainCount]);

    [tracker release]; // count: 2
    NSLog (@"%d", [tracker retainCount]);

    [tracker release]; // count: 1
    NSLog (@"%d", [tracker retainCount]);

    [tracker retain]; // count 2
    NSLog (@"%d", [tracker retainCount]);

    [tracker release]; // count 1
    NSLog (@"%d", [tracker retainCount]);

    [tracker release]; // count: 0, вызов dealloc
    return (0);
} // main
```

В реальной программе, конечно, вы не будете выполнять неоднократные увеличения и уменьшения счетчика ссылок, как в этой простой функции. В процессе жизненного цикла объекта он может столкнуться с последовательностями увеличений и уменьшений счетчика ссылок, подобными приведенной, но исходящими из разных мест. Наша же программа позволяет посмотреть на изменения счетчика ссылок в процессе работы.

```
init: счетчик равен 1.
2
3
2
1
2
1
Вызван dealloc. Прощайте!
```

Так что если вы создаете объект (при помощи методов `alloc`, `new` или `copy`), то должны по окончании работы с ним освободить его, чтобы ненужный объект был уничтожен, а использованная им память была возвращена системе.

Владение объектом

“Ну и что тут сложного? — можете подумать вы. — В чем проблема? Создаем объект, пользуемся им, освобождаем, и система управления памятью довольна. Не вижу никаких сложностей!” Но это еще не конец, и все станет более сложным при рассмотрении концепции *владения объектом*. Когда кто-то говорит, что он “владеет объектом”, это означает, что он отвечает за его удаление и освобождение памяти.

Говорят, что объект с переменными экземпляра, которые указывают на другие объекты, *владеет* этими другими объектами. Например, в программе `CarParts` автомобиль владеет двигателем и колесами, на которые указывает. Аналогично функция, создающая

объект, владеет этим объектом. В программе CarParts функция main() создает новый объект автомобиля, так что мы говорим, что этим объектом владеет функция main().

Сложности начинаются, когда у некоторого объекта имеется несколько владельцев, т.е. его счетчик ссылок становится больше 1. В случае программы RetainCount1 функция main() владеет объектом RetainTracker, так что за его освобождение и удаление отвечает именно функция main().

Вспомните метод установки двигателя класса Car:

```
- (void) setEngine: (Engine *) newEngine;
```

и то, как он вызывается из функции main():

```
Engine *engine = [Engine new];
[car setEngine: engine];
```

Кто теперь владеет двигателем? Функция main() или объект car? Кто отвечает за отправку двигателю сообщения release о том, что он более не нужен? Это не может быть функция main(), поскольку объект класса Car использует двигатель. И это не может быть объект класса Car, поскольку функция main() может использовать двигатель позже.

Трюк заключается в том, чтобы заставить объект класса Car владеть двигателем, увеличивая его счетчик ссылок до 2. Это имеет смысл, поскольку двигатель теперь используют два владельца, объект класса Car и main(). Объект класса Car должен получить двигатель во владение в методе setEngine:, а функция main() должна освободиться от владения. Тогда объект класса Car будет отвечать за освобождение двигателя по окончании работы (в своем методе dealloc), и ресурсы двигателя окажутся освобождены и возвращены системе.

Владение и освобождение в методах доступа

Первая попытка написания версии метода setEngine, корректно работающего с управлением памятью, может иметь следующий вид:

```
- (void) setEngine: (Engine *) newEngine
{
 engine = [newEngine retain];
 // ПЛОХО. См. исправленную версию ниже.
} // setEngine
```

К сожалению, этого недостаточно. Представьте следующую последовательность вызовов в функции main():

```
Engine *engine1 = [Engine new]; // count: 1
[car setEngine: engine1];       // count: 2
[engine1 release];              // count: 1
```

```
Engine *engine2 = [Engine new]; // count: 1
[car setEngine: engine2];       // count: 2
```

Ой! У нас проблема с `engine1`: счетчик ссылок все еще равен 1. Функция `main()` уже уменьшила счетчик ссылок `engine1`, но объект класса `Car` этого никогда не делает. В результате получаем утечку памяти, на которую ссылается указатель `engine1`, а утечка двигателей еще никогда ни к чему хорошему не приводила. Первый объект `engine` оказывается всеми позабыт и позаброшен, но при этом потребляет блок памяти.

Попробуем переписать метод `setEngine:`.

```
- (void) setEngine: (Engine *) newEngine
{
 [engine release];
 engine = [newEngine retain];
 // ЕЩЕ ХУЖЕ. См. исправленную версию ниже.
} // setEngine
```

Этот код исправляет ранее рассмотренную утечку памяти, связанной с указателем `engine1`. Но он не работает в случае, когда объект, на который ссылается указатель `newEngine`, и старый двигатель оказываются одним и тем же объектом.

```
Engine *engine = [Engine new];   // count: 1
Car *car1 = [Car new];
Car *car2 = [Car new];

[car1 setEngine: engine];        // count: 2
[engine release];                // count: 1

[car2 setEngine: [car1 engine]]; // Ой!
```

В чем здесь проблема? Вот что произошло. `[car1 engine]` возвращает указатель на объект `engine`, счетчик ссылок которого равен 1. Первая строка метода `setEngine` — `[engine release]` — обнуляет счетчик ссылок, и объект уничтожается. Теперь и указатель `newEngine`, и переменная экземпляра `engine` указывают на освобожденную память, что весьма плохо. Есть лучший способ написать метод `setEngine`.

```
- (void) setEngine: (Engine *) newEngine
{
 [newEngine retain];
 [engine release];
 engine = newEngine;
} // setEngine
```

Если сначала увеличить счетчик нового двигателя, а объект, на который ссылается указатель `newEngine`, совпадает с объектом `engine`, то счетчик будет увеличен и тут же уменьшен. Но счетчик не уменьшится до нуля, и двигатель не окажется внезапно уничтоженным — что было бы весьма плохо. В методах доступа вполне безопасно увеличить счетчик ссылок нового объекта перед освобождением старого.

ПРИМЕЧАНИЕ. Имеются различные школы написания методов доступа, и между их последователями регулярно возникают бурные споры. Метод, изложенный в данном разделе, хорошо работает и достаточно прост в понимании, но не удивляйтесь, если у других программистов вы увидите другие методы (и не нервничайте, если ваш подход будут критиковать).

Автоматическое освобождение

Управление памятью может оказаться крепким орешком, и подтверждением тому — проблемы, с которыми мы встретились при написании устанавливающего метода доступа. Сейчас мы будем разбираться с еще одним затруднением. Вы знаете, что по окончании работы с ними объекты должны быть освобождены. В некоторых случаях получить информацию о том, что вы завершили работу с объектом, не так уж легко. Рассмотрим случай метода `description`, который возвращает строку класса `NSString`, описывающую объект.

```
- (NSString *) description
{
 NSString *description;
 description = [[NSString alloc]
     initWithFormat: @"Мне %d лет", 5];

 return (description);
} // description
```

Здесь при помощи сообщения `alloc` создается и возвращается новый экземпляр строки, счетчик ссылок которого получает значение 1. Кто же отвечает за удаление строкового объекта?

Это не может быть сообщение `description`. Если вы освободите строку описания до того, как вернете ее, счетчик ссылок уменьшится до нуля и объект будет удален немедленно.

Код, который использует описание, может воспользоваться строкой, а после освободить ее, но это делает использование описаний исключительно неудобным. Одна строка должна превратиться в три.

```
NSString *desc = [someObject description];
NSLog (@"%@", desc);
[desc release];
```

Должен быть другой способ. И, к счастью, он есть!

Все в пул!

В среде Cocoa есть концепция *пула автоматического освобождения*. Вы уже видели класс `NSAutoreleasePool` в шаблонном коде, генерируемом программой Xcode. Пришло время разобраться, что же это такое.

Само имя говорит о том, что это такое. Это *пул* (коллекция) чего-то, предположительно объектов, который автоматически выполняет освобождение.

У класса `NSObject` имеется метод `autorelease`.

```
- (id) autorelease;
```

Этот метод планирует отправление сообщения `release` в некоторый момент в будущем. Возвращаемое значение метода — объект, который получает это сообщение; метод `retain` использует ту же методику, которая упрощает объединение вызовов в цепочки. При отправлении сообщения `autorelease` объекту он добавляется к объекту класса `NSAutoreleasePool`. При уничтожении пула всем объектам, находящимся в нем, отсылается сообщение `release`.

> **ПРИМЕЧАНИЕ.** В концепции автоматического освобождения нет никакой магии. Вы можете написать собственный пул, используя класс `NSMutableArray` для хранения объектов и отправки всем хранящимся объектам сообщения `release` в методе `dealloc` данного пула. Но это не имеет смысла — Apple уже сделала всю сложную работу за вас.

Теперь метод `description` можно переписать следующим образом:

```
- (NSString *) description
{
 NSString *description;
 description = [[NSString alloc]
 initWithFormat: @"Мне %d лет", 5];

 return ([description autorelease]);
} // description
```

Так что можно написать следующий код:

```
NSLog (@"%@", [someObject description]);
```

Теперь управление памятью отработает корректно, поскольку метод `description` создаст новую строку, передаст ей сообщение `autorelease` и вернет ее функции `NSLog()` для вывода. Поскольку эта строка описания получила сообщение `autorelease`, она помещена в текущий пул автоматического освобождения, и в некоторый момент позже, после того как код функции `NSLog()` завершит свою работу, пул будет уничтожен.

О деструкции

Когда же уничтожается пул автоматического освобождения (и тем самым рассылает сообщения `release` всем содержащимся в нем объектам)? А когда он создается? Существуют два способа создать пул автоматического освобождения.

- С помощью ключевого слова `@autoreleasepool`.
- С помощью объекта класса `NSAutoreleasePool`.

При использовании каркаса Foundation создание и уничтожение пула выполняются явно. Поэтому при использовании выражения `@autoreleasepool{}`, находящийся внутри фигурных скобок, будет помещен в новый пул. Если ваши вычисления очень интенсивно используют память, вы можете делать пулы автоматического освобождения вложенными. При этом следует помнить, что любые переменные, определенные в фигурных скобках, невидимы снаружи; в этом случае применяются правила определения области видимости из языка C, как в циклах.

Второй, более явный способ основан на использовании объекта класса `NSAutoreleasePool`. В этом случае код, расположенный между словами `new` и `release`, использует новый пул.

```
NSAutoreleasePool *pool;
pool = [NSAutoreleasePool new];
...
[pool release];
```

Когда вы создаете пул автоматического освобождения, он автоматически становится активным. Когда вы освобождаете этот пул, его счетчик ссылок обнуляется, так что он удаляется, а в процессе этого он освобождает все находящиеся в нем объекты.

Какой из этих способов предпочесть? Используйте ключевое слово. Этот способ работает быстрее, чем объект, поскольку система выполнения программ на языке Objective-C лучше знает, как создавать и уничтожать пулы памяти.

При использовании AppKit Cocoa автоматически регулярно создает и удаляет пул автоматического освобождения. Он делает это после обработки программой текущего события (такого, как щелчок мыши или нажатие клавиши). Вы можете использовать столько автоматически освобождаемых объектов, сколько хотите, и пул автоматически будет их освобождать, как только пользователь выполнит какое-то действие.

ПРИМЕЧАНИЕ. В генерируемом программой Xcode исходном тексте вы могли встретить альтернативный способ уничтожения объектов пула автоматического освобождения: метод `-drain`. Этот метод опустошает пул, не уничтожая его. Метод `-drain` доступен только в операционной системе Mac OS X 10.4 (Tiger) и более поздних ее версиях. В вашем собственном коде (не генерируемом Xcode) лучше использовать `-release`, поскольку этот метод работает в любой операционной системе.

Пулы в действии

Программа `RetainTracker2` демонстрирует пул автоматического освобождения в действии. Ее можно найти в папке `09.02 RetainTracker-2`. Эта программа использует тот же класс `RetainTracker`, что и программа `RetainTracker1` (этот класс сообщает посредством функции `NSLog()` о создании и освобождении объектов).

Вот как выглядит функция `main()` этой программы:

```
int main (int argc, const char *argv[])
{
  NSAutoreleasePool *pool;
  pool = [[NSAutoreleasePool alloc] init];

  RetainTracker *tracker;
  tracker = [RetainTracker new]; // count: 1

  [tracker retain];              // count: 2
  [tracker autorelease];         // count: все еще 2
  [tracker release];             // count: 1

  NSLog (@"Освобождение пула");
  [pool release];

  // стирается, посылает сообщение release объекту tracker

  @autoreleasepool
  {
     RetainTracker *tracker2;
     tracker2 = [RetainTracker new]; // count: 1

    [tracker2 retain]; // count: 2
    [tracker2 autorelease]; // count: still 2
    [tracker2 release]; // count: 1

    NSLog (@"Автоматическое освобождение пула");
  }

return (0);
} // main
```

Начинаем с создания пула автоматического освобождения.

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

Когда мы посылаем сообщение `autorelease` объекту, тот попадает в данный пул.

```
RetainTracker *tracker;
tracker = [RetainTracker new];     // count: 1
```

Здесь создается новый объект `tracker`. Так как он создается при помощи сообщения `new`, его счетчик ссылок равен 1.

```
[tracker retain]; // count: 2
```

Теперь объект удерживается нами — просто в демонстрационных целях. Его счетчик ссылок становится равным 2.

```
[tracker autorelease]; // count: все еще 2
```

После этого объект с неизменным счетчиком ссылок попадает в пул автоматического освобождения. Когда пул, который теперь содержит ссылку на объект, будет уничтожен, объекту `tracker` будет послано сообщение `release`.

```
[tracker release]; // count: 1
```

Теперь захваченный ранее объект освобожден. Его счетчик ссылок остается больше нуля, так что жизненный цикл объекта продолжается.

```
NSLog (@"Освобождение пула");
[pool release];
```

Теперь мы освобождаем пул. Объект класса `NSAutoreleasePool` — такой же обычный объект, как и другие, — подчиняется тем же правилам управления памятью, что и все объекты. Поскольку пул был создан при помощи сообщения `alloc`, его счетчик ссылок равен 1. Сообщение `release` обнуляет счетчик ссылок, так что пул уничтожается (с вызовом его метода `dealloc`).

Наконец, функция `main` возвращает значение `0`, указывающее, что программа выполнена успешно.

```
 return (0);
} // main
```

Можете ли вы предположить, каким будет вывод рассмотренной программы? Не будем вас томить — он приведен ниже.

```
init: счетчик равен 1.
Освобождение пула
Вызван dealloc. Прощайте!
init: счетчик равен 1.
Автоматическое освобождение пула
Вызван dealloc. Прощайте!
```

Как вы, вероятно, и предполагали, сначала вызывается функция `NSLog()` из функции `main()`, а потом — функция `NSLog()` из класса `RetainTracker`.

Правила управления памятью Cocoa

Теперь вы знакомы со всеми сообщениями — `retain`, `release` и `autorelease`. В среде Cocoa есть ряд соглашений, связанных с управлением памятью. Это достаточно простые правила, последовательно применяемые во всем инструментарии.

> **ПРИМЕЧАНИЕ.** Распространенная ошибка при программировании с использованием среды Cocoa заключается в забывании этих правил, а также в попытке считать их сложнее, чем они есть на самом деле. Если вы рассыпаете по всему исходному тексту вызовы методов `retain` и `release` в надежде исправить некоторую ошибку, значит, вы не поняли эти правила. Это означает, что пора остановиться, глубоко вдохнуть, может быть, даже сделать перерыв и перекусить, а потом прочесть их снова на свежую голову.

Эти правила приведены ниже.

- Когда создаете объект с использованием сообщений `new`, `alloc` или `copy`, счетчик ссылок объекта становится равен 1. Вы отвечаете за отправку объекту сообщения `release` или `autorelease` по окончании работы с ним. Таким образом, этот объект уничтожается, когда становится ненужным.
- Если вы захватываете объекты при помощи некоторого другого механизма, считайте, что его счетчик ссылок равен 1 и что ему было послано сообщение `autorelease`. Вы не должны предпринимать никакие дальнейшие действия для гарантии его уничтожения. Если объект вам нужен в течение некоторого времени, перед началом работы захватите его при помощи сообщения `retain`, а по окончании — освободите посредством сообщения `release`.
- Если вы захватили объект, то в конечном счете обязаны освободить его. Сообщения `retain` и `release` должны быть сбалансированы.

И это все — только три правила.

Можете запомнить и повторять мантру: полученное с помощью сообщений `new`, `alloc` и `copy` надо либо освободить, либо автоматически освободить.

Владея объектом, вы должны помнить две вещи: как его получить и как долго им владеть (см. табл. 9.1).

Таблица 9.1. Правила управления памятью.

Получен с помощью...	Временный объект	Долговременный объект
new, alloc или copy	Освобождается по завершении	Освобождается методом dealloc
Другой способ	Не требует никаких действий	При получении удерживается; освобождается методом dealloc

Временные объекты

Рассмотрим обычный сценарий жизненного цикла. Мы используем объект в некотором фрагменте кода временно и не намерены хранить его очень долго. Если мы получаем объект при помощи сообщений `new`, `alloc` или `copy`, то должны обеспечить и его удаление, обычно посредством сообщения `release`.

```
NSMutableArray *array;
array = [[NSMutableArray alloc] init]; // count: 1

// Использование массива

[array release];                       // count: 0
```

Если вы получаете объект как-то иначе, например при помощи метода `arrayWithCapacity:`, то не должны беспокоиться о его уничтожении.

```
NSMutableArray *array;
array = [NSMutabelArray arrayWithCapacity: 17];
// count: 1, autorelease
// Использование массива
```

Метод `arrayWithCapacity:` — не `alloc`, не `new` и не `copy`, так что можете считать, что возвращенный методом объект имеет счетчик ссылок, равный 1, и уже помещен в пул автоматического освобождения. При уничтожении этого пула объекту `array` посылается сообщение `release`, его счетчик ссылок обнуляется, а память возвращается системе.

Фрагмент кода, использующего класс `NSColor`, приведен ниже.

```
NSColor *color;
color = [NSColor blueColor];
// Использование color
```

Метод `blueColor` — не `alloc`, не `new` и не `copy`, так что можете считать, что возвращенный методом объект имеет счетчик ссылок, равный 1, и уже помещен в пул автоматического освобождения. Объект `blueColor` возвращает глобальный *синглтон*, т.е. объект, который используется всеми нуждающимися в нем программами, и этот объект никогда не уничтожается, но вам незачем беспокоиться о деталях его реализации. Все, что вам надо знать, — это то, что вы *не* должны явно освобождать объект `color`.

Удержание объектов

Зачастую вам надо, чтобы объект существовал дольше, чем просто пару строк кода. Обычно такие объекты помещаются в переменные экземпляра других объектов, добавляются в коллекции наподобие `NSArray` или `NSDictionary` или, что встречается существенно реже, хранятся как глобальные переменные.

Если вы получаете объект при помощи сообщений `init`, `new` или `copy`, то не должны выполнять никакие особые действия. Счетчик ссылок такого объекта оказывается равен 1, так что он никуда не исчезает. Просто следует освободить этот объект в методе `dealloc` объекта-владельца, захватившего его.

```
- (void) doStuff
{
 // flonkArray — переменная экземпляра
 flonkArray = [NSMutableArray new]; // count: 1
} // doStuff

- (void) dealloc
{
 [flonkArray release];              // count: 0
 [super dealloc];
} // dealloc
```

Если вы получаете объект каким-то другим способом, вам надо не забыть захватить его. При написании приложений с графическим интерфейсом пользователя следует

думать в терминах цикла событий. Следует обязательно захватывать автоматически освобождаемые объекты, продолжительность жизни которых превышает текущий цикл событий.

Что же такое цикл событий? Типичное графическое приложение затрачивает массу времени на ожидание ввода пользователя. Программа дремлет в ожидании, пока этот тормоз-пользователь наконец-то щелкнет мышью или нажмет клавишу. Когда это случится, программа немедленно проснется и выполнит необходимые действия в ответ на происшедшее событие. После обработки события приложение снова засыпает в ожидании очередного события. Чтобы ваше приложение потребляло поменьше памяти, Cocoa создает пул автоматического освобождения перед началом обработки события и уничтожает его после того, как событие обработано. Тем самым количество памяти, выделяемой для временных объектов, поддерживается на минимальном уровне.

При использовании автоматически освобождаемых объектов предыдущие методы должны быть записаны следующим образом:

```
- (void) doStuff
{
 // flonkArray — переменная экземпляра
 flonkArray = [NSMutableArray arrayWithCapacity: 17];
 // count: 1, autorelease
 [flonkArray retain];  // count: 2, 1 autorelease
} // doStuff

- (void) dealloc
{
 [flonkArray release]; // count: 0
 [super dealloc];
} // dealloc
```

В конце текущего цикла событий (в случае программы с графическим интерфейсом пользователя) или при уничтожении пула автоматического освобождения `flonkArray` получает сообщение `release`, которое уменьшает значение счетчика ссылок с 2 до 1. Так как значение счетчика больше нуля, объект остается в живых. Мы должны освободить объект в своем методе `dealloc`, чтобы освободить занимаемую им память. Если в `doStuff` сообщение `retain` не будет послано, объект `flonkArray` окажется неожиданно уничтоженным.

Вспомните, что пул автоматического освобождения очищается в точно определенные моменты: когда он явно уничтожается в вашем коде и, при использовании AppKit, в конце цикла события. Вам не надо беспокоиться ни об освобождении пула в случайные моменты времени, ни о захвате каждого используемого объекта при помощи сообщения `retain`, так как пул не будет очищен прямо среди выполнения вашей функции.

ПРИНУДИТЕЛЬНАЯ ОЧИСТКА. Иногда пулы автоматического освобождения очищаются не так часто, как хотелось бы. Очень часто в различных рассылках, посвященных Cocoa, поднимается один и тот же вопрос: "Я послал всем используемым мною объектам сообщение `autorelease`, но потребляемая моей программой память вырастает до неимоверных размеров". Такая проблема обычно вызывается кодом наподобие следующего:

```
int i;
for (i = 0; i < 1000000; i++) {
 id object = [someArray objectAtIndex: i];
 NSString *desc = [object description];
 // Какие-то действия с описанием
}
```

Здесь работает цикл, который генерирует автоматически освобождаемый объект (а может, два или десять) всякий раз при выполнении очередной итерации. Вспомните, что пул автоматического освобождения очищается только в точно определенные моменты времени, и средина цикла в эти моменты не входит. В данном цикле создается миллион строк описания, и все они помещаются в текущий пул автоматического освобождения, так что у нас получается миллион строк, ожидающих своего часа. Когда пул будет уничтожен, всему этому миллиону строк придет конец, но не раньше.

Обходной путь заключается в создании собственного пула автоматического освобождения внутри цикла. Таким образом, например, каждую тысячу итераций вы можете уничтожать пул и создавать новый (как будет показано ниже; новый код выделен полужирным шрифтом).

```
NSAutoreleasePool *pool;
  pool = [[NSAutoreleasePool alloc] init];
  int i;
  for (i = 0; i < 1000000; i++) {
   id object = [someArray objectAtIndex: i];
   NSString *desc = [object descrption];
   // Какие-то действия с описанием
   if (i % 1000 == 0) {
       [pool release];
       pool = [[NSAutoreleasePool alloc] init];
   }
  }
  [pool release]
```

Каждую тысячу проходов по циклу новый пул уничтожается и создается вновь. Теперь одновременно сосуществует не более тысячи строк описания, так что потребляемая память падает на порядки. Создание и уничтожение пула автоматического освобождения — операция достаточно дешевая, так что новый пул можно создавать даже на каждой итерации цикла.

Пулы автоматического освобождения поддерживаются в виде стека: при создании нового пула он добавляется на вершину стека. Сообщение `autorelease` помещает его получателя в верхний пул в стеке. Если вы помещаете объект в пул, а затем создаете новый пул и уничтожаете его, указанный автоматически освобождаемый объект продолжает существование, так как пул, хранящий этот объект, продолжает существовать.

Чисто там, где убирают

В языке Objective-C 2.0 введен автоматический механизм управления памятью — режим сбора мусора. Программисты на языках Java или Python хорошо знакомы с этой концепцией. Вы создаете и используете объекты, а затем — вы не поверите! — просто забываете о них. Система автоматически разбирается, какие объекты все еще используются, а какие можно убирать. Включить режим сбора мусора очень просто — перейдите на вкладку Build Settings окна информации о проекте и выберите на ней Required [-fobjc-gc-only], как показано на рис. 9.1.

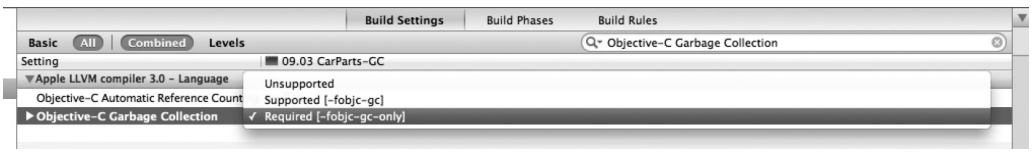

Рис. 9.1. *Включение режима сбора мусора*

ПРИМЕЧАНИЕ. Опция `-fobjc-gc` предназначена для создания кода, поддерживающего как сбор мусора, так и сообщения `retain/release`, как, например, код библиотеки, которая может быть использована в обеих средах.

При включении режима сбора мусора вызовы обычной системы управления памятью превращаются в холостые команды, не выполняющие никаких действий.

Вот как работает сборщик мусора в языке Objective-C. Время от времени сборщик мусора начинает просмотр ваших переменных и объектов, а также следует по всем исходящим из них указателям. Любой объект, на который в программе нет ни одного указателя, считается мусором, и его память освобождается. Худшее, что вы можете сделать, — это сохранить указатель на более не требующийся вам объект. Поэтому, если вы указали на объект в переменной экземпляра (вспомните композицию), убедитесь, что для этой переменной установлено значение `nil`, которое удаляет ссылку на этот объект и позволяет сборщику мусора удалить его.

Подобно пулу автоматического освобождения, сбор мусора запускается в конце цикла событий. Вы можете также самостоятельно запустить ее, если ваша программа не использует графический интерфейс пользователя, но эта тема выходит за рамки рассмотрения настоящей главы.

При использовании механизма сбора мусора вам не нужно беспокоиться об управлении памятью. Есть некоторые нюансы при работе с памятью, выделенной при помощи функции `malloc`, или с объектами Core Foundation, но здесь мы не будем их обсуждать. Пока что можете просто создавать объекты и не беспокоиться об их освобождении.

Заметим: вы не сможете использовать режим сбора мусора при разработке программного обеспечения для устройства iPhone. В действительности компания Apple рекомендует при программировании для iPhone избегать использования сообщения `autorelease` в вашем собственном коде и функций, которые возвращают автоматически освобожденные объекты.

Автоматический подсчет ссылок

Чем объясняется отсутствие механизма сбора мусора в системе iOS? Основной аргумент против режима сбора мусора в системе iOS заключается в том, что вы не знаете точно, когда следует его включать. Как и в реальной жизни, вы можете знать, что сборщик мусора придет в понедельник, но не знаете точное время его прихода. Что, если вы уйдете из дома именно тогда, когда приедет мусорная машина? Сбор мусора может значительно повлиять на использование мобильного устройства, которое носит более персональный характер, чем компьютер, и обычно имеет более скромные ресурсы, чем компьютер. Пользователи совсем не хотят, чтобы во время игры или телефонного разговора возникла пауза из-за включения режима сбора мусора.

Компания Apple преложила решение под названием *автоматический подсчет ссылок* (automatic reference counting — ARC). Как следует из названия, система ARC отслеживает ваши объекты и решает, какой из них следует хранить, а какой нет. Эта система управления памятью действует, как дворецкий или персональный секретарь. Когда вы используете систему ARC, вы размещаете объекты в памяти и удаляете их оттуда как обычно, а компилятор вставляет сообщения `retains` и `releases` автоматически — вы об этом можете не беспокоиться.

Механизм ARC *не является* системой сбора мусора. Как указывалось выше, механизм сбора мусора работает во время выполнения программы с помощью кода, который периодически запускается и проверяет ваши объекты. В противоположность этому механизм ARC работает на этапе компиляции. Он вставляет в код соответствующие сообщения `retains` и `releases`, и система управления памятью работает так, будто это вы написали очень хорошо написанный код. Правда, в этом случае управлением памятью занимаетесь не вы, а компилятор. После запуска программы сообщения `retains` и `releases` передаются как обычно. Система не знает и не интересуется, кто вставил в код эти команды — вы или компилятор.

Механизм ARC является функциональной возможностью, которая включается *по согласию* (opt-in feature), а это значит, что вы должны явным образом включать и отключать ее.

Ниже перечислены компоненты, необходимые для использования механизма ARC.

- Программа Xcode 4.2+.
- Компилятор Apple LLVM 3.0+ (который поставляется с программой Xcode).
- OS X 10.7+.

Для запуска кода, использующего механизм ARC, требуются следующие условия.

- Операционная система iOS 4.0+ или OS X 10.6+ с 64-битовой системой выполнения программ.
- Обнуляемые слабые ссылки (zeroing weak references) (см. далее) в системах iOS 5.0+ или OS X 10.7+.

ПРИМЕЧАНИЕ. Механизм ARC работает только с указателями на хранящиеся в памяти объекты (retainable object pointer — ROP). Существуют три вида объектов на хранящиеся в памяти объекты:

1) указатели на блоки;

2) указатели на объекты Objective-C;

3) инструкции typedef с пометкой `__attribute__((NSObject))`.

В основном мы будем говорить об указателях ROP, особенно при описании инструмента преобразования из механизма ARC в среде Xcode.

Все другие типы указателей, такие как `char *` и объекты `CF`, а также объекты класса `CFStringRef`, не совместимы с механизмом ARC.

Если вы используете указатели, не поддерживаемые механизмом ARC, то должны сами ими управлять. Ничего страшного в этом нет, потому что механизм ARC взаимодействует с механизмом ручного управления памятью.

Если вы хотите использовать механизм ARC в своем коде, то должны выполнить три требования.

- Иметь возможность надежно идентифицировать объект, которым следует управлять.
- Иметь возможность указать, как именно управлять объектом.
- Иметь надежный механизм передачи владения объектом.

Рассмотрим эти требования по очереди. Первое требование означает, что корневой объект коллекции объектов должен знать, как управлять своими дочерними объектами. Например, допустим, что у нас есть массив строк, созданных с помощью функции `malloc`.

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

Этот код создает C-массив, указывающий на десять строк. Поскольку C-массивы не являются хранимыми, к этой структуре нельзя применять механизм ARC.

Это приводит ко второму требованию. Это требование обычно означает, что вы должны иметь возможность увеличивать и уменьшать хранимый счетчик на объект, который является производным от класса `NSObject`. К этой категории относится большинство объектов, которыми необходимо управлять.

Третье требование утверждает, что при передаче объекта ваша программа должна иметь возможность передавать владение этим объектом от вызывающего метода вызываемому. Этот вопрос мы рассмотрим позднее.

Подумав обо всех этих ограничениях, вы можете спросить себя: "А нужен ли мне механизм ARC вообще?" Конечно, нужен! Мы гарантируем, что он избавит вас от сложностей и сэкономит время при длительном выполнении программ.

Иногда слабость — это сила

Как указывалось выше, если у нас есть указатель на объект, мы можем либо управлять его памятью (с помощью сообщений `retain`/`release`), либо просто указывать на него, не участвуя в управлении памятью. Если вы управляете памятью, которую занимает объект, говорят, что вы имеете *сильную ссылку* (strong reference) на объект. Если вы не управляете его памятью, то имеете (как уже, возможно, догадались) *слабую ссылку* (weak reference). Например, если вы используете атрибут `assign` для свойства, то создаете слабую ссылку.

Зачем нужны слабые ссылки? Они помогают справляться с циклами хранения. Рассмотрим это явление подробнее.

Допустим, у нас есть объект A, созданный другим объектом. Счетчик ссылок на объект A равен 1 (рис. 9.2). Объект A создает объект B как дочерний объект, и счетчик ссылок на объект B равен 1 (рис. 9.3). Объект B должен иметь доступ к своему родителю; это очень распространенная схема в программах на языке Objective-C.

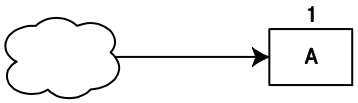

Рис. 9.2. *Счетчик ссылок на объект A равен 1*

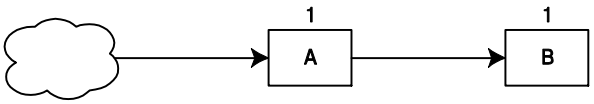

Рис. 9.3. *Счетчик ссылок на объект B, являющийся дочерним по отношению к объекту A, также равен 1*

В нашем примере объект A имеет сильную ссылку на объект B, так как объект A создан объектом B. Теперь, если объект B получит сильную ссылку на объект A, счетчик ссылок на объект A увеличится до 2 (рис. 9.4).

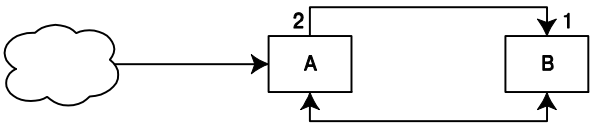

Рис. 9.4. *Объект B имеет сильную ссылку на объект A, и счетчик ссылок на объект A увеличивается*

Все в мире когда-то заканчивается, так что в конце концов владелец объекта A больше не будет в нем нуждаться. В этом случае владелец вызовет сообщение `release` для объекта A, что приведет к уменьшению его счетчика ссылок на 1.

Однако объект B, созданный объектом A, владеет этим счетчиком ссылок. Поскольку оба объекта имеют ненулевые счетчики ссылок, ни один из них не удаляется из памяти. Это классическая утечка памяти: программа не имеет доступа к объектам, но по-прежнему занимает память (рис. 9.5).

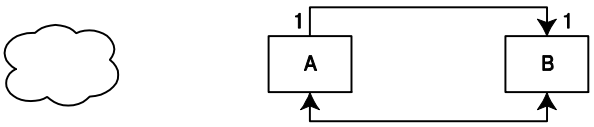

Рис. 9.5. *Классическая утечка памяти*

Для решения этой проблемы используется слабая ссылка. Мы используем атрибут `assign`, чтобы получить ссылку объекта B на объект A. При слабой ссылке счетчик ссылок не увеличится. Итак, когда владелец объекта A удалит его, соответствующий счетчик цикла станет равным нулю, будет удален объект B, и память, занимаемая объектами A и B, будет освобождена (рис. 9.6). Отлично, не так ли?

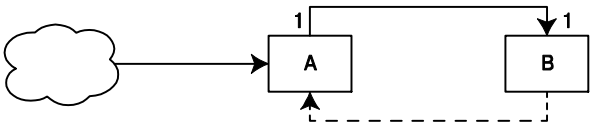

Рис. 9.6. *Объект B имеет слабую ссылку на объект A*

Лучше, но пока не идеально. Если у вас есть три объекта (назовем их A, B и C), вы останетесь в результате с объектом A, владеющим объектом B и имеющим сильную ссылку на него, в то время как объект C будет иметь слабую ссылку на объект B (рис. 9.7).

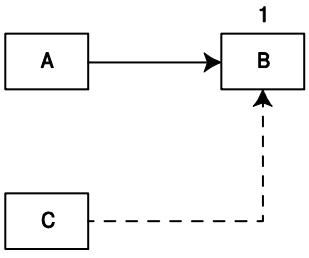

Рис. 9.7. *Два объекта ссылаются на объект B: один сильно, другой слабо*

Если объект A освободит объект B, то объект C по-прежнему будет иметь слабую ссылку на него, но эта ссылка больше не будет корректной, и ее использование скорее всего приведет к краху, поскольку по данному адресу больше нет корректного объекта (рис. 9.8).

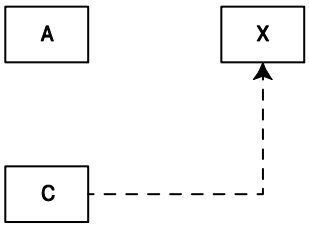

Рис. 9.8. *Объект C по-прежнему имеет ссылку на освобожденный объект. Это плохо*

Вот решение: объект, автоматически удаляющий свои слабые ссылки. Эти особые слабые ссылки называются *обнуляемыми слабыми ссылками* (zeroing weak references) потому что, когда объект, на который они ссылаются, удаляется, они приравниваются к нулю (`nil`) и могут обрабатываться как любой другой нулевой указатель (рис. 9.9). Обнуляемые слабые ссылки доступны только в операционной системе iOS 5 и более поздних версиях, а также в операционной системе OS X 10.7 и ее новейших версиях.

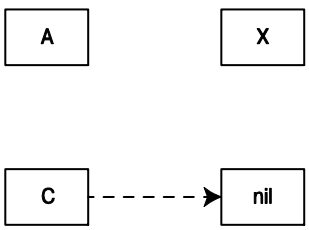

Рис. 9.9. *Объект C имеет обнуляемую слабую ссылку, которая превращается в нуль. если перестает быть корректной*

Для использования обнуляемых слабых ссылок их необходимо явно объявить. Существуют два способа объявления обнуляемых слабых ссылок: с помощью ключевого слова `__weak`, если объявляется переменная, и с помощью атрибута `weak`, если объявляется свойство.

```
__weak NSString *myString;
@property(weak) NSString *myString;
```

Что делать, если вы хотите использовать механизм ARC в более старых операционных системах, в которых обнуляемые слабые ссылки недоступны? Компания Apple предлагает использовать ключевое слово `__unsafe_unretained` и атрибут `unsafe_unretained`, которые сообщают механизму ARC, что указанная ссылка является слабой.

Существует несколько ограничений на использование механизма ARC.

- Нельзя использовать свойство, имя которого начинается со слова “new”. Например, не допускается объявление `@property NSString *newString`.
- Нельзя использовать свойство только для чтения без атрибутов управления памятью. Если вы не используете механизм ARC, то можете использовать объявление

`@property (readonly) NSString *title`. Но если вы используете механизм ARC, то должны указать, кто управляет памятью, так что достаточно просто вставить ключевое слово `unsafe_`unretained, потому что по умолчанию используется атрибут `assign`.

Для симметрии в языке Objective-C есть ключевое слово `__strong` и атрибут `strong`. Имейте в виду, что ключевые слова и атрибуты, связанные с управлением памятью, являются взаимоисключающими.

Хорошо! Теперь мы готовы преобразовать программу `CarParts`, чтобы она использовала механизм ARC. Когда мы это сделаем, вы заметите, что код стал еще меньше, чем был раньше. Наш лозунг — это лозунг всех программистов: “меньше, значит лучше”.

Преобразование проекта

Программа Xcode предоставляет удобную процедуру для преобразование существующих проектов в проекты, использующие механизм ARC. Перед началом мы должны отключить режим сбора мусора; сбор мусора и механизм ARC нельзя использовать одновременно. Если вы попытаетесь преобразовать проект, не отключив режим сбора мусора, то получите предупреждение, показанное на рис. 9.10.

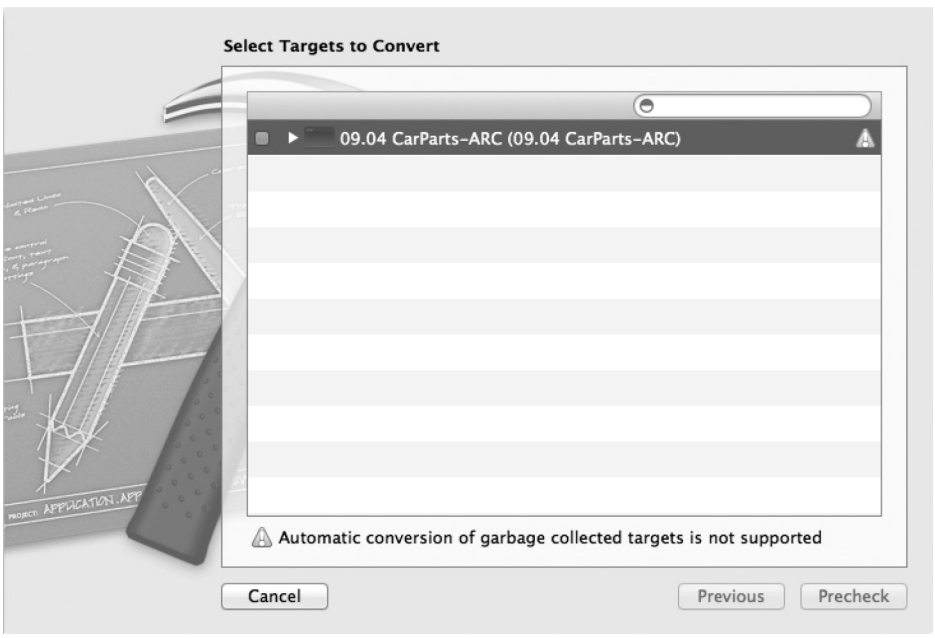

Рис. 9.10. *Перед преобразованием проекта необходимо отключить режим сбора мусора*

Рассмотрим процесс отключения механизма сбора мусора. На первом этапе необходимо выбрать цель, для которой мы хотим отключить сбор мусора. Откройте ваш проект (если он еще не открыт) и выберите проект в области навигатора. В верхней части области редактора вы увидите ваш проект, а ниже — цель (рис. 9.11).

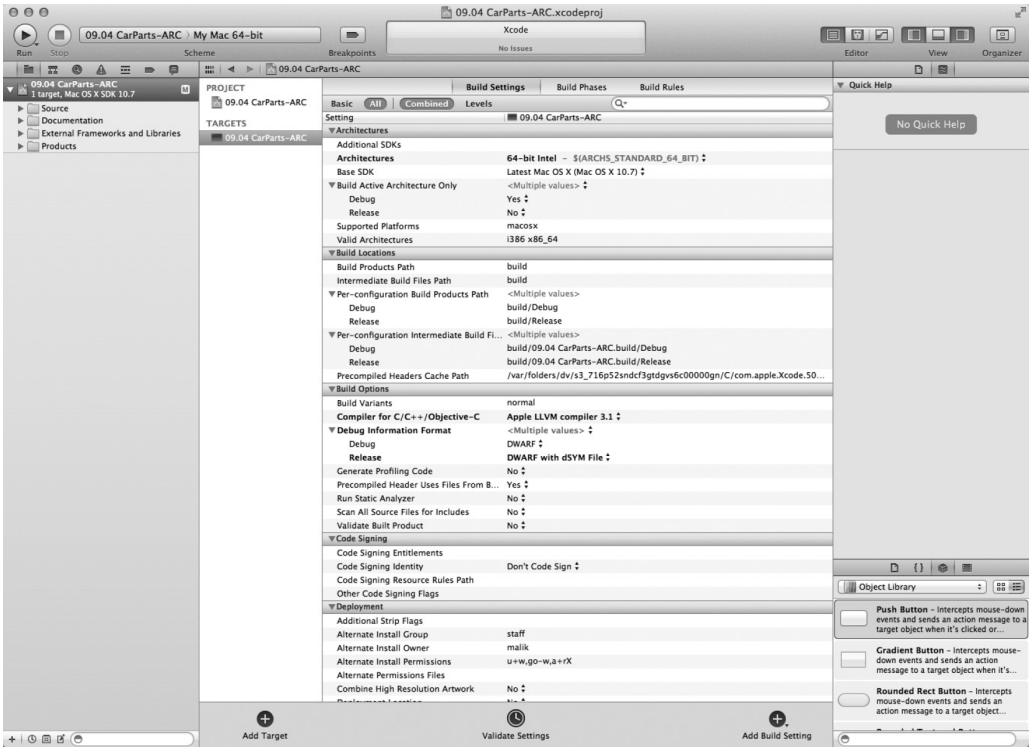

Рис. 9.11. *Откройте ваш проект в среде Xcode*

Теперь выберите цель, для которой хотите отключить механизм сбора мусора. Затем щелкните на вкладке Build Settings в верхней части области редактора. Вы увидите, что у вашей цели очень много настроек, которые могут быть лишними. Найдите среди них настройку “Objective-C Garbage Collection”. Это можно сделать простым просмотром, но есть более удобный способ.

Компания Apple знает, как трудно бывает найти нужную настройку, поэтому, если вы посмотрите чуть ниже вкладок Build Settings, то увидите удобное поле поиска. Введите в это поле строку *garbage collection*. Поле поиска сузит все множество настроек до подмножества настроек, которые связаны со сбором мусора. Это много лучше! Теперь вам достаточно просмотреть два пункта.

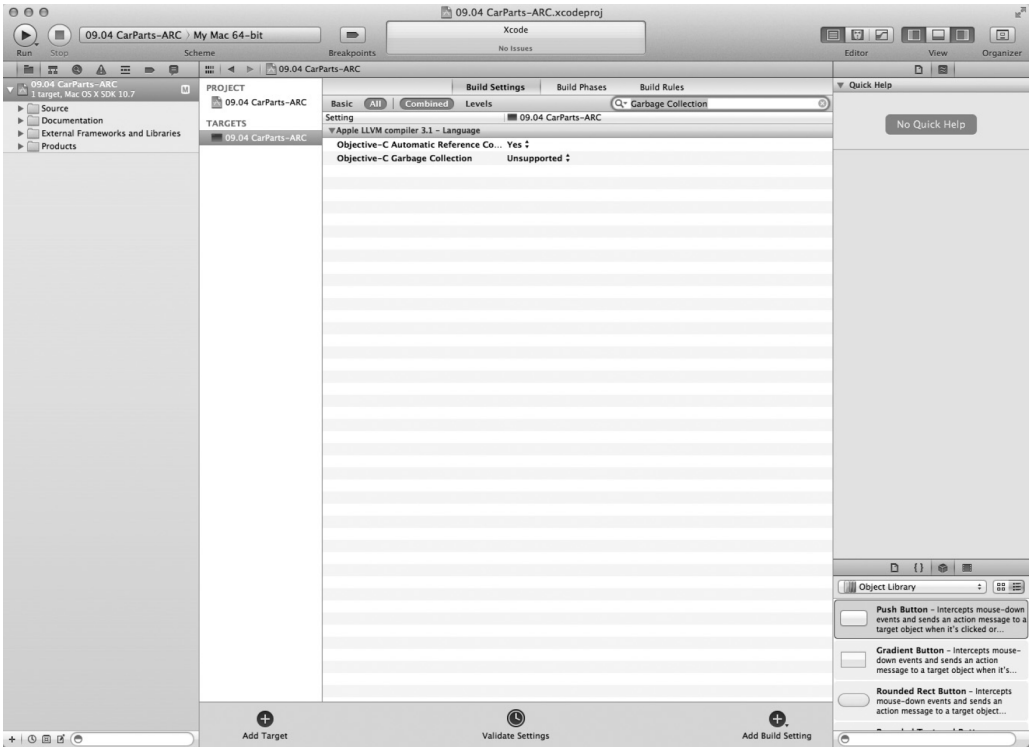

Рис. 9.12. *Настройки механизма сбора мусора*

Когда вы будете просматривать пункт “Objective-C Garbage Collection”, контекстное меню предоставит вам три варианта выбора.

- Unsupported. Вы должны выбрать эту команду. если ваш проект использует механизм ARC или вы по каким-то причинам не хотите включать механизм сбора мусора.
- Supported. Ваше приложение может использовать механизм сбора мусора.
- Required. Ваше приложение *обязано* использовать механизм сбора мусора.

Мы выбираем первый вариант: unsupported (рис. 9.13). Он отключает механизм сбора мусора.

Преобразование проекта состоит из двух этапов. На первом этапе вы должны убедиться, что ваш код соответствует требованиям механизма ARC, а на втором этапе выполнить фактическое преобразование.

> **ПРИМЕЧАНИЕ.** Преобразование ARC является односторонним. Перейдя на использование механизма ARC, вы не сможете вернуться обратно.

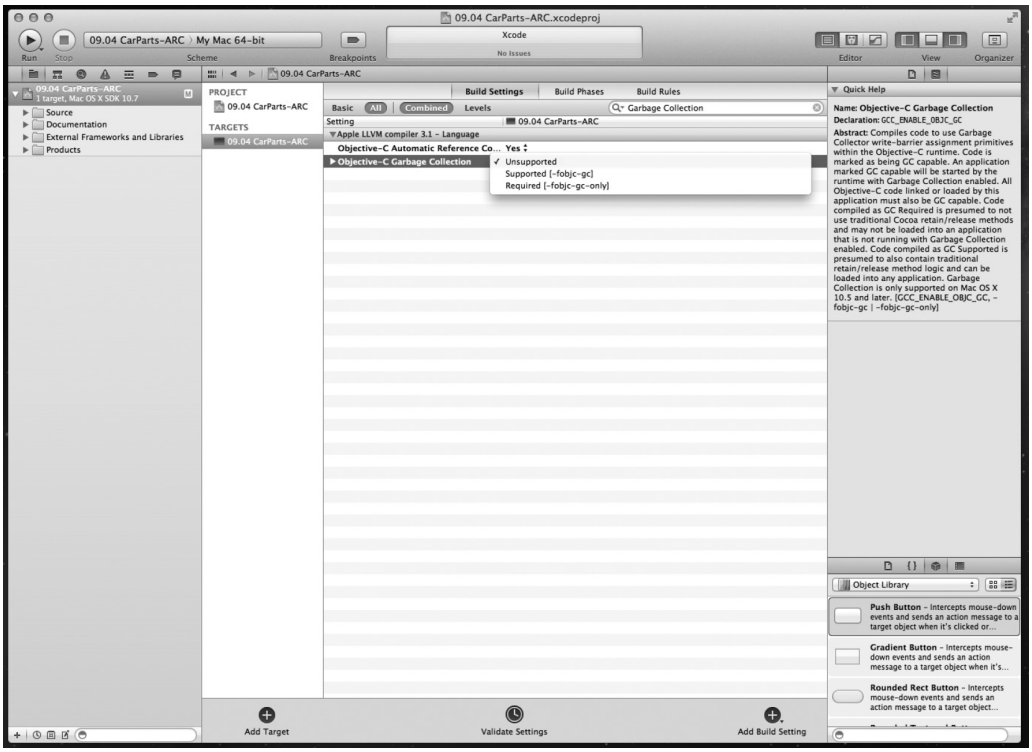

Рис. 9.13. *Выбор команды* Unsupported *отключает механизм сбора мусора*

Для начала выберите цель, которую хотите преобразовать, а затем команду Edit⇨Refactor⇨Convert to Objective-C ARC (рис. 9.14).

Поскольку механизм ARC работает на файловом уровне, вы можете одновременно использовать как файлы ARC, так и файлы, не использующие механизм ARC, для достижения одной и той же цели.

Далее вы увидите список целей, доступных для преобразования (рис. 9.15). Если ваша цель зависит от других целей, можете выбрать процесс, состоящий из небольших шагов. Например, вы можете начать с каркасов и библиотек и, закончив с ними, перейти к целям, которые их используют.

Когда вы выберете цель для преобразования, по умолчанию будут выбраны все файлы реализации. Если у вас есть файлы, которые используются одновременно несколькими проектами, и вы не хотите их преобразовывать, можете выбрать только те файлы, которые захотите. Когда вы закончите выбирать файлы, щелкните на кнопке Precheck. После завершения процесса предварительной проверки можно переходить на следующий этап — к реальному преобразованию (наконец), как показано на рис. 9.16.

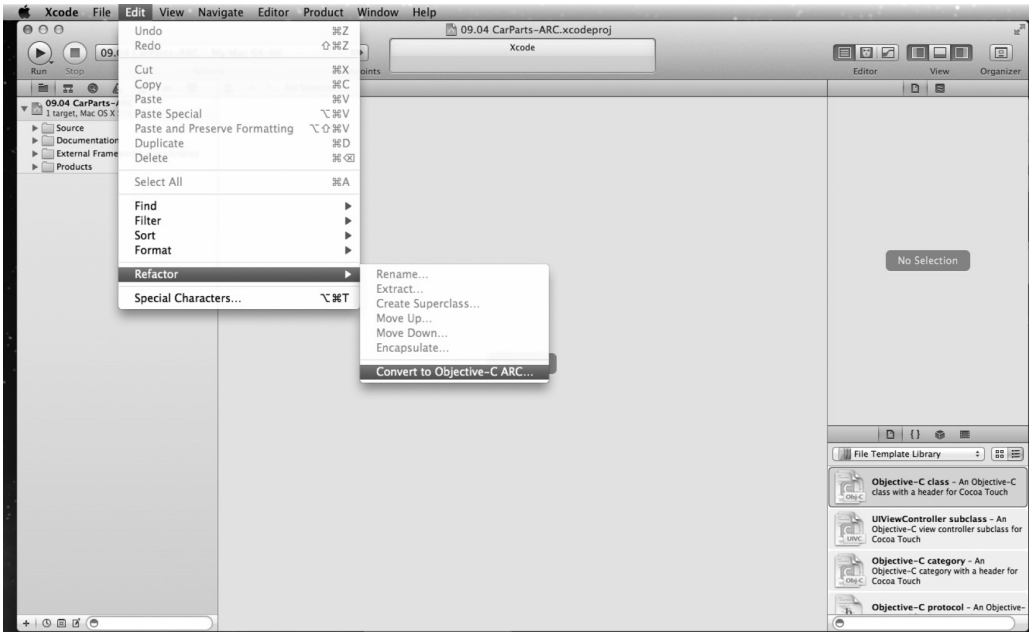

Рис. 9.14. *Для выбора команды преобразования в ARC используется меню* Edit

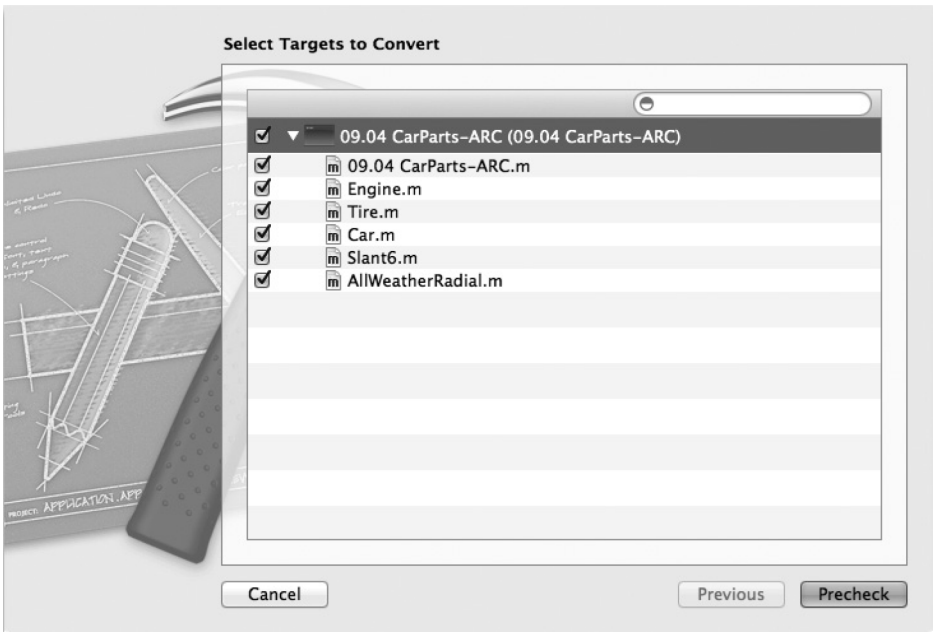

Рис. 9.15. *Выберите файлы, которые хотите преобразовать*

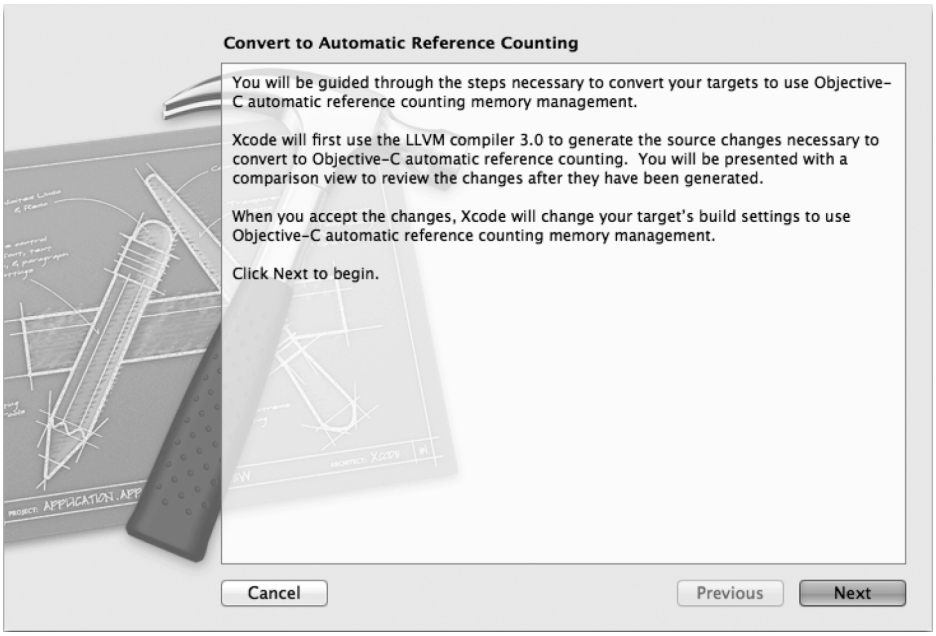

Рис. 9.16. *Вводный экран преобразования ARC*

После того как вы прочитаете полезное вводное сообщение и щелкнете на кнопке Next, будет выполнен этап преобразования и ваш проект будет превращен в проект, использующий механизм ARC. На этом этапе программа Xcode находит каждый файл, который должен быть скомпилирован с помощью механизма ARC, и модифицирует ваш исходный код.

По завершении процесса программа Xcode выводит представление кода для сравнения. В этом представлении вы можете увидеть содержание каждого файла до и после преобразования и решить, подходят ли вам внесенные изменения.

Мы близки к финишу, но проект еще не полностью преобразован. Эти изменения внесены в копии ваших файлов, чтобы вы могли отменить свое решение и вернуться в первоначальному варианту (напоминаем, что процесс преобразования проекта необратим). Для завершения преобразования можете сохранить копии предыдущих версий файлов, а затем закончить работу.

Это довольно легко, но следует признать, что мы описали самый лучший сценарий, в котором все выполняется идеально и автоматически. Реальная жизнь не всегда соответствует нашим ожиданиям. Конвертер работает только с исходным кодом; он не читает комментарии, и его текущая версия не может распознать ваши намерения. Если конвертер найдет малейшую неоднозначность, он пометит ее, и вы должны будете устранить проблему до завершения преобразования.

Владение имеет привилегии

Одно из требований, которые мы обсуждали ранее, заключалось в том, что указатели в коде ARC должны быть указателями на хранимые объекты (ROP). Это значит, например, что вы не можете просто присвоить указатель ROP указателю, который не является указателем ROP, потому что при этом произойдет передача владения указателем. Рассмотрим следующий код:

```
NSString *theString = @"Learn Objective-C";
CFStringRef cfString = (CFStringRef)theString;
```

Если вы читали много программ на языке Objective-C, то, вероятно, видели этот шаблон. Что он делает? Один указатель, `theString`, является указателем ROP, а другой — `CFStringRef` — нет. Для того чтобы все было в порядке, вы должны сообщить компилятору, кто владеет указателем. Для этого можно использовать прием языка C, называемый *мостовым приведением* (bridged cast). Это стандартное приведение типов в языке C, но с дополнительными ключевыми словами: `__bridge`, `__bridge_retained` или `__bridge_transfer`. Термин "мост" означает возможность использовать разные типы данных для одной и той же цели, чтобы вы не испытывали ощущения "перепрыгивания" при выполнении действительно сложного преобразования ARC. Рассмотрим описания каждого из трех типов мостовых преобразований.

- `(__bridge Type)` *операнд*. Этот вид приведения преобразует указатель, но не передает права владения им. В предыдущем примере операндом был объект `theString`, а `Type` — `CFStringRef`. При использовании этого ключевого слова один указатель является указателем ROP, а второй — нет. При этом приведении владение указателем остается у операнда. Как может выглядеть наш пример в этом случае, показано ниже.

  ```
  cfString = (__bridge CFStringRef)theString;
  ```

  Объект `cfString` получает результат присваивания, но владение остается у объекта `theString`.

- `(__bridge_retained Type)` *операнд*. При выполнении этого приведения владение передается указателю, который не является указателем ROP. Один из указателей является указателем ROP, а второй — нет. Поскольку механизм ARC интересуют только указатели ROP, вы обязаны удалить этот объект, когда он не будет вам нужен. Это приведение увеличивает счетчик ссылок на объект на единицу, поэтому вы должны уменьшить его на единицу, как при работе со старым стандартным механизмом управления памятью. Ниже приведен соответствующий вид нашего примера.

```
cfString = (__bridge_retained CFStringRef)theString
```

В этом случае строка cfString владеет указателем, а ее счетчик увеличивается на единицу. Вы должны самостоятельно управлять этим счетчиком с помощью сообщений `retain`/`release`.

- `(__bridge_transfer Type)` *операнд*. Это приведение противоположно предыдущему; оно передает владение указателю ROP. В данном случае механизм ARC владеет объектом и гарантирует его удаление, как и любого другого объекта ARC.

Другое ограничение заключалось в том, что указатели ROP не могли быть членами структур и объединений, поэтому следующий код является неправильным:

```
// Неправильный код. Не копируйте его.
struct {
    int32_t foo;
    char *bar;
    NSString *baz;
} MyStruct;
```

Это ограничение можно обойти с помощью ключевого слова `void *` и мостового приведения. Для выполнения присваивания и получения строки обратно ваш код должен выглядеть следующим образом:

```
struct {
    int32_t foo;
    char *bar;
    void *baz;
} MyStruct;
MyStruct.baz = (__bridge_retained void *)theString;
NSString *myString = (__bridge_transfer NSString *)MyStruct.baz;
```

Как видите, код ARC не всегда генерируется автоматически. Это очень глубокая тема, заслуживающая подробного изучения. В заключение нашего обсуждения механизма ARC перечислим методы, которые не должны применяться к объектам, управляемым механизмом ARC.

- `retain`
- `retainCount`
- `release`
- `autorelease`
- `dealloc`

Поскольку иногда необходимо удалять объекты, не управляемые механизмом ARC, или выполнять другие операции по освобождению памяти, вы можете по-прежнему использовать методы `dealloc`, но не можете выполнять вызов `[super dealloc]`.

Кроме того, существуют методы, которые нельзя замещать в объектах ARC.

- retain
- retainCount
- release
- autorelease

Исключительность

Что такое исключение? Это неожиданное событие, например, выход за пределы массива, которое разрушает поток выполнения программы, поскольку программа действительно не знает, как поступать в сложившейся ситуации.

Когда это происходит, программа может создать объект исключения и передать его системе выполнения программ, которая решает, что делать дальше. Исключения в среде Cocoa представлены классом `NSException`. Среда Cocoa требует, чтобы все исключения имели тип `NSException`, так что если вы сгенерировали исключение из других объектов, среда Cocoa не станет с ними работать. Вместо этого вы должны создать подкласс класса `NSException`, чтобы создать собственные исключения.

> **ПРИМЕЧАНИЕ.** Обработка исключений действительно предназначена для предотвращения ошибок, генерируемых вашей программой. Каркасы среды Cocoa обычно обрабатывают ошибки, выполняя выход из программы, а это не то, чего мы хотим. Вы должны сгенерировать и перехватить исключение в своем коде, а не дать ему убежать на уровень каркаса.

Для того чтобы включить поддержку исключений, убедитесь, что флаг -fobj-exceptions установлен. Вы можете установить его с помощью команды Enable Objective-C Exceptions в среде Xcode (рис. 9.17).

Создание исключения и его обработка системой выполнения программы называется *генерированием* (throwing) исключения, а иногда *возбуждением* исключения (raising an exception). Обратите внимание на то, что класс `NSException` имеет несколько методов, имя которых начинается со слова `raise`. Некоторые из них являются методами класса, а сам метод `raise` является методом экземпляра.

Перехват (catching) исключения — это обработка сгенерированного исключения.

> **ПРИМЕЧАНИЕ.** Если исключение сгенерировано, но не перехвачено, то программа останавливает работу в точке исключения и передает его дальше.

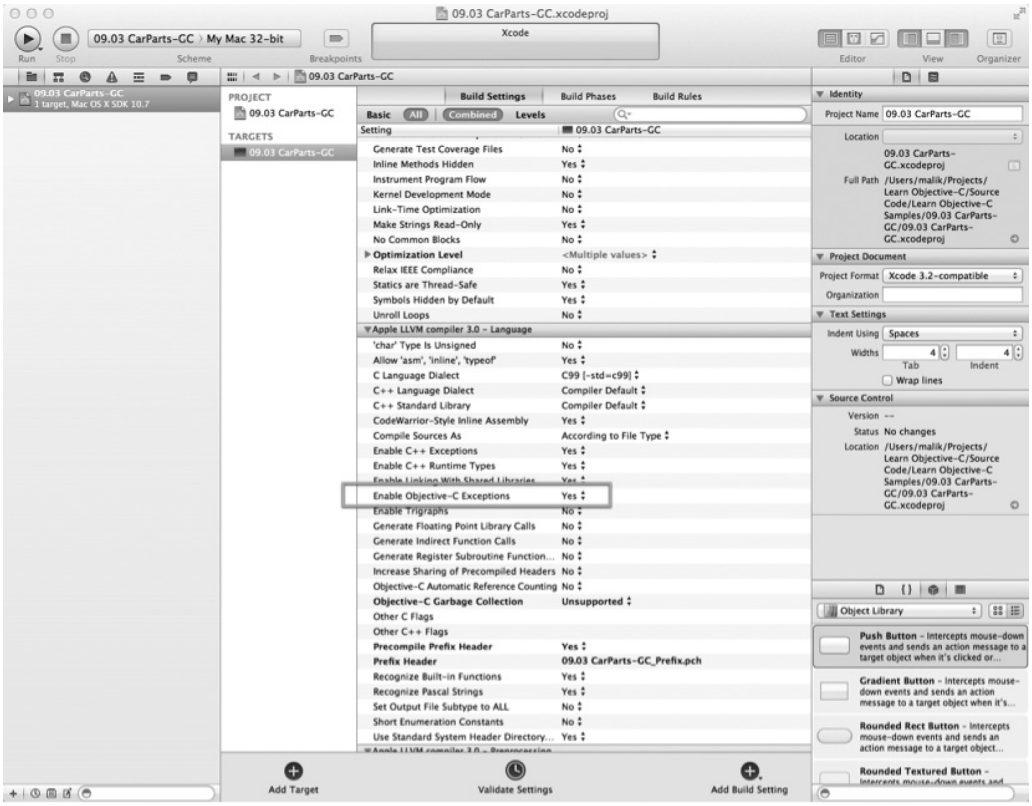

Рис. 9.17. *Включение исключений*

Ключевые слова для работы с исключениями

Все ключевые слова для работы с исключениями начинаются символом `@`. Перечислим их.

- `@try`. Определяет блок, который будет проверяться, чтобы определить, следует ли генерировать исключение.
- `@catch()`. Определяет блок кода для обработки сгенерированного исключения. Принимает аргумент, обычно имеющий тип `NSException`, но может иметь и другой тип.
- `@finally`. Определяет блок кода, который должен быть выполнен независимо от того, было ли сгенерировано исключением или нет. Этот код выполняется всегда.
- `@throw`. Генерирует исключение.

ПРИМЕЧАНИЕ. Ранее мы указывали, что исключения можно генерировать из объектов, не имеющих тип `NSException`, но вы должны избегать этого. Причина заключается в том, что не каждый каркас среды Cocoa перехватывает исключения, не имеющие тип `NSException`. Для того чтобы среда Cocoa правильно работала с вашими исключениями, вы должны генерировать только объекты класса `NSException`.

Обычно ключевые слова `@try`, `@catch` и `@finally` используются вместе в одной структуре. Эта структура похожа на следующую конструкцию.

```
@try
{
    // Код, который может сгенерировать исключение.
}
@catch (NSException *exception)
{
     // Код для обработки исключения.
}
@finally
{
     // Код, который выполняется всегда. Обычно освобождает память.
}
```

Перехват разных типов исключений

Вы можете расставить несколько блоков `@catch` в зависимости от типа перехватываемого исключения. Ваши обработчики должны работать по порядку в направлении от самого общего до самого конкретного исключения, а универсальный обработчик должен выполняться после всех остальных. Рассмотрим пример.

```
@try{
} @catch (MyCustomException *custom) {
} @catch (NSException *exception) {
} @catch (id value) {
} @finally {
}
```

ПРИМЕЧАНИЕ. В коде для обработки исключений программисты на языке C часто используют функции `setjmp` и `longjmp`. Вы не сможете использовать функции `setjmp` и `longjmp` в блоке `@try`, если переход выводит поток выполнения за пределы фигурных скобок блока `@try`. Однако для выхода из кода обработки исключения можно использовать инструкции `goto` и `return`.

Генерирование исключений

Когда программа обнаруживает исключение, она должна передать его в блок кода, предназначенного для ее обработки. Этот код называется обработчиком исключения. Как указывалось ранее, процесс распространения исключения называется генерированием, или возбуждением исключения.

Программа генерирует исключение, создавая экземпляр класса `NSException`, одним из двух способов.

- С помощью директивы `@throw`.
- С помощью посылки сообщения `raise` объекту класса `NSException`.

Например, создадим исключение

```
NSException *theException = [NSException exceptionWithName: …];
```

Затем мы можем сгенерировать его с помощью директивы

```
@throw theException;
```

или

```
[theException raise];
```

Использовать можно только один из этих двух методов, но не оба сразу. разница между ними заключается в том, что сообщение `raise` можно использовать только для объектов класса `NSException`, а директива `@throw` работает со всеми объектами.

Обычно исключения генерируются за пределами кода, предназначенного для их обработки. Ваш код может распространить исключение, послав сообщение `raise` или выполнив директиву `@throw`.

```
@try
{
    NSException *e = ...;
    @throw e;
}
@catch (NSException *e) {
       @throw; // Повторное генерирование исключения e.
}
```

ПРИМЕЧАНИЕ. В блоке `@catch`, предназначенном для обработки исключений, можно повторно сгенерировать исключение, не указывая объект исключения.

Блок `@finally`, связанный с локальным обработчиком исключений `@catch`, выполняется до того, как директива `@throw` вызовет следующий обработчик исключения, относящийся к более высокому уровню, поскольку директива `@finally` выполняется до директивы `@throw`.

Исключения в языке Objective-C совместимы с системой исключений в языке C++.

ПРИМЕЧАНИЕ. Исключения в языке Objective-C интенсивно потребляют ресурсы. Исключения не следует использовать в общем потоке выполнения или просто для вылавливания ошибок. Установка блока генерирования исключения с помощью директивы `@try` не требует дополнительных затрат, но перехват исключения связан с существенными затратами ресурсов и снижением скорости выполнения программы.

Исключения тоже требуют управления памятью

При работе с исключениями управление памятью может быть затруднено. Рассмотрим простой код.

```
- (void)mySimpleMethod
{
    NSDictionary *dictionary = [[NSDictionary alloc] initWith....];
    [self processDictionary:dictionary];
    [dictionary release];
}
```

Теперь представьте себе, что метод `processDictionary` генерирует исключение. Программа выходит из метода и ищет обработчик исключения. Однако, поскольку в этой точке метод прекращает работу, объект словаря остается в памяти и возникает ее утечка.

Существует простой способ избежать этой неприятности — использовать директиву `@try` или `@finally`, выполняя в блоке `@finally` определенные действия, связанные с освобождением памяти, поскольку он выполняется всегда (как мы уже говорили).

```
- (void)mySimpleMethod
{
    NSDictionary *dictionary = [[NSDictionary alloc] initWith....];
    @try {
      [self processDictionary:dictionary];
    }
    @finally {
     [dictionary release];
    }
}
```

Этот шаблон встречается и при управлении памятью в стиле языка C, например, при использовании функций `malloc` и `free`.

Исключения и пулы автоматического освобождения

Обработка исключений иногда может немного усложниться из-за автоматического освобождения объекта исключения. Исключения почти всегда создаются как автоматически освобождаемые объекты, потому что мы не знаем, когда они должны быть удалены из памяти. После разрушения пула автоматического освобождения все содержавшиеся в нем объекты также разрушаются, включая исключение. Рассмотрим следующий код, который внешне выглядит вполне безопасным:

```
- (void)myMethod
{
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
NSDictionary *myDictionary =
    [[NSDictionary alloc] initWithObjectsAndKeys:@"asdfads", nil];
@try {
    [self processDictionary:myDictionary];
} @catch (NSException *e) {
```

```
        @throw;
    } @finally {
        [pool release];
    }
}
```

Внешне все прекрасно: мы создаем объекты и освобождаем их, потому что так принято. Но постойте! Если подумать об обработке исключений, то обнаружится проблема. Ранее мы говорили о том, что можем повторно генерировать исключения в блоке `@catch`. При этом блок `@finally` обязательно выполняется перед повторным генерированием исключения. В свою очередь, это разрушает локальный пул до передачи исключения и приводит к появлению *исключения-зомби* (zombie exception). К счастью, существует простой способ решения этой проблемы: мы просто сохраняем его за пределами нашего пула.

Новая версия метода выглядит следующим образом:

```
- (void)myMethod
{
    id savedException = nil;
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSDictionary *myDictionary =
        [[NSDictionary alloc] initWithObjectsAndKeys:@"asdfads", nil];
    @try {
        [self processDictionary:myDictionary];
    } @catch (NSException *e) {
        savedException = [e retain];
        @throw;
    } @finally {
        [pool release];
        [savedException autorelease];
    }
}
```

Используя сообщение `retain`, мы сохраняем исключение в родительском пуле. Когда пул будет освобожден, мы уже сохраним указатель, а когда родительский пул будет освобожден, исключение будет удалено вместе с ним.

Резюме

В этой главе вы познакомились с методами управления памятью Cocoa: `retain`, `release` и `autorelease`. Мы также обсудили механизм сбора мусора и систему подсчета ссылок (ARC) — новейшее изобретение компании Apple по управлению памятью.

Каждый объект поддерживает счетчик ссылок. Объекты создаются со счетчиком ссылок, равным 1. При захвате объекта его счетчик увеличивается на 1, а при освобождении объекта — уменьшается на 1. Когда счетчик ссылок обнуляется, объект уничтожается. Сначала вызывается метод `dealloc` объекта, а затем его память возвращается системе и может быть использована при создании другого объекта.

Когда объект получает сообщение `autorelease`, его счетчик ссылок немедленно не изменяется; вместо этого объект помещается в пул `NSAutoreleasePool`. При уничтожении этого пула всем находящимся в нем объектам посылается сообщение `release`. При этом у всех объектов, которые были автоматически освобождены, счетчик ссылок уменьшается на 1. Если счетчик обнуляется, объект уничтожается. При работе с AppKit пул автоматического освобождения создается и уничтожается в точно определенные моменты времени, в частности, по окончании обработки события пользователя. В прочих случаях вы отвечаете за создание вашего собственного пула автоматического освобождения (соответствующий код включается в шаблон исходного текста Foundation).

В среде Cocoa есть три правила по работе с объектами и их счетчиками ссылок.

- Когда вы создаете объект с использованием сообщений `new`, `alloc` или `copy`, счетчик ссылок объекта становится равен 1.
- Если вы захватываете объекты при помощи некоторого другого механизма, считайте, что его счетчик ссылок равен 1 и что ему было послано сообщение `autorelease`.
- Если вы захватили объект, в конечном счете обязаны освободить его.

Система ARC обрабатывает сообщения `retain` и `release` автоматически, вставляя их вызовы в код во время его компиляции.

Далее мы познакомимся с методами `init`: и покажем, как заставить ваши объекты действовать.