Аарон Хиллегасс

OBJECTIVE-C

Программирование для iOS и MacOS

- Основы программирования на C и Objective-C
- Работа с Xcode и документацией Apple
- Разработка приложений iOS и Cocoa
- Сотни примеров и практических упражнений

big nerd ranch

ПИТЕР

I

Первые шаги

1. Вы и эта книга

Давайте немного поговорим о вас. Вы Хотите писать приложения для iOS или Mac OS, но прошлый опыт программирования у вас невелик (а может, его и вовсе нет). Ваши друзья хвалили другие мои книги (iOS Programming: 'The Big Nerd Ranch Guide и Cocoa Programming for Mac OS X),но они написаны для опытных программистов. Что делать? Читайте эту книгу.

Есть другие похожие книги, но вам стоит прочитать именно эту. Почему? Я уже давно учу людей писать приложения для iOS и Mac, и я успел понять, что вам необходимо знать на этой стадии вашего путешествии. Я усердно трудился над тем, чтобы собрать эти знания и отбросить все лишнее. В этой книге очень много полезного и очень мало шелухи.

Пожалуй, кому-то мой подход покажется необычным. Вместо того чтобы подробно разжевывать синтаксис Objective-C, я расскажу вам, как работает программирование и что о нем думают опытные программисты.

Из-за выбранного подхода уже на относительно ранних страницах книги излагается довольно серьезный материал. Не ждите, что эта книга станет легким чтивом. Кроме того, почти каждая идея сопровождается экспериментом в программировании.

Сочетание учебных концепций и их немедленного практического применения - лучший способ изучения программирования.

C и Objective-C

Когда вы запускаете программу файл копируется из файловой системы в память компьютера, и компьютер выполняет инструкции, содержащиеся в файле. Человеку эти инструкции совершенно непонятны, поэтому люди пишут компьютерные программы на специальных языках программирования. Язык программирования самого низкого уровня называется ассемблером. На ассемблере программисту приходится описывать каждое действие, выполняемое процессором (то есть мозгом компьютера). Полученный код преобразуется в машинный код (родной язык компьютера).

Программы на ассемблере скучны, длинны и зависимы от процессора (потому что мозг вашего новенького iMac может основательно отличаться от мозга любимого, но изношенного PowerBook). Иначе говоря, если вы захотите запустить программу на компьютере другого типа, вам придется переписать ассемблерный код.

Чтобы программный код можно было легка переносить с одного типа компьютеров на другой, были разработаны, «высокоуровневые языки программирования». С высокоуровневыми языками программисту не нужно думать о

специфике конкретного процессора - он пишет обобщенные инструкции, а специальная программа (называемая компилятором) преобразует этот код в высокооптимизированный машинный код для конкретного процессора. Одним из таких языков является язык C. Программисты C пишут код на языке C, а компилятор C преобразует код C в машинный код.

Язык C был создан в начале 1970-х годов в фирме AT&T. Операционная система Unix, заложенная в основу Mac OS X и Linux, была написана на C с небольшими вставками ассемблерного кода для самых низкоуровневых операций. Операционная система Windows тоже большей частью написана на C.

Язык программирования Objective-C основан на C, но в нем добавлена поддержка объектно-ориентированного программирования. Именно на Objective-C пишутся приложения для операционных систем Apple iOS и Mac OS X.

Как работает эта книга

В этой книге вы узнаете о языках программирования C и Objectivc-C достаточно, для того чтобы научиться писать приложения для устройств iOS и Mac.

Почему я буду сначала учить вас C? Каждый грамотный программист Objective-C должен достаточно глубоко понимать C. Кроме того, многие идеи, которые в Objective-C выглядят довольно сложными, уходят корнями к простым концепциям C. Часто я буду представлять некоторую идею на C, а затем направлять вас к современному аналогу той же идеи на Objective-C.

Предполагается, что вы будете читать эту книгу перед Mac, Выбудете читать объяснения различных идей и проводить практические эксперименты, которые поясняют эти идеи. Эксперименты - обязательная часть обучения. Без них вы не сможете понять материал. Лучший способ изучения программирования - вводить программный код, делать опечатки, исправлять их и постепенно привыкать к основным закономерностям языка. От простого чтения кода и теоретического изучения идей ни вам, ни вашим навыкам программирования особого проку не будет.

В конце каждой главы приводятся упражнения. Они позволят вам дополнительно потренироваться и закрепить только что полученные знания. Я настоятельно рекомендую выполнить столько упражнений, сколько сможете.

Также в конце некоторых глав встречаются разделы, озаглавленные. «Для любознательных». В них содержится углубленное объяснение вопросов, рассмотренных в главе. Эти разделы не являются абсолютно необходимыми для достижения ваших целей, но я надеюсь, что приведенная информация покажется вам интересной и полезной.

У серии Big Nerd Ranch имеется форум, на котором читатели обсуждают книги и содержащиеся в них упражнения. Форум можно найти по адресу http://forums.bignerdranch.com/.

И эта книга, и программирование вообще покажутся вам намного более приятными, если вы владеете навыками слепой печати. Кроме ТОГО, что слепая печать намного ускоряет ввод, она еще и позволяет вам смотреть на экран и в книгу

вместо клавиатуры. Это существенно упрощает выявление ошибок. Навыки слепой печати пригодятся вам на протяжении всей вашей карьеры.

Как живут программисты

Открыв эту книгу, вы решили стать программистом. А раз так, вы должны знать, что вас ждет.

Жизнь программиста в основном состоит из непрестанной борьбы. Решение задач в постоянно меняющемся техническом окружении означает, что программисты всегда изучают что-то новое. Впрочем, «изучение чего-то нового» сказано скорее для приличия: в основном речь идет о борьбе с собственным невежеством». Даже если программист работает со знакомой технологией, создаваемые программы порой настолько сложны, что на обычный поиск ошибки порой уходит целый день.

Если вы собираетесь программировать, вас ждет борьба. Большинство профессиональных программистов приучается бороться час за часом, день за днем, терпеливо и без (особого) раздражения. Это еще один навык, который вам сильно пригодится. Если вас интересует жизнь программистов и ведение современных программных проектов, я рекомендую книгу Скотта Розенберга (Scott RosenЬerg) «Dreamil1g in Code ».

А теперь пора взяться за дело и написать вашу первую программу.

2. Ваша первая программа

Итак, теперь вы знаете, как устроена эта книга. Давайте посмотрим, как же происходит программирование для Mac, iPhone и iPad. Для этого вам предстоит:
• установить средства разработчика фирмы Apple;
• создать простой проект с использованием этих средств;
• познакомиться с тем, как работает инструментарий разработчика. чтобы обеспечить работоспособность нашего проекта.

К концу этой главы вы успешно напишете свою первую программу для Mac.

Установка средств разработчика Apple

Для написания приложений для Mac OS (Macintosh) или iOS (iPhone и iPad) вы будете использовать средства разработчика Apple. Программы можно загрузить по адресу http://developer.apple.com/ или приобрести их в Mac App Store.

После установки средств разработчика найдите папку /Developer в корневом катологе своего жесткого диска. В этой папке содержится все необходимое для разработки приложений для настольных систем Mac OS X и мобильных устроств iOS.

Почти вся наша работа в этой книге будет выполнятся в одном приложении - Xcode из папки /Developer/Applications. (Я бы рекомендовал перетащить значок Xcode на док-панель; вы будете очень часто его использовать)

Знакомство с Xcode

Xcode - *интегрированная среда разработки* (IDE, Integrated Development Environment) фирмы Apple. Это означает, что в Xcode присутствует все необходимое для написания, построения и запуска новых приложений в Xcode.

Одно замечание по поводу терминологии: любые исполняемые файлы на компьютере называются программами. Некоторые программы имеют графический интерфейс; их мы будем называть *приложениями*.

Другие программы не имеют графического интерфейса и целыми днями работают в фоновом режиме: они называются *демонами*. Термин звучит устрашающе, но в самих программах ничего страшного нет. Вероятно, прямо сейчас на вашем Mac

выполняется около 60 демонов. Они ждут своего часа, чтобы помочь вам в выполнении какой-либо операции, Например, один из демонов, работающих в вашей системе, называется **pboard**. Когда вы выполняете операцию копирования/вставки, демон **pboard** хранит копируемые данные.

Некоторые программы не имеют графического интерфейса и выполняются в течение непродолжительного времени в терминальном окне; мы будем называть их *программами командной строки*. В книге мы в основном будем писать программы командной строки, чтобы сосредоточиться на основах программирования, не отвлекаясь на создании и управлении пользовательским интерфейсом.

Итак, сейчас мы напишем в Xcode простую программу командной строки, чтобы вы в общих чертах поняли, как это делается.

В ходе написания программы вам приходится создавать и редактировать некоторые файлы. Xcode хранит информацию об этих файлах в проекте. Запустите Xcode, выберите в меню **File** команду **New**, а затем **New Project**...

Чтобы вам было проще приступить к работе, Xcode предлагает несколько возможных шаблонов проектов. Шаблон выбирается в зависимости от того, какую программу вы собираетесь писать. В левом столбце выберите в разделе **Mac OS X** строку **Application**. Затем выберите значок **Command Line Tool** среди вариантов в области справа.

Нажмите кнопку **Next**.

Присвойте проекту имя **AGoodStart**. Идентификатор компании в упражнениях нашей книги роли не играет, но вам придется ввести его для продолжения работы. Введите строку *BigNerdRanch* или другое обозначение по вашему усмотрению. В раскрывающемся списке **Type** выберите строку **C**, потому что программа будет написана на языке C. Наконец, проследите за тем, чтобы флажок **Use Automatic Reference Counting** был установлен.

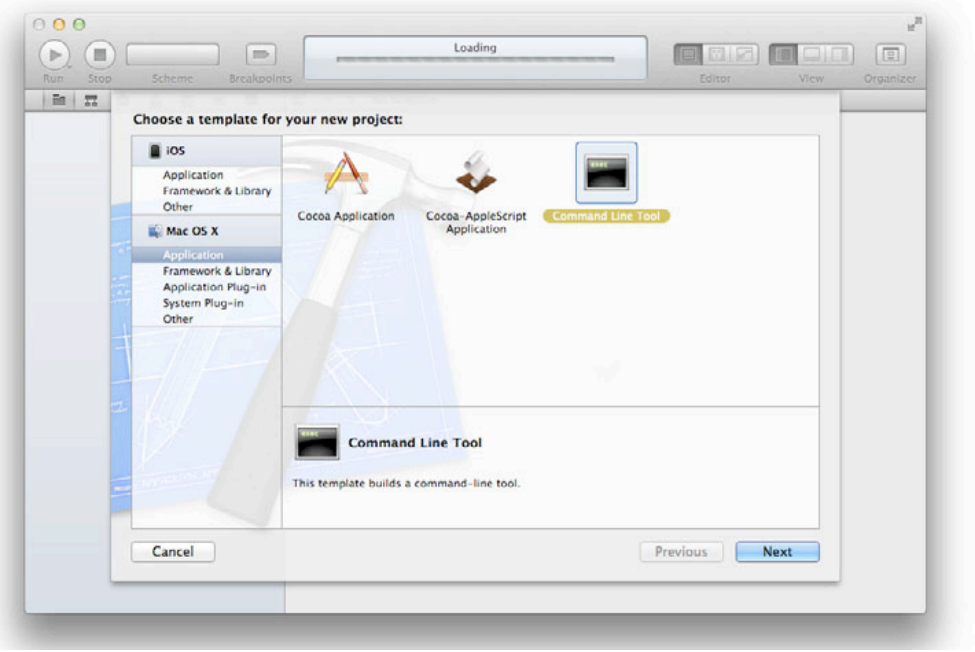


Рис 2.1 Выбор шаблона

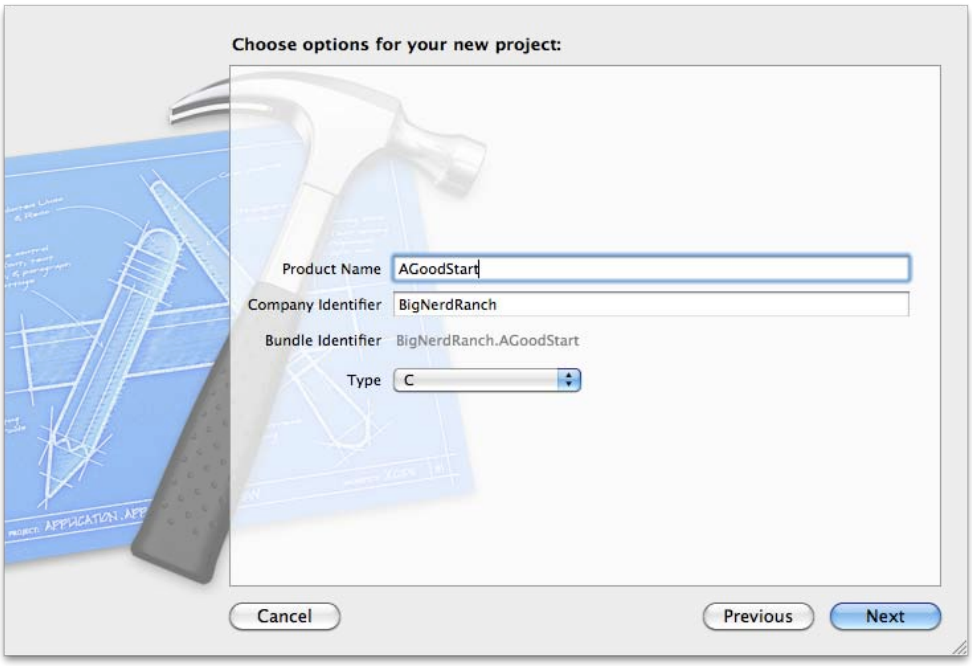


Рис 2.2 Настройка конфигурации

Нажмите кнопку **Next**.

Теперь выберите папку, в которой будет создан каталог проекта. Репозиторий для контроля версий вам не понадобится, этот флажок можно снять. Остается щелкнуть по кнопке **Create**.

Вам предстоит создавать проекты этого типа в нескольких ближайших главах. В будущем я просто напишу: «Создайте новый проект программы командной строки C с именем *таким-то*» - выполните ту же последовательность действий.

(Почему C? Вспомните, что Objective-C построен на базе языка программирования C. Вы должны усвоить основные понятия C, прежде чем мы сможем перейти к специфике Objective-C.)

C чего начать?

После создания проекта на экране появляется окно с информацией `AGoodStart`.

В этом окне приводятся разные подробности: в каких версиях Mac OS X будет запускаться ваше приложение, какие конфигурации должны использоваться при компиляции написанного кода, какие локализации должны применяться в проекте. Но пока не будем обращать на них внимания и найдем простую отправную точку для начала работы.

Найдите в верхней части левой панели файл с именем *main.c* и щелкните на нем. (Если вы не видите строку *main.c*, щелкните на треугольнике рядом с папкой **AGoodStart**, чтобы открыть содержимое папки.)

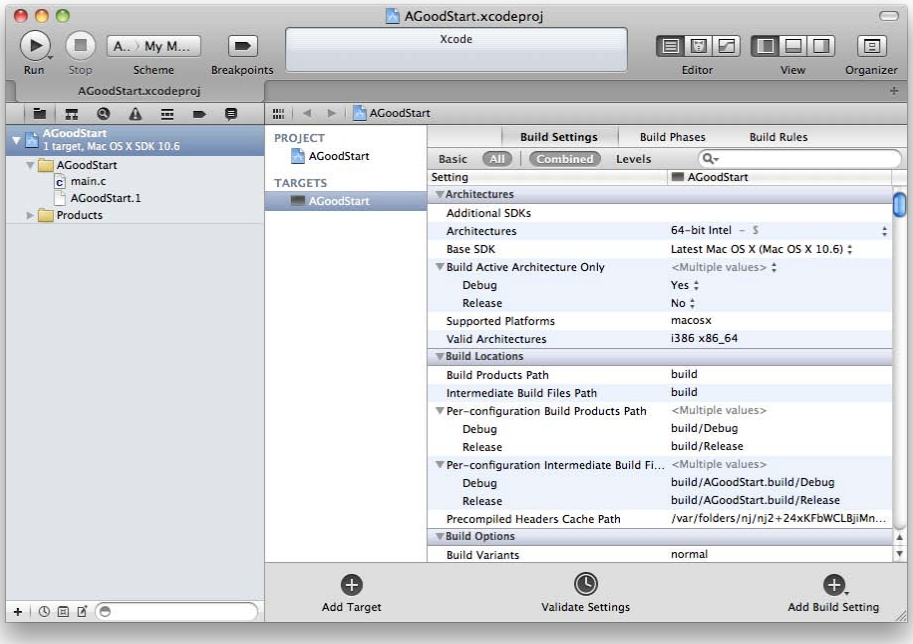


Рис. 2.3. Первый взгляд на проект AGoodStart

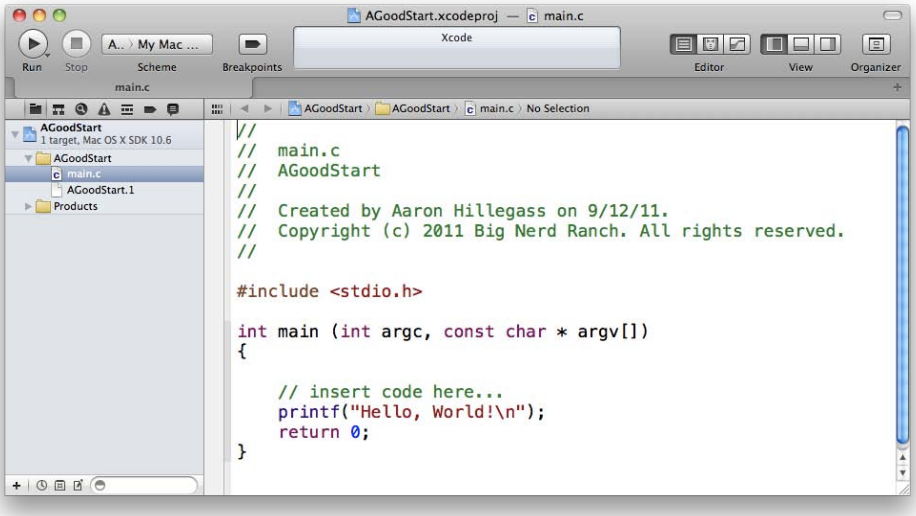


Рис. 2.4. Файл main.c в группе AGoodStart

Обратите внимание: исходное представление с информацией о приложении заменяется содержимым файла *main.c*. Файл *main.c* содержит функцию с именем **main**.

Функция представляет собой список инструкций, которые должен выполнить компьютер. У каждой функции есть имя. В программах на языке C или Objective-C функция **main** вызывается при запуске программы.

```
#include <stdio.h>
int main (int argc, const char * argv[]) {
    // insert code here...
    printf("Hello, World!\n");
    return 0;
}
```

В этой функции присутствует два вида информации: программный код и комментарии

- Программный код - набор инструкций, которые приказывают компьютеру выполнить некоторые действия.
- Комментарии игнорируется компьютером .Мы, программисты, используем их для документированная написанного нами кода. Чем труднее задача, которую вы пытаетесь решить, тем больше пользы от комментариев, описывающих ее решение. Информация становится особенно важной, когда вы вернетесь своей работе через несколько месяцев, посмотрите на код, который вы забыли прокомментировать, и подумаете: « Наверняка очень умная штука, вот только я начисто забыл, как она работает».

В C н Objective-C комментарии можно отличить от кода по двум признакам:

- Если строка начинается с символов //, то все, начиная от этих символов и до канна строки, считается комментарием. Пример - комментарий «insert code here» от фирмы Apple.
- Если вы захотите добавить более обширное описание, используется теобо значения /* и */ для пометки начала и конца комментариев, занимающих несколько строк

Эти правила пометки комментариев являются частью *синтаксиса* C. Синтаксисом называется набор правил, которые определяют, как должен быть написан код на данном языке программирования. Правила синтаксиса в высшей степени точные и конкретные; если вы нарушите их, то ваша программа работать не будет.

Синтаксис комментариев относительно прост, но синтаксис программного кода сильно зависит от того, что и как делает ваш код. Впрочем,одна особенность остается неизменной: каждая команда завершается символом «точка с запятой» (;). (Вскоре мы рассмотрим примеры команд в программном коде). Если забыть об этом символе, в программе появится с синтаксическая ошибка, и она работать не будет.

К счастью, Xcode старается предупредить вас о подобных ошибках. Собственно, одна из первых проблем, с которыми вы столкнетесь как программист, - умение понять, о чем сообщает Xcode при обнаружении ошибок, и исправить эти ошибки. На страницах этой книги мы увидим примеры сообщений Xcode о стандартных ошибках.

Давайте внесем в main.c некоторые изменения. Для начала нам понадобится свободное место. Найдите фигурные скобки({ и }), отмечающие начало и конец функции **main**. Удалите все, что находится между ними.

Теперь отредактируйте файл *main.c*, чтобы он выглядел так, как показано ниже. Добавьте в функцию **main** комментарий, две строки кода и еще один комментарий. Не беспокойтесь, если вы пока не понимаете смысл вводимых символов - мы только начинаем. Впереди целая книга, и вы скоро поймете, что все это означает.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Первые строки романа
```

```
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");

    /*Так ли уж хорошо получилось?
  Возможно, стоит переписать. */

    return 0;
}
```

(Обратите внимание: новый код, который вам нужно ввести, выделен жирным шрифтом. Обычным шрифтом выводится код, который уже находится в файле. Это правило будет соблюдаться во всей книге).

Вероятно, вы заметите, что во время ввода Xcode пытается помочь полезным советом. Эта чрезвычайно удобная функция называется *автозавершением*. Пока вы можете не обращать внимания на подсказки и вводить весь текст самостоятельно. В последующих главах книги попробуйте поэкспериментироватъ с автозавершением и убедитесь в том, что с ним код пишется удобнее и точнее. Различные параметры автозавершения задаются в настройках Xcode, вызываемых из меню Xcode.

Также обращайте внимание на цвет шрифта. Xcode использует разноцветные шрифты, чтобы вам было проще узнать комментарии и разные части кода. (Скажем, комментарии выделяются зеленым цветом.) Цветовое выделение удобно: через некоторое время работы с Xcode вы начнете инстинктивно замечать, когда цвета выглядят неестественно. Часто это указывает на синтаксическую ошибку в написанном коде (например, пропущенный завершитель «точка с запятой»). И чем скорее вы узнаете о допущенной синтаксической ошибке, тем проще ее будет исправить.

Разноцветные шрифты - всего лишь один из способов, которым Xcode сообщает о (возможно) допущенных ошибках.

Как запустить программу?

Когда содержимое файла *main.c* будет соответствовать приведенному ранее, пора запустить программу и посмотреть. что она делает. Этот процесс состоит из двух шагов: Xcode сначала строит вашу программу, а потом запускает ее. В ходе построения Xcode готовит программный код к выполнению. В частности, при этом проверяются синтаксические и другие возможные ошибки.

Снова взгляните на левую панель окна Xcode. Эта область называется *областью навигаци*. В верхней части области навигации находится группа кнопок. В настоящее время на панели отображается *навигатор проекта*, в котором представлены файлы проекта. Кнопка навигатора проекта отмечена значком .

Теперь найдите кнопку и щелкните на ней, чтобы открыть навигатор журнала. Xcode использует журнал (log) для вывода сообщений в ходе построения и запуска вашей программы.

Вы также можете использовать журнал для своих собственных целей. Например, строка программного кода

```
printf ("It was the best of times.\n");
```

приказывает вывести строку « It was the best of times.» в журнале.

Так как вы еще не построили и не запустили программу, в навигаторе журнала ничего нет. В левом верхнем углу окна проекта найдите кнопку, подозрительно похожую на кнопку **Play** в iTunes или на DVD-проигрывателе. Если навести указатель мыши на кнопку, появляется подсказка с надписью «BuiId and then run the current scheme». В терминологии Xcode это означает: «Нажми эту кнопку, и я построю и запущу твою программу».

Если все было сделано правильно, вы будете вознаграждены следующим результатом:

А если нет, результат будет таким:

Что тогда делать? Тщательно сравните свой код с приведенным в книге. Поищите опечатки и пропущенные символы «точка с запятой». Среда Xcode выделяет строки, в которых, по ее мнению, есть ошибки. Обнаружив ошибку, исправьте ее и снова нажмите кнопку `Run`. Повторяйте, пока программа не будет построена успешно.

(Не отчаивайтесь, если построение этого кода или любой другой программы в книге не прошло с первого раза. Когда вы совершаете и исправляете ошибки, это помогает вам лучше понять, что вы делаете. Это даже лучше, чем если вам повезет правильно все сделать с первого раза.)

Когда построение пройдет успешно, найдите в верхней части навигатора журнала объект с меткой `Debug AGoodStart`. Щелкните на нем, чтобы вывести содержимое журнала для последнего запуска программы.

Содержимое журнала может быть довольно длинным. Для нас важна цитата из Диккенса в самом конце вывода. Это результат выполнения вашего кода!

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Tue Jul  5 07:36:45 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
[Switching to process 2723 thread 0x0]
It was the best of times.
It was the worst of times.
```

(На момент написания книги фирма Apple работала над новым отладчиком LLDB. Вскоре он должен заменить GDB, текущий отладчик. Если вы не видите всю информацию GDB, это означает, что стандартным отладчиком Xcode теперь является LLDB. Я вам завидую; так здорово оказаться в будущем!)

Что же такое программа?

Итак, вы построили и запустили свою первую программу. Теперь давайте заглянем вовнутрь. Программа состоит из набора функций. Функция представляет собой набор операций, выполняемых процессором. У каждой функции есть имя; только что написанная нами функция называется main. Также в программном коде используется другая функция - `printf`. Вы не писали ее, но использовали в своем коде. (О том, откуда взялась функция `printf`, будет рассказано в главе 5).

Для программиста функция напоминает кулинарный рецепт: «довести до кипения литр воды. Добавить чашку муки. Подавать горячим». В середине 1970-х годов поваренная книга Бетти Крокер стала продаваться в виде набора карточек с рецептами. Карточка с рецептами - довольно неплохая метафора для функции. Как и у функции каждой карточки есть имя и набор инструкций. Повар выполняет инструкции в рецепте, а компьютер выполняет инструкции в функции.

Рецепты Бетти Крокер Написаны на английском языке. В первой части книги ваши функции будут написаны на языке программирования C. Однако процессор компьютера понимает только инструкции на машинном коде. Откуда он берется?

Вы пишете программу на языке C (который хорошо подходит вам), а *компилятор* преобразует функции вашей программы в машинный код (который хорошо подходит процессору). Компилятор это тоже программа, которую запускает Xcode, когда вы нажимаете кнопку Run. Компиляция и построение программы - одно и то же; я буду использовать эти термины как синонимы.

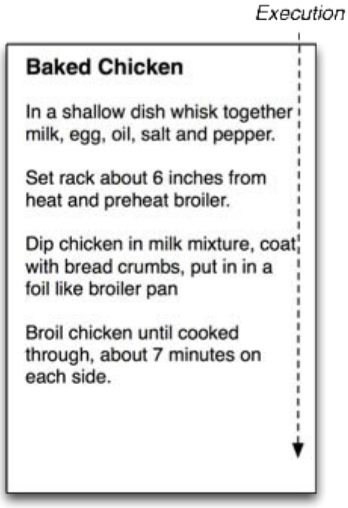


Рис. 2.5. Карточка с рецептом приготовления курицы

При запуске программы откомпилированные функции копируются с жесткого диска в память, и процессор выполняет функцию с именем `main`. Функция `main` обычно вызывает другие функции. Скажем, в нашем примере функция main вызывает функцию `printf`. (О том, как функции взаимодействуют друг с другом, будет рассказано в главе 5.)

Не останавливайтесь

Вероятно, к этому времени вы уже испытали немало разочарований: проблемы с установкой, опечатки, множество незнакомых терминов. А может, вы вообще не понимаете, что здесь происходит. Это все абсолютно нормально.

Моему сыну Отто шесть лет. Отто чувствует себя сбитым с толку по несколько раз в день. Он постоянно пытается усвоить знания, которые не укладываются в его существующие представления. Замешательство наступает так часто, что его это почти не беспокоит. Он никогда не начинает задумываться: «Почему все так сложно? Может, мне лучше отложить эту книгу»

Становясь старше, мы все реже попадаем в тупик - и не потому. что мы все знаем, а потому, что мы склонны уходить от вещей, которые нас озадачивают. Например, читать книгу по истории приятно, потому что мы находим в неи кусочки знаний, которые можем легко разложить по своим умственным полкам. Это простые знания.

Изучение нового языка - пример трудного знания. Вы знаете, что миллионы людей без всяких проблем говорят на этом языке, однако в ваших устах он кажется невероятно странным и неуклюжим. И когда другие пытаются заговорить с нами, мы часто теряем дар речи.

Изучение программирования тоже относится к трудному знанию. Время от времени вы будете чувствовать себя сбитым с толку особенно на первых порах. Это нормально. Собственно, это даже занятно - словно вам снова шесть лет.

Не бросайте книгу; я обещаю. что все трудности рассеются к последней странице.

II

Как работает программирование

3. Переменные и типы

Продолжим метафору с рецептами из предыдущей главы: иногда шеф-повар заводит в кухне настенную доску для записей. Например, распаковывая индейку, он находит наклейку с надписью: «14,2 фунта». Прежде чем выкинуть упаковку, он пишет на доске: «`weight` (вес) = 14,2». А потом, перед тем как ставить индейку в духовку, он вычисляет время приготовления (15 минут + 15 минут на каждый фунт веса), обращаясь к записям на доске.

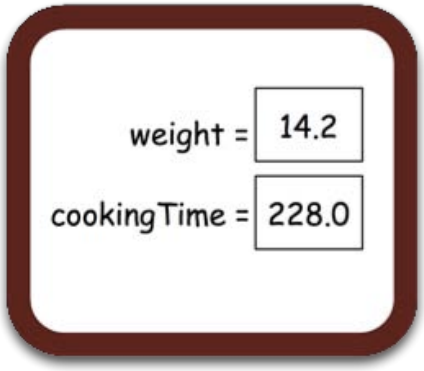


Рис. 3.1. Данные, записанные на доске

Во время выполнения программе часто требуется место для хранени данных, которые будут использоваться позднее Место в котором можно разместить один кусок данных, называется *переменной*. У каждой переменной есть имя (например, `cookingTime` (время приготовления)) и *тип* (скажем, число). Кроме того, во время выполнения программы переменная принимает некоторое значение (например, 228.0).

Типы

При создании новой переменной в программе необходимо *объявить* ее тип и имя. Пример объявления переменной:

```
float weight;
```

Переменная объявляется с типом `float` и именем `weight`. На этот момент переменная еще не имеет значения.

В С тип каждой переменной в программе должен быть объявлен при ее создании. Для этого есть две причины:

- Информация о типе позволяет компилятору проверить вашу работу и сообщить о возможных ошибках или проблемах, Предположим, вы создали переменную типа, предназначенного для хранения текста. Если вы попытаетесь вычислить логарифм программы, компилятор скажет вам что-нибудь вроде: «Для этой переменной вычисление логарифма не имеет смысла» .
- Тип сообщает компилятору, сколько памяти (в байтах) следует выделить для хранения переменной.

Ниже приведена краткая сводка наиболее распространенных типов. Все они будут подробно рассмотрены

short, int, long	Все три типа используются для хранения целых чисел. Тип short обычно на несколько байтов короче long, а тип int находится где-то посередине. В переменной типа long можно сохранить намного большее число, чем в short
float, double	Тип float предназначен для хранения вещественных чисел-то есть чисел, которые могут иметь дробную часть. В памяти данные float представляются двумя числами: мантиссой н экспонентой. Например, число 346,2 хранится в виде 3,462 x 10. Тип double используется для хранения чисел двойной точности; как правило, в них выделяется больше разрядов для хранения мантиссы и поддерживаются большие значения экспоненты
char	Однобайтовое целое число, которое обычно интерпретируется как символ - например, буква 'a'
указатели	В указателе хранится адрес памяти. Указатель объявляется при помощи символа «звездочка»(*). Например, переменная, объявленная как int *, может использоваться для хранения адреса памяти, по которому хранится переменная типа int. Само число в указателе не хранится, но зная адрес int, можно легко перейти к значению. Указатели чрезвычайно полезны, и мы непременно рассмотрим их подробнее... Намного подробнее
struct	Структурный тип (или проще - структура) складывается из других типов. Вы можете создавать собственные определения struct. Представьте, что вам нужен тип GeoLocation, представляющий точку земной поверхности, и в нем должны храниться два числа: широта и долгота. В таком случае вы определяете структуру

Каждый программист C постоянно использует эти типы в своей повседневной работе. Вы не поверите, какие сложные идеи могут быть выражены этими пятью простыми типами.

Программа с переменными

Возвращаемся в Xcode: пора создать следующий проект. Для начала закройте проект `AGoodStart`, чтобы новый код случайно не был введен в старом проекте.

Теперь создайте новый проект (File → New → New Project...).

Найдите в навигаторе проекта файл *main.c* и откройте его. Отредактируйте *main.c* так, чтобы его содержимое выглядело следующим образом:

```
#include stdio.h
int main (int argc, const char * argv[])
{
    // Объявление переменной 'weight' типа float
    float weight;

    // Сохранение числа в переменной
    weight = 14.2;

    // Вывод информации для пользователя
    printf("The turkey weighs %f.\n", weight);

    // Объявление другой переменно типа float
    float cookingTime;

    // Вычисление времени приготовления
    // В данном случае символ, '*' озночает умножить на
    cookingTime = 15.0 + 15.0 * weight;

     // Вывод информации для пользователя
    printf("Cook it for %f minutes.\n", cookingTime);
    // Успешное завершение функции
return 0;
}
```

Постройте и запустите программу. Либо щелкните на кнопке Run в левой верхней части окна Xcode, либо воспользуйтесь комбинацией клавиш Command+R. Затем щелкните на кнопке , чтобы открыть навигатор журнала. Выберите объект с меткой `Debug Turkey`, чтобы перейти к результату работы программы.

Он должен выглядеть так:

```
The turkey weighs 14.200000.
Cook it for 228.000000 minutes.
```

Теперь щелкните на кнопке , чтобы вернуться к навигатору проекта. Выберите файл *main.c* - в окне Xcode снова появляется программный код. Давайте посмотрим, что же мы сделали.

В строке:

```
float weight;
```

мы приказываем «объявить переменную `weight` с типом `float`».

В следующей строке переменная получает значение:

```
weight = 14.2;
```

Эта команда копирует данные в переменную. Программисты говорят, что «переменной присваивается значение 14.2».

В современном языке C можно совместить объявление переменной и присваивание начального значения в одной строке:

```
float weight = 14.2;
```

А вот другая команда присваивания:

```
cookingTime = 15.0 + 15.0 * weight;
```

Справа от знака = записано *выражение* - нечто такое, что можно вычислить и получить результат (значение). Вообще говоря, в каждой команде присваивания справа от знака = находится выражение.
Например, в строке:

```
weight = 14.2;
```

выражением является число 14.2.

Переменные используются во всех программах. В этой главе мы лишь в самых общих чертах познакомились с переменными.

Вам еще предстоит узнать много нового о том, как работают переменные . и как их использовать в программах.

Упражнение

Создайте новую программу командной строки C с именем `TwoFloats`. В функции `main()` объявите две переменные типа `float` и присвойте каждой число с дробной частью - например, 3.14 или 42.0. Объявите другую переменную типа `double` и присвойте ей сумму двух переменных типа `float`. Выведите результат при помощи `printf()`. За примерами синтаксиса обращайтесь к коду, приведенному в этой главе.

4. If/else

Одна из важнейших идей программирования - выполнение разных действий в зависимости от обстоятельств. Все ли обязательные поля заполнены на форме заказа? Если все, то разблокировать кнопку `Submit`. Остались ли у игрока неиспользованные жизни? Если остались, продолжить игру, а если нет - отобразить картинку с могилкой и воспроизвести печальную музыку.

Такое поведение реализуется с использованием конструкции `if /else`, которая имеет следующий синтаксис:

```
if (conditional) {
    // Этот код выполняется, если условие истинно (true)
} else {
   // Этот код выполняется, если условие ложно (false)
}
```

В этой главе мы не будем создавать проект. Тщательно просмотрите код примеров, обращая внимание на то, что вы узнали в двух предыдущих главах.
Пример кода с использованием `if /else`:

```
float truckWeight = 34563.8;
// Порог не превышен?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
} else {
    printf("It is a heavy truck\n");
}
```

Если секция `else` не нужна, ее можно не указывать:

```
float truckWeight = 34563.8;
// Порог не превышен?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
}
```

Результатом условного выражения всегда является одно из двух значений: `true` (истина) или `false` (ложь). В языке С было решено, что `false` будет представляться 0, а любые не нулевые значения будут считаться равными `true`,

В приведенном примере с обеих сторон от оператора < стоят числа. Если число слева меньше числа справа, то результат выражения равен 1 (очень распространенный способ представления истинности). Если число слева больше либо равно числу справа, то результат выражения равен О(единственный способ представления ложности)

Подобные операторы часто встречаются в условных выражениях. В табл. 4.1 представлены основные операторы, используемые при сравнении чисел (и других типов, которые интерпретируются компьютером как числа).

Таблица 4.1. Операторы сравнения

<	Число слева меньше числа справа?
>	Число слева больше числа справа?
<=	Число слева меньше либо равно числу справа?
>=	Число слева больше либо равно числу справа? == Числа равны?
!=	Числа не равны?

Оператор == заслуживает особого упоминания: в программировани и оператор == используется для проверки равенства. Как вы уже знаете, одиночный знак равенства = используется для присваивания значений. Очень, очень многие ошибки происходят из-за того, что программисты вводят = вместо ==. Так что лучше отвыкайте думать о = как о «знаке равенства» отныне это «оператор присваивания.

В некоторых условных выражениях необходимы логические операторы. Представьте, что вам нужно проверить, входит ли число в некоторый диапазон - скажем, больше 0 и меньше 40 000? Для определения диапазонов можно использовать логический оператор AND (&&):

```
if ((truckWeight > 0.0) && (truckWeight < 40000.0)) {
    printf("Truck weight is within legal range.\n");
}
```

Три логических оператора приведены в табл. 4.2,

Таблица 4.2. Логические операторы

&&	Логический оператор AND -- true только в том случае, если оба значения равны true
||	Логический оператор OR -- false только в том случае, если оба значения равны false
!	Логический оператор NOT -- true превращается false, и наоборот

Если у вас уже имеется опыт программирования на другом языке, учтите, что логического оператора «исключающего OR» в Objective-C нет, поэтому здесь он не упоминается.

Логический оператор NOT (!) вычисляет отрицание выражения, приведенного в скобках справа:

```
// Значение входит в разрешенный диапозон?
if (!((truckWeight > 0.0) && (truckWeight < 40000.0))) {
```

```
    printf("Truck weight is not within legal range.\n");
}
```

Логические переменные

Как видите, выражения довольно быстро становятся длинными и сложными. Иногда бывает удобно поместить значение выражения в переменную с подходящим именем:

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight <  40000.0));
if (isNotLegal) {
    printf("Truck weight is not within legal range.\n");
}
```

Переменная, которая может принимать значения `true` и `false`, называется *логической переменной*. Традиционно программисты C всегда хранила логические значения в переменных типа `int`. Программисты Objective-C обычно используют для логических переменных типа `BOOL`, поэтому в приведенном фрагменте был выбран именно он. (Впрочем `BOOL` - всего лишь синоним для целого типа.) Чтобы использовать тип `BOOL` в функции C, необходимо включить соответствующий заголовок:

```
#include <objc/objc.h>
```

Небольшое замечание по поводу синтаксиса: если код, следующий за условным выражением, состоит всего из одной команды, то фигурные скобки необязательны. Таким образом, следующий фрагмент эквивалентен предыдущему:

```
BOOL isNotLegal = ! ((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal)
      printf ("Truck weight is not within legal range. \n") ;
```

Но если код состоит не из одной, а из нескольких команд, фигурные скобки обязательны

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal) {
printf ("Truck weight is not within legal range. \n");
printf("Impound truck.\n");
}
```

Почему? Представьте, что мы убрали фигурные скобки

```
BOOL isNotLegal = ! ((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal)
printf( "Truck weight is not within legal range. \n");
printf( "Impound truck. \n");
```

В этом случае команда вторая команда с вызовом `printf` будет выполняться всегда, независимо от результата проверки. Когда компилятор не обнаруживает фигурную скобку после условия, он считает, что в команду `if` входит только следующая команда. Следовательно, вторая команда `printf` будет выполняться всегда. (А как же отступ? Отступы очень полезны для нас, людей, читающих код; для компилятора они не значат ничего.)

else if

А если возможностей больше двух? Можно проверить их одну за одной при помощи конструкции `else if`. Представьте, что грузовик по весу принадлежит к одной из трех категорий: невесомый(floating), легкий(light) или тяжелый(heavy).

```
if (truckWeight <= 0) { printf("A floating truck\n");
} else if (truckWeight < 40000.0) { printf("A light truck\n");
}else {
printf("A heavy truck\n");
}
```

Условий `else if` может быть сколько угодно. Они будут проверяться в порядке следования до тех нор, пока результат одного из условий не окажется равным `true`. Порядок следования важен. Обязательно расположите условия так, чтобы избежать ложных положительных срабатываний. Например, если поменять, местами первые два условия в приведенном примере, невесомые грузовики никогда не будут найдены, потому что они будут перехватываться условиями для легких грузовиков. Завершающая секция `else` не обязательна, но она пригодится, если вы хотите обработать все значения не подходящие ни под одно из перечисленных условий.

Для любознательных: условный (тернарный) оператор

Конструкции `if/else` довольно часто используются для присваивания значения переменным. Возьмем для примера следующий код:

```
int minutesPerPound; if (isBoneless)
minutesPerPound = 15;
else
minutesPerPound = 20;
```

Каждый раз, когда значение присваивается переменной в зависимости от некоторого условия, знайте, что у вас имеется возможный кандидат для применения *условного оператора* `?`. (его также часто называют *тернарным оператором*).

```
int minutesPerPound = isBoneless ? 15 : 20;
```

Эта строка эквивалентна всему предыдущему фрагменту. Вместо `if` и `else` достаточно написать единственную команду присваивания. Часть перед `?` - проверяемое: условие, а значения после `?` - альтернативы для истинного и ложного результата проверки.

Если эта запись кажется вам странной, ничто не мешает продолжать пользоваться `if` и `else`. Полагаю, что со временем вы оцените тернарный оператор как компактную и точную запись для условного присваивания значений. И что еще важнее - вы увидите. как он используется другими программистами, и сможете понять увиденное!

Упражнение

Имеется следующий фрагмент кода:

```
int i =20;
int i =25;

int k = ( i > j ) ? 10 : 5;

if (5 < j -k){// первое выражение
      printf ("The first expression is true. ") ;
} else if ( j > i ) { // второе выражение
      printf ("The second expression is true. ") ;
}else {
      printf("Neither expression is true.");
}
```

Что будет выведено на консоль?

5. Функции

В главе 3 я объяснил, что такое такое переменная: имя, связанное с блоком данных. Функция представляет собой имя, связанное с блоком кода. Функции можно передать информацию. Функцию можно выполнить. Ее можно выполнить. Ее можно заставить вернуть нам результат

Функция имеют основополагающее значение в программировании, поэтому эта глава получилось длинной - три новых проекта, новая программа, много новых идей. Начнем с упражнения, которое наглядно показывает, для чего нужны функции.

Когда использовать функцию?

Предположим, вы пишите программу, которая поздравляет студентов с прохождением учебного курса. Прежде чем думать о том, как прочитать список студентов из базы данных или напечатать сертификаты на бланках дипломов, вы хотите по экспериментировать с сообщением, которое должно печататься на сертификатах.

Создайте для экспериментов новый проект, программу командной строки C с именем `ClassCertificates`

Первая идея выглядит примерно так:

```
int main (int argc, const char * argv[])
{
     printf("Mark has done as much Cocoa Programming as I could fit into 5 day
\n");
     printf("Bo has done as much Objective-C Programming as I could fit into 2
days\n");
    printf("Mike has done as much Python Programming as I could fit into 5 days
\n");
      printf("Ted has done as much iOS Programming as I could fit into 5 days
\n");
return 0;
}
```

Вас беспокоит необходимость вбивать все эти сообщения? Они раздражают, вас своим однообразием? Поздравляю, у вас задатки отличного программиста. Когда вам приходится помногу раз делать нечто очень похожее (в данном случае - набирать сообщения в команде `printf`), подумайте, нельзя ли воспользоваться функцией для достижения того же результата.

Как написать и использовать функцию?

Итак, нам нужна функция; остается понять, как ее написать. Откройте файл *main.c* в проекте `ClassCertificates` и введите новую функцию перед функцией `main`. Этой функции будет присвоено имя `congratulateStudent`.

```
#include <stdio.h>
void congratulateStudent(char *student, char *course, int numDays)
{
    printf("%s has done as much %s Programming as I could fit into %d days.\n",
        student, course, numDays);
}
```

(Вы спрашиваете, что означают эти загадочные `%s` и `%d`? Потерпите; мы поговорим о них в следующей главе.)

Теперь отредактируйте функцию `main`, чтобы в ней использовалась новая функция:

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Mark", "Cocoa", 5);
    congratulateStudent("Bo", "Objective-C", 2);
    congratulateStudent("Mike", "Python", 5);
    congratulateStudent("Ted", "iOS", 5);
return 0;
}
```

Постройте и запустите программу. Вероятно, вы получите предупреждение, помеченное восклицательным знаком в желтом треугольнике. Предупреждения в Xcode не мешают выполнению вашей программы; они всего лишь привлекают ваше внимание к возможной проблеме. Текст предупреждения выводится справа от кода. В этом предупреждении говорится что-то вроде «Нет предшествующего прототипа для функции 'congratulateStudent'». Пока не обращайте внимания, мы вернемся к предупреждению в конце раздела.

Найдите вывод программы в навигаторе журнала, Он должен быть идентичен тому, что вы бы увидели, если бы каждая строка была введена вручную.

```
Mark has done as much Cocoa Programming as I could fit into 5 days.
Bo has done as much Objective-C Programming as I could fit into 2 days.
Mike has done as much Python Programming as I could fit into 5 days.
Ted has done as much iOS Programming as I could fit into 5 days.
```

Подумайте, что здесь произошло. Мы заметили повторяющуюся закономерность. Мы извлекли все общие характеристики проблемы (повторяющийся текст) и переместили их в отдельную функцию. Осталось разобраться с различиями (имя студента, название курса, количество дней). Для них в функцию были включены три параметра. Давайте еще раз посмотрим на ту строку, в которой присваивается имя функции.

```
void congratulateStudent(char *student, char *course, int numDays)
```

Каждый параметр состоит из двух частей: типа данных и имени параметра. Параметры разделяются запятыми и перечисляются в круглых скобках справа от имени функции.

А что означает `void` слева от имени функции? Это тип информации, возвращаемой функцией. Если функция не возвращает ничего, используйте ключевое слово `void`. Мы поговорим о возвращаемых значениях функций позднее в этой главе.

Новая функция использовалась (или *вызывалась*) в функции `main`. При вызове `congratulateStudent` мы передаем этой функции значения, которые называются аргументами. Значение аргумента присваивается соответствующему параметру. Имя параметра используется внутри функции как переменная, содержащая переданное значение.

Рассмотрим конкретный пример. В первом вызове `congratulateStudent` передаются три аргумента: "Mark", "Cocoa" и 5.

```
congratulateStudent("Mark", "Cocoa", 5);
```

Пока займемся третьим аргументом. Переданное функции `congratulateStudent` значение 5 присваивается третьему параметру (`numDays`). Аргументы присваиваются параметрам в порядке следования, при этом значения должны иметь тот же (или очень близкий) тип. У нас 5 - целое число, а параметр `numDays` имеет тип `int`... Подходит.

Теперь, когда вы написали эту удобную функцию, се можно будет использовать в других программах. Вносить изменения тоже будет проще - достаточно изменить формулировку фразы внутри функции, И изменения немедленно вступят в силу: повсюду, где вызывается эта функция.

Еще раз посмотрите на первую версию `ClassCertificates` с однообразными командами. Зачем мы использовали функцию? Что бы не набирать лишние символы на компьютере? Да, но это определенно не все. Не стоит забывать и о проверке ошибок. Чем меньше вы набираете , и чем больше за вас работает компьютер, тем меньше вероятность опечатки. Если вы ошибетесь при вводе имени функции. Xcode предупредит вас, но Xcode понятия не имеет об ошибках при вводе обычного текста.

Еще одно преимущество функции - возможность повторного использования. Теперь, когда вы написали эту удобную функцию, ее можно будет использовать в других программах. Вносить изменения тоже будет проще - достаточно изменить формулировку фразы внутри функции, и изменения немедленно вступят в силу повсюду, где вызывается эта функция.

И последнее преимущество Функций: если в функции вдруг допущена ошибка, вы исправляете эту одну функцию, и все се вызовы вдруг начинают работать правильно. Короче говоря, разбиение кода на функции упрощает его понимание и сопровождение.

А теперь вернемся к предупреждению в коде. На практике функция очень часто *объявляется* в одном месте, а *определяется* в другом. Объявление функции просто предупреждает о том, что где-то дальше имеется функция с таким именем. В определении функции вы перечисляете те действия, которые эта функция должна выполнять. В нашем упражнении функция объявлялась и определялась в одном месте. Так как это нетипично, Xcode выдает предупреждение, если функция не была объявлена заранее.

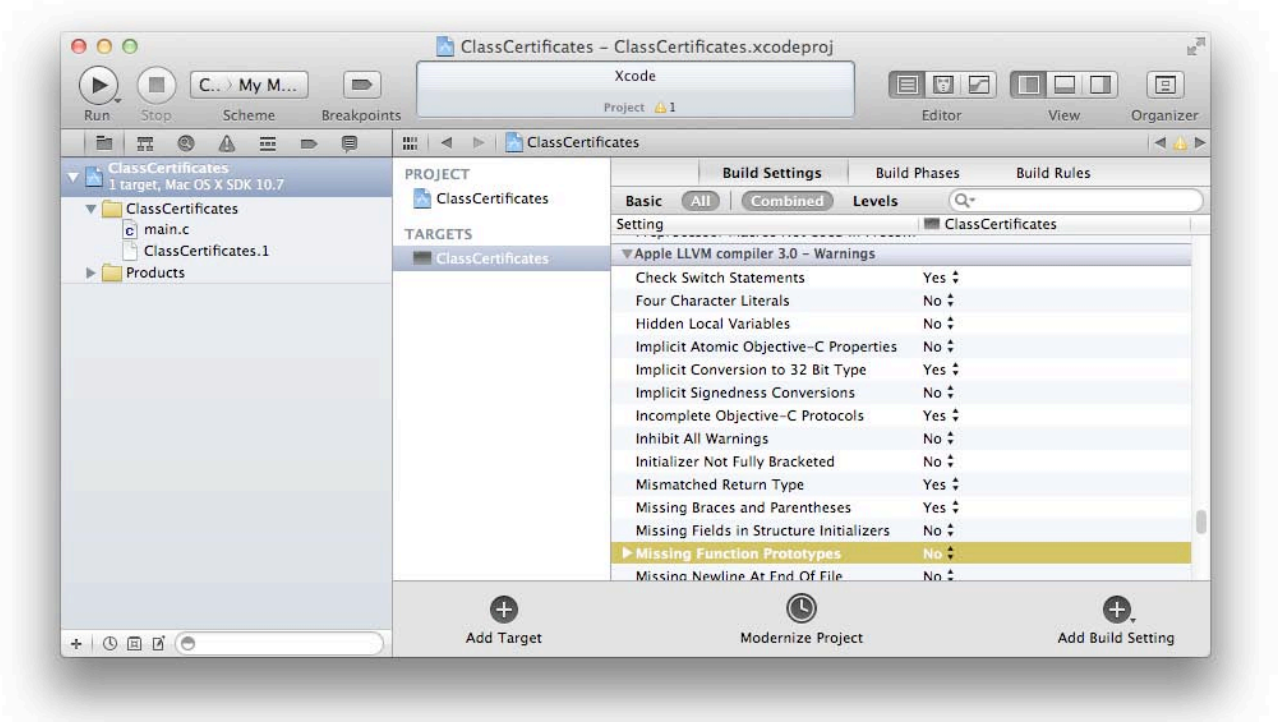


Рис. 5.1. Отключение предупреждений о прототипах функции

Работая над проектами, которые мы будем создавать в этой книге, на предупреждения можно просто не обращать внимания. А если хотите, отключите это предупреждение - выберите объект `ClassCertificates` в верхней части навигатора проекта. На панели редактора выберите раздел `All` на вкладке `Build Settings`. Прокрутите разные варианты настройки построения приложений, найдите строку `Missing Function Prototypes` и измените ее значение на `No`.

Как функции работают друг с другом

Программа - это набор функций. Когда вы запускаете программу, эти функции копируются с жесткого диска в память, процессор находит функцию с именем `main` и выполняет ее.

Вспомните, как мы сравнивали функцию с карточкой рецепта. Начиная выполнять рецепт «Печеная курица», я могу обнаружить, что вторая инструкция требует «Приготовить панировку»; о том, как это сделать, объясняется на другой

карточке. Программист в таком случае скажет: «Функция "Печеная курица" вызывает функцию "Панировка"».

Точно так же и функция main может вызывать другие функции. Например, функция main из проекта `ClassCertificates` вызвала функцию `congratulateStudent`, которая в свою очередь вызвала `printf`.

Во время приготовления панировки вы перестаете выполнять рецепт с карточки «Печеная курица», а когда панировка будет готова - снова возвращаетесь к карточке «Печеная курица» и продолжаете ее выполнение. Так же и функция **main** прекращает выполняться и «приостанавливается» до тех пор, пока не будет выполнена вызванная ей функция.

Чтобы увидеть, как это происходит, мы воспользуемся функцией `sleep`, которая не делает ничего, а только ожидает некоторое время. Включите в функцию main вызов `sleep`.

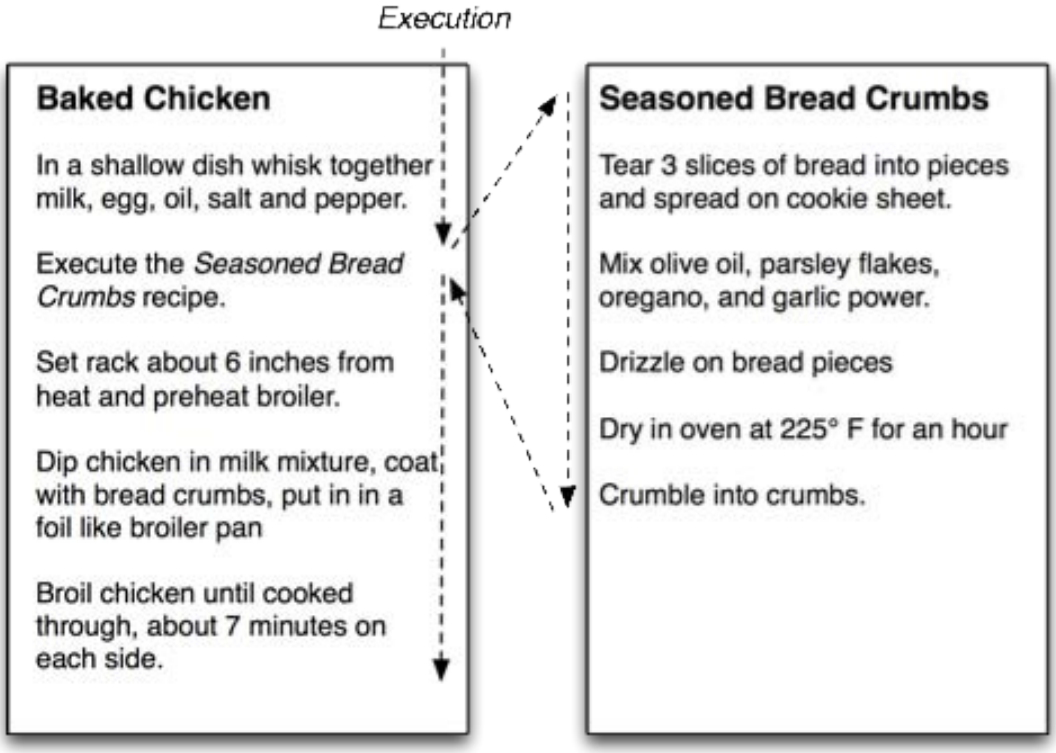


Рис 5.2. Карточки с рецептами

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Mark", "Cocoa", 5);
sleep(2);
    congratulateStudent("Bo", "Objective-C", 2);
sleep(2);
    congratulateStudent("Mike", "PHP and PostgreSQL", 5);
sleep(2);
    congratulateStudent("Ted", "iOS", 5);
return 0;
}
```

Постройте и запустите программу. (На предупреждение об отсутствии объявления пока не обращайте внимания.) Вы увидите, что перед каждым сообщением следует 2-секундная пауза. Это происходит из-за того, что функция main перестает выполняться до того момента, как вызванная функция `sleep` завершит свою работу.

Обратите внимание: при вызове функции указывается имя и круглые скобки для аргументов. Соответственно когда функция упоминается в тексте, за ее именем тоже обычно следует пара круглых скобок. С этого момента, говоря о функции **main**, мы будем использовать запись `main`().

Ваш компьютер уже содержит много встроенных функций. Вообще-то это не совсем точно - вот вам самая настоящая правда: до того, как на вашем компьютере была установлена система Mac OS X, он был не чем иным, как дорогостоящим нагревателем. В частности, вместе с Mac OS X были установлены файлы содержащие набор заранее откомпилированных функций. Эти наборы функций называются *стандартными библиотеками*; функции `sleep`() и `printf` () входят в эти стандартные библиотеки.

Вначале файла *main*.c находится директива включения файла `stdio.h`. В нем содержится объявление функции `printf`(), по которому компилятор может проверить правильность использования функции. Функция `sleep`() объявлена в файле `stdlib.h`. Включите и этот файл, что бы компилятор перестал жаловаться на отсутствие объявления `sleep`()

```
#include <stdio.h>
#include <stdlib.h>

void congratulateStudent(char *student, char *course, int numDays)
{
      ...
```

Стандартные библиотеки приносят двойную пользу

- В них собраны большие фрагменты кода, который вам не придется писать, и сопровождать самостоятельно. Соотственно, с ним вы сможете создавать намного большие и качественные программы, чем без них.

- Стандартные библиотеки помогают выдержать общий стиль оформления и поведения многих программ.

Программисты проводят много времени за изучением стандартных библиотек операционных систем, в которых они работают. Каждая компания, создавшая операционную систему, также создает документацию для прилагаемых к ней стандартных библиотек. О том, как просматривать документацию для Mac OS и iOS, рассказано в главе 15,

Локальные переменные, кадры и стек

В каждой функции могут определяться локальные переменные. Так называются переменные, объявленые внутри функции. Они существуют внутри только во время выполнения этой функции, и обращаться к ним можно только в пределах этой функции. Например, представьте. что мы пишем функцию для вычисления продолжительности приготовления индейки. она может выглядеть так:

```
void showCookTimeForTurkey(int pounds)
{
   int necessaryMinutes = 15 + 15 * pounds;
   printf("Cook for %d minutes.\n", necessaryMinutes);
}
```

Здесь `necessaryMinutes` - локальная переменная. Она начинает свое существование с началом выполнения `showCookTimeForTurkey()` и уходит в небытие после завершения выполнения этой функции. Параметр функции `pounds` тоже является локальной переменной. Параметр представляет собой локальную переменную, инициализированную значением соответствующего аргумента.

Функция может иметь несколько локальных переменных; все они хранятся в *кадре* этой функции. Представьте себе кадр в виде настенной доски, на которой можно писать во время работы функции. Когда выполнение функции завершается, с доски все стирается.

Допустим, мы работаем с рецептом печеной курицы. В вашей кухне у каждого рецепта имеется собственная доска для записи, поэтому доска с рецептом печеной курицы уже готова. Теперь при выполнении рецепта панировки вам понадобится новая доска. Где ее разместить? Прямо поверх доски с рецептом печеной курицы. В конце концов, выполнение этого рецепта было приостановлено, пока не будет готова панировка. Рецепт печеной курицы (кадр стека) вам все равно не понадобится, пока рецепт приготовления панировки не будет выполнен, а его кадр не освободится. Кадры, наложенные друг на друга, образуют стопку, или стек.

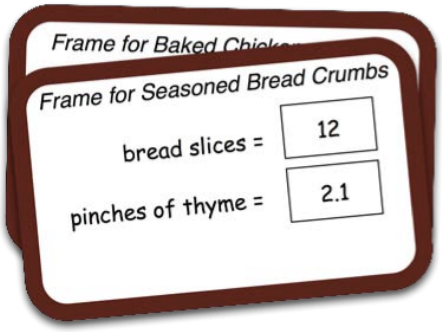


Рис. 5.3. Две доски с рецептами

Программисты говорят: «При вызове функции ее кадр создается на вершине *стека*. При завершении функции ее кадр извлекается из стека и уничтожается».

Давайте поближе познакомимся с работой стека, поместив функцию showCookTimeForTurkey() в гипотетическую программу:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
}
int main(int argc, const char * argv[])
{
    int totalWeight = 10;
    int gibletsWeight = 1;
    int turkeyWeight = totalWeight - gibletsWeight;
    showCookTimeForTurkey(turkeyWeight);
    return 0;
}
```

Вспомните, что функция main() всегда выполняется первой. Функция main() вызывает функцию showCookTimeForTurkey(), которая начинает выполняться. Как будет выглядеть стек программы сразу же после умножения pounds на 15?

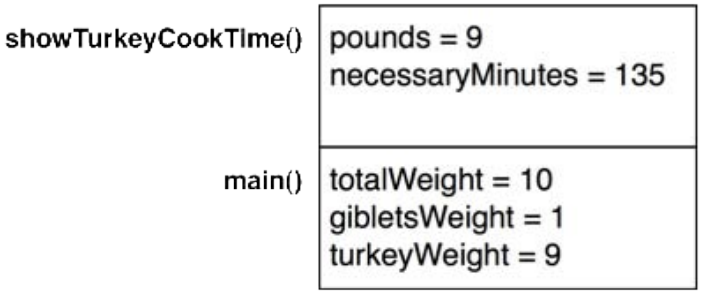


Рис 5.4. Два кадра в стеке

Стек работает по принципу «последним пришел, первым вышел». Иначе говоря showCookTimeForTurkey() извлекает свой кадр из стека до того как main() извлечет свой кадр.

Обратите внимание: pounds, единственный параметр showCookTimeForTurkey(), является частью кадра. Вспомните, что параметр представляет собой локальную переменную, которой присваивается значение соответствующего аргумента. В нашем примере переменная turkeyWeight со значением 9 передается в аргументе showCookTimeForTurkey(). Ее значение присваивается параметру pounds и копируется в кадр функции.

Рекурсия

Может ли функция вызвать сама себя? А почему бы и нет? Это называется *рекурсией*. Возможно, вы слышали скучную длинную песню «99 бутылок пива». Создайте новую

программу командной строки C с именем `BeerSong`. Откройте файл *main*.c, добавьте функцию для вывода слов песни и вызовите ее из `main()`:

```
#include <stdio.h>
void singTheSong(int numberOfBottles)
{
    if (numberOfBottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n");
    } else {
        printf("%d bottles of beer on the wall. %d bottles of beer.\n",
               numberOfBottles, numberOfBottles);
        int oneFewer = numberOfBottles - 1;
         printf("Take one down, pass it around, %d bottles of beer on the wall.
\n",
               oneFewer);
        singTheSong(oneFewer); // This function calls itself!
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               numberOfBottles);
      }
}

int main(int argc, const char * argv[])
{
    singTheSong(99);
return 0;
}
```

Постройте и запустите программу. Результат будет выглядеть так;

```
99 bottles of beer on the wall. 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.
98 bottles of beer on the wall. 98 bottles of beer.
Take one down, pass it around, 97 bottles of beer on the wall.
97 bottles of beer on the wall. 97 bottles of beer.
...
1 bottles of beer on the wall. 1 bottles of beer.
Take one down, pass it around, 0 bottles of beer on the wall.
There are simply no more bottles of beer on the wall.
Put a bottle in the recycling, 1 empty bottles in the bin.
Put a bottle in the recycling, 2 empty bottles in the bin.
...
Put a bottle in the recycling, 98 empty bottles in the bin.
Put a bottle in the recycling, 99 empty bottles in the bin.
```

Как выглядит стек, когда со стены снимается последняя бутылка?

Вообще-то кадры и стек обычно не рассматриваются в начальном курсе программирования, но мой опыт показал. что эти идеи чрезвычайно полезны для неопытных программистов. Во-первых, они дают более конкретное понимание ответов на вопросы типа: «А что происходит с моими локальными переменными, завершает выполнение?» Во-вторых, они помогают разобраться в работе *отладчика* - программы, которая показывает, что происходит вашей программе (а эта информация

упрощает поиск и исправление ошибок). Когда вы строите и запускаете программу в Xcode, отладчик *присоединяется* к программе, чтобы вы могли с ним работать.

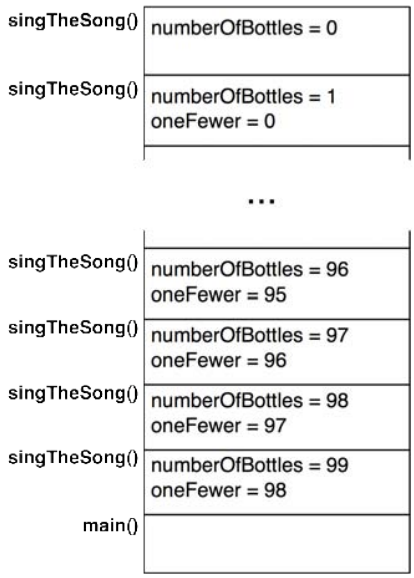


Рис 5.5. Кадры вызовов рекурсивной функции в стеке

Просмотр кадров в отладчике

Вы можете воспользоваться отладчиком для просмотра кадров в стеке, однако для этого придется прервать выполнение вашей программы на середине. В противном случае функция `main()` отработает до конца, и никаких кадров не останется. Чтобы увидеть как можно больше кадров в программе `BeerSong`, следует прервать выполнение в строке, которая выводит сообщение «There are simply no more bottles of beer on the wall».

Как это сделать? Найдите в файле *main.c* строку

```
printf ("There are simply no more bottles of beer on the wall,\n");
```

Слева от кода находятся два слегка затененных столбца. Щелкните на более широком, левом столбце рядом с приведенной строкой.

Синий индикатор показывает, что вы установили *точку прерывания*. Так называется позиция в программном коде, в котором отладчик должен приостановить выполнение вашей программы. Снова запустите программу. Она запускается, а потом останавливается прямо перед строкой, в которой была установлена точка прерывания.

Сейчас ваша программа «застыла на месте», и вы можете изучить ее более внимательно. В области навигатора щелкните на значке ≡, открывающем *навигатор отладки*. В этом навигаторе отображаются все кадры, находящиеся в стеке в данный момент (эта информация также называется *трассировкой стека*).

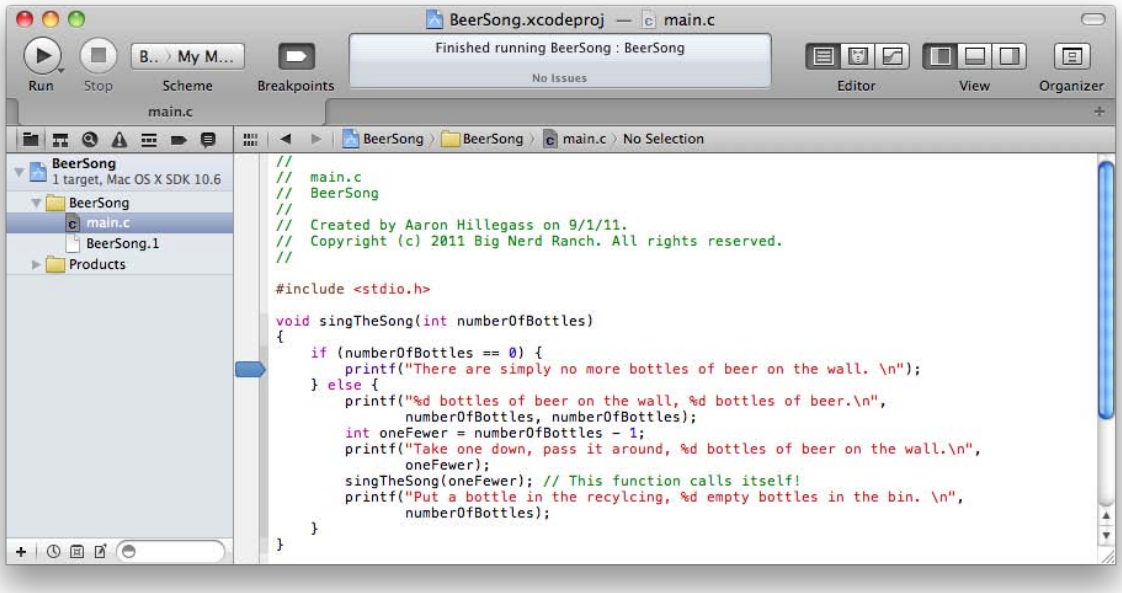


Рис. 5.6. Установка точки прерывания

В трассировке стека кадры идентифицируются по именам своих функций. С учетом того, что наша программа состоит почти исключительно из рекурсивной функции, в кадрах будут указаны одинаковые имена, а различать их придется по передаваемому значению `oneFewer`. В самом конце стека, конечно, расположен кадр функции `main()`.

Выбрав кадр, можно просмотреть переменные в этом кадре и исходный код текущей выполняемой строки кода, Выберите кадр первого вызова `singTheSong`.

В левой нижней части окна находится список переменных выбранного кадра с их текущими значениями. Справа в области, называемой консолью, отображается вывод программы. (Если консоль не видна, найдите кнопки у правого края экрана. Щелкните на средней кнопке, чтобы открыть консоль). На консоли виден эффект установки точки прерывания: программа остановилась до достижения последней строки, завершающей «песню».

Теперь точку прерывания нужно убрать, чтобы программа продолжила нормальное выполнение. Перетащите мышью синий индикатор с левого поля, или щелкните на значке в верхней части панели навигатора, чтобы открыть *навигатор точек прерывания* и просмотреть весь список точек прерывания в проекте. Найдите в этом списке свою точку прерывания и удалите ее.

Чтобы продолжить выполнение программы, щелкните на кнопке, на панели отладчика, между редактором и списком переменных.

Мы всего лишь рассмотрели самые общие возможности отладчика, чтобы вы поняли, как работают кадры. Использование отладчика для назначения точек прерывания и просмотра кадров в стеке программы будет особенно полезным тогда, когда ваша программа работает не так, как ожидалось, и вы хотите разобраться, что же в ней происходит.

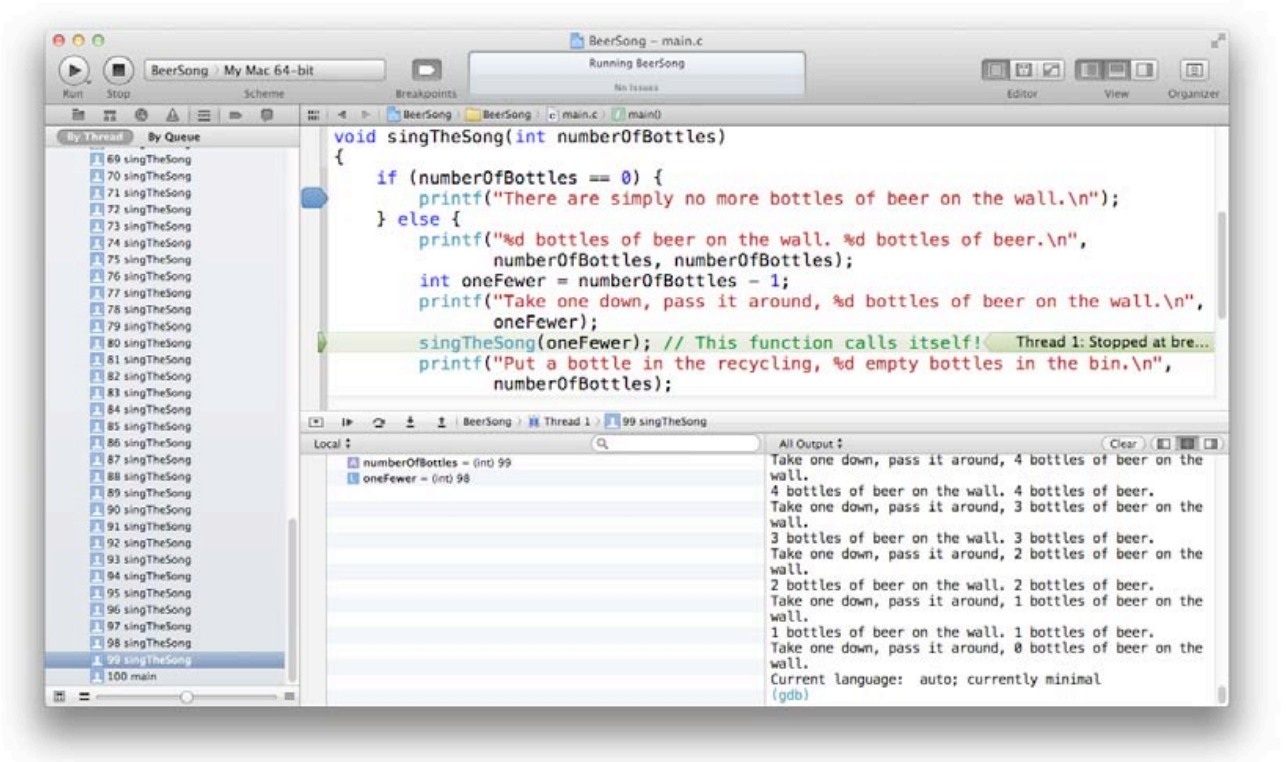


Рис. 5.7. кадры рекурсивной функции в стеке

return

Многие функции в конце своей работы возвращают некоторое значение. Тип данных, возвращаемых функцией, указывается перед ее именем. (Если функция не возвращает никакого значения, вместо типа указывается ключевое слово `void`).

Создайте новую программу командной строки C с именем `Degrees`. В файле *main.c* добавьте перед `main()` функцию, которая преобразует температуру по Цельсию в шкалу Фаренгейта. Включите вызов новой функции в `main()`.

```
#include <stdio.h>
float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}
int main(int argc, const char * argv[])
{
float freezeInC = 0;
float freezeInF = fahrenheitFromCelsius(freezeInC);
printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
return 0;
}
```

Заметили, как мы берем возвращаемое значение `fahrenheitFromCelsius()` и присваиваем его переменной `freezeInF` типа `float`? Здорово, верно?

Выполнение функции прекращается при возврате управления командой return. Например, возьмем следующую функцию:

```
float average(float a, float b)
{
    return (a + b)/2.0;
    printf("The mean justifies the end\n");
}
```

Если вызвать эту функцию, вызов `printf()` выполняться никогда не будет.

Возникает естественный вопрос: «Почему мы всегда возвращаем 0 из функции **main**()»? Возвращая системе 0, вы тем самым говорите: «Все прошло нормально». Если программа завершается из-за каких-то неполадок, возвращается значение 1.

На первый взгляд это противоречит смыслу 0 и 1 в командах `if`; значение 1 соответствует `true`, а значение 0-`false`, поэтому вроде бы естественно рассматривать 1 как признак успеха, а 0 - как признак неудачи. Поэтому считайте, что `main()` возвращает количество ошибок. В этом случае 0 - лучший результат! Ни одной ошибки - программа выполнена успешно.

Чтобы смысл возвращаемого значения стал более наглядным, некоторые программисты используют константы `EXIT_SUCCESS` **и** `EXIT_FAILURE`, которые представляют собой обычные псевдонимы для значений 0 и 1 соответственно. Эти константы определяются в заголовочном файле *stdlib.h:*

```
#include <stdio.h>
#include <stdlib.h>
float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}
int main(int argc, const char * argv[])
{
float freezeInC = 0;
float freezeInF = fahrenheitFromCelsius(freezeInC);
printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
return EXIT_SUCCESS;
}
```

В этой книге мы будем использовать 0 вместо **EXIT_SUCCESS**.

Глобальные и статические переменные

В этой главе упоминались локальные переменные, существующие только во время выполнения функции. Также существуют переменные, доступные в любой функции и в любое время. Они называются *глобальными переменными*. Чтобы переменная стала глобальной, объявите ее за пределами какой-либо функции. Например, можно

добавить переменную `lastTemperature` для хранения преобразованной температуры по Цельсию. Включите в программу объявление глобальной переменной:

```
#include <stdio.h>
#include <stdlib.h>
// Обьявление глобальной переменной float lastTemperature;
float fahrenheitFromCelsius(float cel)

{
    lastTemperature = cel;
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}

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

{
float freezeInC = 0;
float freezeInF = fahrenheitFromCelsius(freezeInC);
printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
printf("The last temperature converted was %f\n", lastTemperature);
return EXIT_SUCCESS;
}
```

Любая нетривиальная программа состоит из десятков файлов, содержащих разные функции. Глобальные переменные доступны в коде каждого из этих файлов. Иногда совместное использование переменной разными файлами - именно то, что нужно. С другой стороны, доступ к переменной из множества функций способен породить величайшую путаницу. Для решения этой проблемы существуют *статические переменные*. Статическая переменная, как и глобальная переменная, объявляется за пределами какой-либо функции - но при этом она доступна только в коде того файла, в котором была объявлена. Таким образом, вы получаете все преимущества переменной не локальной, существующей вне каких-либо функций, но при этом избегаете проблем типа «Кто трогал мою переменную?»

Нашу глобальную переменную можно легко преобразовать в статическую, но тик как наша программа состоит из единственного файла *main*.c, такая замена ни на что не повлияет.

```
// Объявление статической переменной static float lastTemperature;
```

И статическим, и глобальным переменным можно присвоить значение при создании:

```
// Переменная lastTemperature инициализируется 50 градусами static float
lastTemperature = 50.0;
```

Если переменной не было присвоено начальное значение, она автоматически инициализируется нулем.

В этой главе вы познакомились с функциями. В части III,когда мы займемся Objective-C, вам также встретится термин *метод* - методы имеют очень, очень много общего с функциями.

Упражнение

Сумма внутренних углов треугольника должна быть равна 180 градусам. Создайте новую программу командной строки C с именем `Triangle`. Включите в файл *main.c* функцию, которая получает первые два угла и возвращает величину третьего. Вызов функции должен выглядеть так:

```
#include <stdio.h>
// Add your new function here
int main(int argc, const char * argv[])
{
    float angleA = 30.0;
    float angleB = 60.0;
    float angleC = remainingAngle(angleA, angleB);
    printf("The third angle is %.2f\n", angleC);
    return 0;
}
```

В приведенном примере программа должна выдавать следующий результат: The third angle is 90.00

6. Числа

Мы уже использовали числа для измерения и вывода температуры, веса и продолжительности приготовления индейки. Пора поближе разобраться с тем, как работают числа в языке C. На компьютере числа делятся на два вида: целые и вещественные (с плавающей запятой). Мы уже использовали и те и другие. В этой главе я постараюсь принести в систему все то, что программист C должен знать о числах.

printf()

Но прежде чем браться за числа, мы рассмотрим функцию `printf()`, которая тоже уже использовалась в наших программах. Функция `printf()` выводит на консоль строку, то есть цепочку символов. По сути, строка содержит некоторый текст.
Откройте проект `ClassCertificates` и найдите в файле *main.c* функцию `congratulateStudent()`.

```
void congratulateStudent(char *student, char *course, int numDays)
{
    printf("%s has done as much %s Programming as I could fit into %d days.\n",
        student, course, numDays);
}
```

Что делает этот вызов `printf()`? Вообще-то вы уже видели результат и знаете, что она делает. Давайте разберемся, как она это делает.

Функция `printf()` получает аргумент - строку. Чтобы определить *строковый литерал* (в отличие от строки, хранящейся в переменной), следует заключить текст в кавычки.

Строка, передаваемая `printf()`, называется *форматной строкой*. Форматная строка может содержать *заполнители*. В нашем примере строка содержит три заполнителя: `%s`,`%s` и `%d`. При выполнении программы заполнители заменяются значениями переменных, следующих за строкой, - в нашем примере это переменные `student`, `course` и `numDays`. Обратите внимание: переменные подставляются на место заполнителей в указанном порядке. Если поменять местами `student` и `course` в списке переменных. то сообщение будет выглядеть так:

```
Cocoa has done as much Mark Programming as I could fit into 5 days.
```

Тем не менее заполнители и переменные нельзя заменять по своему усмотрению. Заполнитель `%s` означает, что на его место должна подставляться строка,

а заполнитель %d предназначен для целого числа. (Попробуйте поменять их местами и посмотрите, что про изойдет).
Обратите внимание: переменные student и course объявлены с типом char*. Пока считайте, что char* - это обычная строка. Мы еще вернемся к строкам Objective-C в главе 14, а к типу char* - в главе 34.

Наконец, что это за \n? В командах printf() символ перевода строки необходимо явно включать в выводимые данные, иначе все данные будут выводиться в одну строку. Комбинация \n обозначает символ перевода строки (или символ новой строки).

А теперь вернемся к числам.

Целые числа

Целое число не имеет дробной части. Целые числа хорошо подходят для не- которых задач - скажем, для подсчета. Некоторые задачи (скажем, подсчет всего населения планеты) требует очень больших чисел. Для другой задачи (например, подсчет детей в классе) хватит существенно меньших чисел.

Для разных задач были созданы целочисленные переменные с разными размерами. Целочисленная переменная состоит из некоторого количества разрядов (битов), используемых для кодирования числа; чем больше разрядов в переменной, тем больше разных чисел в ней можно сохранить. Типичные размеры переменных - 8, 16, 32 и 64 разрядные.

Кроме того, для одних задач нужны отрицательные числа, а для других нет. Соответственно, целочисленные типы делятся на *знаковые* и *беззнаковые*. 8 - разрядная беззнаковая переменная может хранить любое число от 0 до 255. Откуда берется это значение? 2 = 256 возможных чисел, а мы начинаем отсчет с 0.

В 64-разрядной знаковой переменной может храниться любое целое число из диапазона от -9 223 372 036854775808 до 9223 372 036 854 775 807. Один разряд, отведенный под знак, исключается. Остается 2 = 9 223 372 036 854 775 808.

При объявлении целого числа можно очень точно указать, что именно вам нужно:

```
UInt32 x;  // 32-разрядное целое без знака
SInt16 y; // 16-разрядное целое со знаком
```

Впрочем, программисты чаще используют более общие типы, о которых вы узнали в главе 3.

```
char a; // 8-разрядное значение
short b; // Обычно 16-разрядное (в зависимости от платформы)
int c; // Обычно 32-разрядное (в зависимости от платформы)
long d; // 32- или 64-разрядное (в зависимости от платформы)
long long e; // 64-разрядное значение
```

Почему тип char отнесен к числовым? Любой символ может быть представлен в виде 8-разрядного числа, а компьютеры предпочитают работать с числами. Типы

char, short, int, long **и** long long по умолчанию имеют знак, но вы можете указать ключевое слово unsigned для создания беззнакового эквивалента.

Кроме того, размеры целых чисел зависят от платформы. (Платформой называется сочетание операционной системы и конкретного компьютера или мобильного устройства.) Одни платформы являются 32-разрядными, другие 64-разрядными. Они отличаются прежде всего размером адреса памяти (мы поговорим об этом подробнее в главе 8).

Фирма Apple создала два целых типа, которые являются 32-разрядными на 32-разрядных платформах и 64-разрядными на 64-разрядных платформах:

```
NSInteger g;
NSUInteger h;
```

Эти типы используются во многих приложениях Apple. Они во всех отношениях эквивалентны long **и** unsigned long.

Заполнители для вывода целых чисел

Создайте новый проект командной строки C с именем Numbers. В файле *main.c* создайте целое число и выведите его по основанию 10 (то есть в десятичном представлении) при помощи функции printf():

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
int x = 255;
printf("x is %d.\n", x); return 0;
}
```

Результат должен выглядеть так:

```
x is 255.
```

Как вы видели, заполнитель %d выводит целое число в десятичном представлении. Как работают другие заполнители? Целое число также можно вывести в восьмеричном или шестнадцатеричном представлении. Добавьте в программу пару строк:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
int x = 255;
printf("x is %d.\n", x);
printf("In octal, x is %o.\n", x);
printf("In hexadecimal, x is %x.\n", x);
return 0;
}
```

Запустите ее. Результат будет выглядеть так:

```
x is 255.
In octal, x is 377.
In hexadecimal, x is ff.
```

(Мы вернемся к шестнадцатеричным данным в главе 33.)

А если выводится длинное целое число (то есть содержащее больше разрядов, чем обычное)? В этом случае между `%` и символом формата следует вставить `l` (обозначение `long`) или `ll` (обозначение `long long`). Измените программу так, чтобы вместо `int` в ней использовался тип `long`:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
long x = 255;
printf("x is %ld.\n", x);
printf("In octal, x is %lo.\n", x);
printf("In hexadecimal, x is %lx.\n", x);
return 0;
}
```

Для вывода целого десятичного числа без знака используется заполнитель `%u`:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
unsigned long x = 255; printf("x is %lu.\n", x);
      // восьмеричный и шестнадцатеричный формат уже подразумевает, что число
выводится без знака
    printf("In octal, x is %lo.\n", x);
    printf("In hexadecimal, x is %lx.\n", x);

return 0;
}
```

Операции с целыми числами

Арифметические операторы +, - и * работают так, как и следовало ожидать. Кроме того, они обладают естественными правилами приоритета математических операций: * вычисляется до + или -. Замените в *main.c* предыдущий код вычислением выражения:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
return 0;
}
```

Вы получите следующий результат:

```
3 * 3 + 5 * 2 = 19
```

Целочисленное деление

Многие начинающие программисты C удивляются тому, как работает целочисленное деление. Давайте попробуем:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d\n", 11 / 3);
return 0;
}
```

Должно получиться 11 / 3 - 3.666667, верно? А вот и нет. Результат деления 11 / 3 равен 3. При делении одного целого числа на другое всегда получается целое число, Система округляет результат отсечением дробной части (таким образом, при делении -11 / 3 получится -3). И это выглядит вполне разумно, если рассматривать операцию так: «11 делим на 3 - получается 3 с остатком 2». Однако остаток от деления достаточно часто тоже важен. Оператор `%` похож на `/`, но вместо частного он возвращает остаток от деления:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d remainder of %d \n", 11 / 3, 11 % 3);
return 0;
}
```

А если вы *все равно* хотите получить 3.666667 вместо 3? Преобразуйте `int` во `float` оператором преобразования типа. Нужный тип указывается в круглых скобках слева от переменной, которую нужно преобразовать. Преобразуйте делитель к типу `float` перед выполнением операции:

```
int main (int argc, const char * argv[])
{
printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
printf("11 / 3 = %d remainder of %d \n", 11 / 3, 11 % 3);
printf("11 / 3.0 = %f\n", 11 / (float)3);
return 0;
}
```

Теперь вместо целочисленного деления будет выполняться деление вещественное, и вы получите 3.666667. Запомните правило: оператор / выполняет целочисленное деление только в том случае, если и делимое и делитель относятся к целому типу. Если хотя бы одно из двух чисел является вещественным, то вместо целочисленного деления выполняется вещественное.

Сокращенная запись

Все операторы, рассмотренные нами до настоящего момента, давали новый результат. Например, чтобы увеличить переменную x на 1, вы используете оператор + и заносите результат обратно в x:

```
int x = 5;
x = x + 1; // Значение x теперь равно 6
```

Программисты C выполняют подобные операции так часто, что для них были созданы специальные операторы, изменяющие значение переменной без лишнего присваивания. Например, для увеличения значения x на 1 можно использовать *оператор инкремента* (++):

```
int x = 5;
x++; //Значение x теперь равно 6
```

Также существует *оператор декремента* (--), уменьшающий значение на 1:

```
int x = 5;
x--; //Значение x теперь равно 4
```

А если потребуется увеличить x не на 1, а на 5? Конечно, можно использовать сложение с последующим присваиванием:

```
int x = 5;
x= x+ 5; // Значение x равно 10
```

Но и для этой операции тоже существует сокращенная запись:

```
int x = 5;
x += 5; // x is 10
```

Вторая строка читается как «присвоить x значение x+5». Наряду с оператором += также существуют операторы -=,*=, /= и %=.
Для вычисления модуля (абсолютной величины) целого числа вместо оператора используется функция `abs()`. Если потребуется вычислить модуль значения типа `long`, используйте функцию `labs()`. Обе функции объявляются в файле *stdlib.h*:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main (int argc, const char * argv[])
{
printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
printf("11 / 3 = %d remainder of %d \n", 11 / 3, 11 % 3);
printf("11 / 3.0 = %f\n", 11 / (float)3);
printf("The absolute value of -5 is %d\n", abs(-5));
return 0;
}
```

Вещественные числа

Для хранения числа с дробной частью (например, 3.2) используются вещественные типы (также называемые типами с плавающей запятой). Обычно программисты рассматривают вещественные числа как мантиссу, умноженную на 10 в степени целочисленной экспоненты. Например, число 345.32 рассматривается как 3.4532 x 10. Собственно, именно так эти числа хранятся: в 32-разрядном вещественном числе 8 разрядов выделяются для хранения экспоненты (целое со знаком), а 23 разряда выделяются для хранения мантиссы; еще один разряд используется для хранения знака.

Вещественные числа, как и целые, существуют в нескольких разновидностях. В отличие от целых, вещественные числа всегда имеют знак

```
float g; // 32-разрядное число
double h; // 64-разрядное число
long double i; // 128-разрядное число
```

Заполнители для вывода вещественных чисел

Функция `printf()` также может выводить вещественные значения, для представления которых чаше всего используются заполнители **`%f` и `%e`**. Включите операции вывода вещественных чисел в файл *main.c:*

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

{
    double y = 12345.6789;
    printf("y is %f\n", y);
    printf("y is %e\n", y);
return 0;
}
```

Постройте и запустите программу. Она выводит следующий результат:

```
y is 12345.678900
y is 1.234568e+04
```

Таким образом, заполнитель `%f` использует при выводе обычную десятичную запись, а заполнитель `%e` - экспоненциальную запись.

Обратите внимание: `%f` в настоящее время выводит 6 цифр в дробной части. Часто этого оказывается слишком много. Чтобы ограничить вывод двумя цифрами, слегка измените заполнитель:

```
int main (int argc, const char *argv[])
{
      double y = 12345.6789;
      printf("y is %.2f\n", y);
      printf("y is %.2e\n", y);
      return 0;
}
```

Результат выглядит так:

```
y is 12345.68
y is 1.23e+04
```

Функции для работы с вещественными числами

Операторы +, -, * и / работают так, как и следовало ожидать; ничего особенного о них сказать нельзя. Если ваша программа выполняет много вещественных вычислений, вам пригодится математическая библиотека. Чтобы ознакомиться с ее возможностями, откройте на Mac окно терминала и введите команду `man math`. Вы получите отличную сводку всех возможностей математической библиотеки: тригонометрия, возведение в степень, квадратные и кубические корни и.т.д.

Если вы захотите использовать какие-либо математические функции в своем коде, не забудьте включить заголовок математической библиотеки в начало файла:

Предупреждение: все тригонометрические функции работают с радианами, а не с градусами! [2]

```
#include <math.h>
```

Упражнение

Используйте математическую библиотеку! Включите в *main.c* код, который выводит синус 1 радиана. Число должно выводиться с округлением до трех цифр В дробной части (подсказка: в результате должно получиться `0.841`).

7. Циклы

Создайте в Xcode новый проект командной строки C с именем `Coolness`. Первая программа. которую я написал в своей жизни, выводила фразу «Аарон крут» (мне тогда было 10 лет). Давайте напишем ее:

```
#include <stdio.h>
int main(int argc, const char * argv[])
{
    printf("Aaron is Cool\n");
return 0;
}
```

Постройте и запустите программу.

Давайте предположим, что для пущей уверенности в собственной крутизне мое 10-летнее «Я» желает, чтобы фраза выводилась не один раз, а целую дюжину. Как это сделать?

Есть тупое решение:

```
#include <stdio.h>
int main(int argc, const char * argv[])
{
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    return 0;
}
```

А есть решение умное - создать цикл.

Цикл while

Наше знакомство с циклами начнется с цикла `while`. Конструкция `while` отдаленно напоминает конструкцию `if`, которая рассматривалась в главе 4: 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`.

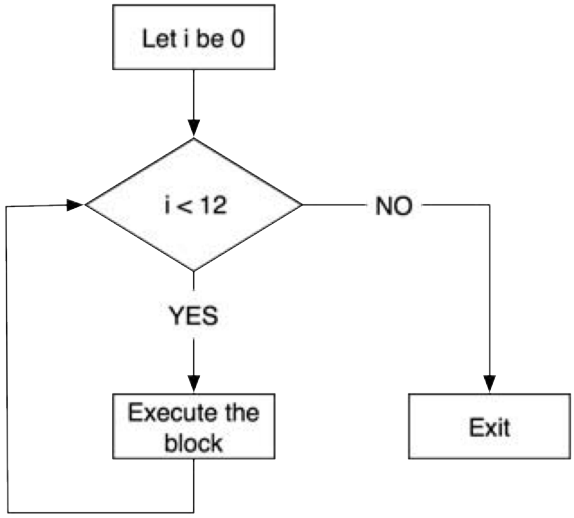


Рис. 7.1. Цикл while

Перепишите функцию `main()` в следующем виде:

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

Постройте и запустите программу

Условие (i < 12) проверяется перед каждым выполнением блока. Как только оно окажется ложным, управление передается следующей команде после блока.
Обратите внимание на увеличение i во второй строке блока. Это очень важный момент: без увеличения переменной цикл будет выполняться бесконечно, потому что выражение всегда будет оставаться истинным. На рис. 7.1 изображена блок-схема этого цикла `while`.

Цикл for

Цикл `while` - обобщенная структура, но программисты C часто используют циклы в своих программах по одной стандартной схеме:

```
инициализация
while (условие) {
      код
```

```
        последний шаг
    }
```

Для нее в языке C была определена специальная сокращенная завись: цикл `for`. В цикле `for` приведенная выше схема записывается так:

```
for (инициализация; условие; последний шаг) {
код;
}
```

Измените программу так, чтобы в ней использовался цикл `for`:

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

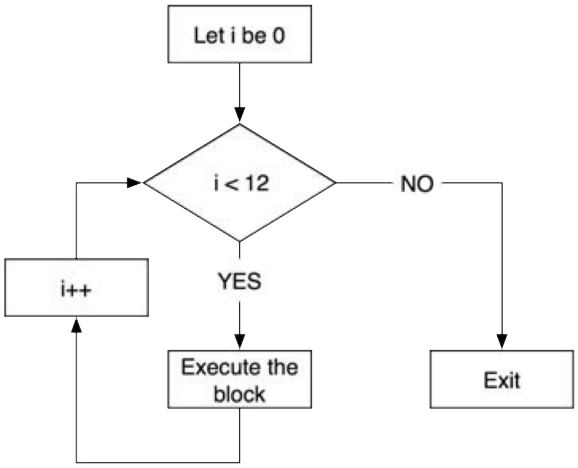


Рис. 7.2. Цикл for

В этом простом примере цикл определяет, сколько раз должна быть выполнена некоторая операция. Однако на практике циклы чаще используются для перебора коллекций элементов - скажем, списка имен, Допустим, я бы мог изменить эту программу так, чтобы цикл вместо повторного вывода одного имени перебирал список имен моих друзей. При каждом проходе цикл сообщал бы о крутизне очередного друга. Тема использования циклов с коллекциями будет рассматриваться, начиная с главы 15.

break

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

при обнаружении нужного числа прерываем выполнение цикла. Внесите изменения в программу:

```
#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;
}
```

Постройте и запустите программу. Результат выглядит так:

```
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.
```

Итак, команда `break` передает управление в конец блока.

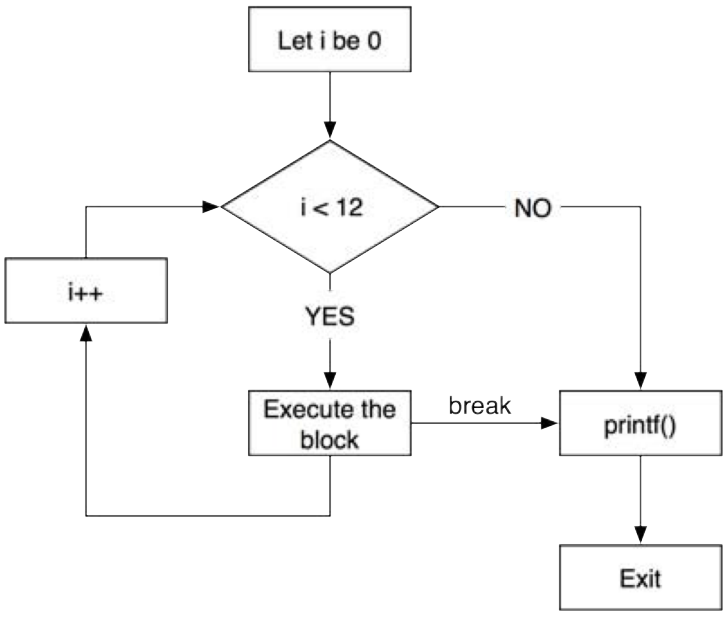


Рис. 7.3, Команда break

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;
}
```

Постройте и запустите программу.

```
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.
```

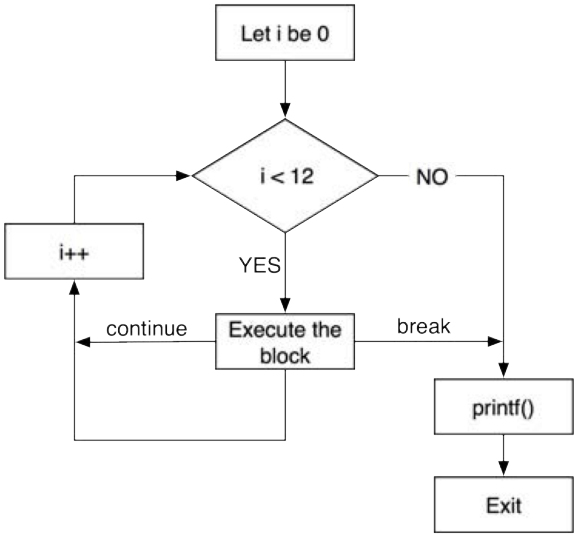


Рис. 7.4. Команда continue

Цикл do-while

Крутые парни не используют 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`:

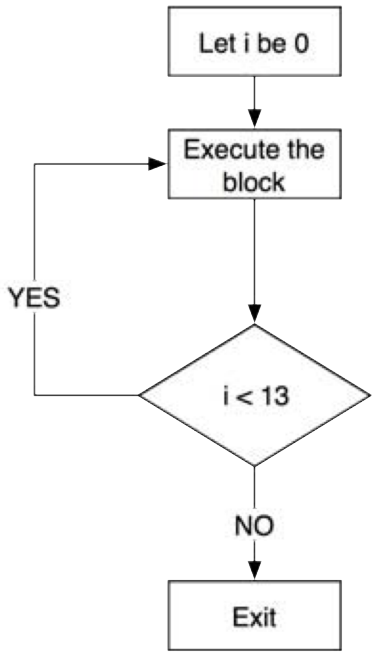


Рис. 7.5. Цикл do-while

Упражнение

Напишите программу, которая считает в обратном направлении от 99 до 0 через 3 и выводит каждое число. Если текущее число кратно 5, то программа также выводит сообщение «`Found one`!» Вывод программы должен выглядеть примерно так:

```
99
96
93
```

```
90
Found one!
87
...
0
Found one!
```

8. Адреса и указатели

В сущности, ваш компьютер состоит из процессора и оперативной памяти - огромного набора переключателей, которые могут включаться и выключаться процессором. Говорят, что 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.

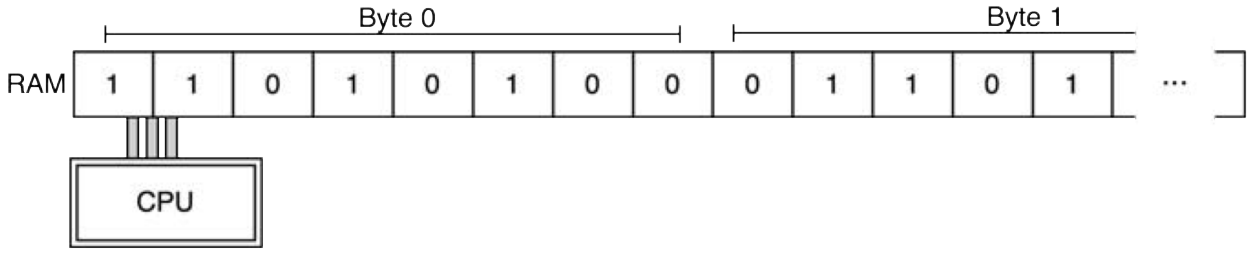


Рис 8.1. Память и процессор

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

Создайте в Xcode новый проект: программу командной строки C с именем `Addresses`.

Адресом переменной называется номер ячейки памяти, в которой хранится значение этой переменной. Для получения адреса переменной используется оператор `&`:

```
#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 выводится 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;
}
```

Постройте и запустите программу.

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 па возможные ошибки R 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;
}
```

Постройте и запустите программу.

Если вы еще не до конца понимаете, как работают указатели, - не огорчайтесь. В этой книге мы проведем довольно много времени за работой с указателями, так что вы еще успеете освоиться с ними.

А теперь совершим типичную ошибку программирования - уберем * из четвертой строки `main()`, чтобы строка приняла вид

```
addressOfI = 89;
```

Xcode выдает предупреждение о несовместимом преобразовании целого числа в указатель присваивании '`int`' вместо '`int` *'.Исправьте ошибку.

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`.

Постройте и 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 пыла 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 нуль. 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.

Хороший стиль объявления указателей

Объявление указателя на float выглядит примерно так:

```
float *powerPtr;
```

Так как переменная объявляется с типом «указателя на float», возникает искушение использовать запись следующего вида:

```
float* powerPtr;
```

Здесь нет ошибки, и компилятор позволит вам использовать такую запись. Однако хороший стиль программирования не рекомендует поступать подобным образом.

Почему? Потому что в одной строке могут содержаться объявления сразу нескольких переменных. Например, если я захочу объявить переменные x, y и z, это можно сделать так

```
float x, y, z;
```

Каждая переменная относится к типу `float`.

Как вы думаете, к какому типу относятся эти переменные?

```
float* b, c;
```

Сюрприз! Переменная b представляет собой указатель на `float`, а переменная c - просто `float`. Если вы хотите, чтобы обе переменные были указателями, поставьте знак * перед именем каждой из них:

```
float *b, *c;
```

Размещение знака * рядом с именем переменной более наглядно показывает, что имел в виду программист.

Упражнения

Напишите программу которая показывает, сколько памяти занимает значение типа `float`.

На Mac тип `short` представляет собой 2-байтовое целое число, в котором один бит задействован для хранения знака. Какое наименьшее число может быть представлено этой переменной? А наибольшее? В переменной типа `unsigned short` могут храниться только не отрицательные числа. Какое наибольшее число может быть представлено этим типом?

9. Передача по ссылке

В языке C есть стандартная функция `modf()`, которая получает значение `double` и возвращает целую и дробную части числа. Например, у числа 3,14 целая часть равна 3, а дробная - 0,14.

Вызывая функцию `modf()`, вы хотите получить обе части числа. Однако функция C может возвращать только одно значение. Как функция `modf()` вернет два числа вместо одного?

При вызове `modf()` следует указать адрес, по которому будет сохранено одно из чисел. А если говорить конкретно, функция возвращает дробную часть числа и копирует целую часть по указанному вами адресу. Создайте новый проект - программу командной строки C с именем PBR.

Отредактируйте содержимое *main.c*:

```
#include <stdio.h>
#include <math.h>
int main(int argc, const char * argv[])
{
    double pi = 3.14;
    double integerPart;
    double fractionPart;

    // Адрес integerPart передается в аргументе
    fractionPart = modf(pi, &integerPart);
    // Определение значения, хранящегося в integerPart
          printf("integerPart = %.0f, fractionPart = %.2f\n", integerPart,
fractionPart);
return 0;
}
```

Такой механизм передачи данных называется *передачей по ссылке*. Иначе говоря, вы передаете адрес («ссылку»), а функция записывает по нему свои данные.

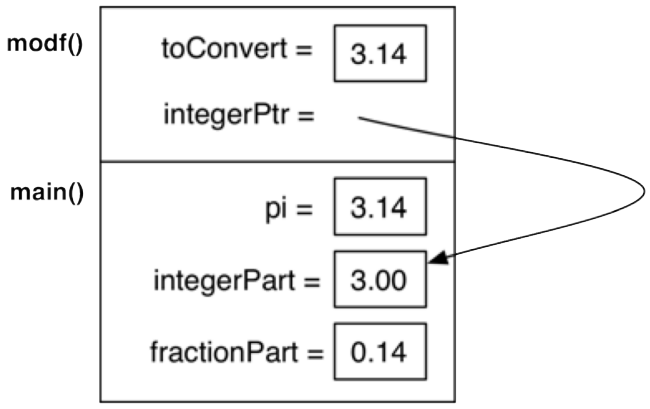


Рис. 9,1. Состояние стека при возврате из modf()

Представьте. что вы выдаете поручения шпионам. Вы говорите: «Мне нужны фотографии министра финансов с его новой подружкой. У статуи ангела в парке находится тайник - короткая железная труба. Когда фотографии будут готовы, сверните и оставьте в тайнике. Я заберу их во вторник
после обеда».

Функция `modf()` тоже работает по принципу тайника. Вы говорите ей, что нужно сделать и где оставить результаты, чтобы вы могли позднее забрать их, когда вам это будет удобно. Только вместо железной трубы вы передаете адрес памяти, по которому функция должна оставить свой результат.

Программирование функций с передачей аргументов по ссылке

Положение точки на плоскости обычно определяется в одной из двух систем координат: декартовой или полярной. В декартовых координатах запись (x,y) обозначает смещение вправо на x и вверх на y от начала координат. В полярных координатах запись (`theta`, `radius`) обозначает поворот влево на `theta` радиан со смещением на расстояние `radius` от начала координат.

Допустим, вам потребовалось написать функцию, преобразующую точку из декартовой системы координат в полярную. Такая функция должна получить два вещественных числа и вернуть два вещественных числа.

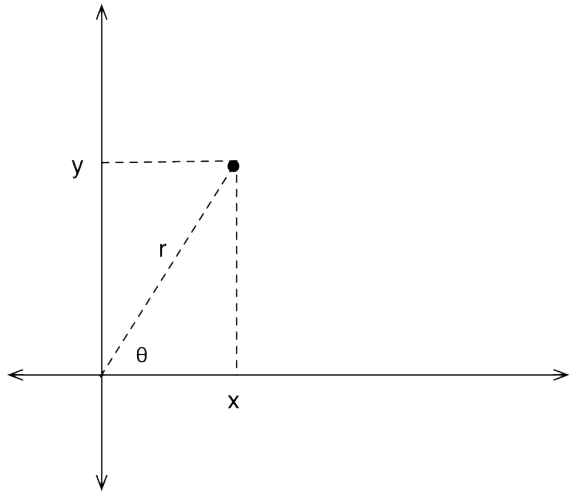


Рис. 9.2, Полярные и декартовы координаты

Объявление функции выглядит примерно так:

```
void cartesianToPolar(float x, float y, float *rPtr, float *thetaPtr)
```

Итак, при вызове функции передаются значения `x` и `y`. Кроме того, функция получает адреса, по которым должны быть сохранены вычисленные значения

radius и theta.

Включите функцию преобразования в начало файла *main*.c и вызовите ее из main():

```
#include <stdio.h>
#include <math.h>

void cartesianToPolar(float x, float y, double *rPtr, double *thetaptr)
{
      // радиус сохраняется по переданному адресу
      *rPtr =sqrt(x *x+y*y);

// вычисление theta;
float theta;
if (x ==0.0) {
      if (y ==0.0) {
            theta = 0.0; // формально неопределенное значение
      }else if (y>0){
            theta =MPI 2;
      }else{
            theta = - M_PI_2;
      }
      }else{
            theta = atan(y/x);
      }
      // значение theta сохраняется по переданному значению
      *thetaPtr = theta;
}

int main(int argc, const char * argv[])
{
double pi = 3.14;
double integerPart;
double fractionPart;

// адрес integerPart передается в аргументе
fractionPart = modf(pi, &integerPart);

// определение значения, сохраненного integerPart
printf("integerPart = %.0f, fractionPart = %.2f\n", integerPart, fractionPart);

double x = 3.O;
double y =4.0;
double radius;
double angle;

cartesianToPolar(x, y, &radius, &angle);
printf("(%.2f, %.2f) becomes (%.2f radians. %.2f)\n", x, y. radius. angle);

      return 0;
}
```

Постройте и запустите программу.

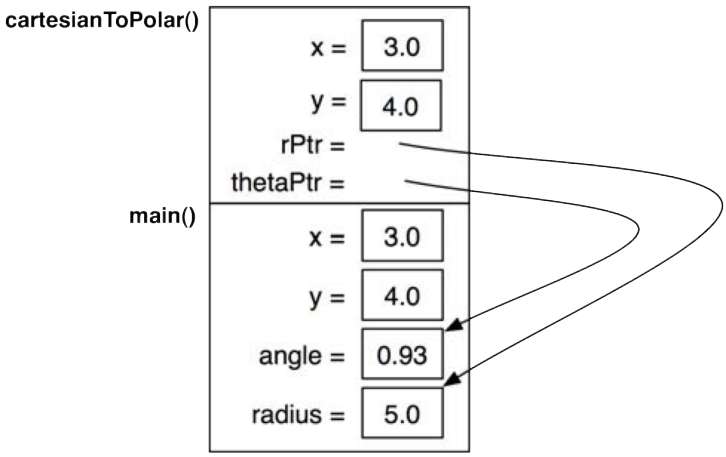


Рис. 9.3. Состояние стека при возврате из `cartesianToPolar()`

Избегайте разыменования NULL

Иногда функция может получать по ссылке сразу несколько значений, тогда как вас интересует только часть из них. Как избежать объявления всех переменных и передачи их адресов, которые все равно не будут использоваться? Обычно вы передаете вместо адреса `NULL`, говоря тем самым функции: «Это конкретное значение мне не понадобится».

Но это означает, что перед разыменованием каждого указателя всегда следует убеждаться в том, что он не равен `NULL`. Включите в функцию `cartesianToPolar()` следующие проверки:

```
void cartesianToPolar(float x, float y,  double *rPtr, double *thetaPtr)
{
    // Указатель rPtr отличен от NULL?
    if (rPtr) {
        // Радиус сохроняется по переданному адресу
        *rPtr = sqrt(x * x + y * y);
    }
    // Указатель thetaPtr равен NULL?
    if (!thetaPtr) {
        // Пропускаем остаток функции
return;
}
// Calculate theta
float theta;
if (x == 0.0) {
    if (y == 0.0) {
        theta = 0.0;
    } else if (y > 0) {
        theta = M_PI_2;
// формально неопределенное значение
    } else {
        theta = - M_PI_2;
}
} else {
...
```

10. Структуры

Нередко возникает необходимость в создании переменной для хранения нескольких взаимосвязанных фрагментов данных. Допустим, вы пишете программу для вычисления индекса массы тела. (Что такое «индекс массы тела», или ИМТ? Это вес в килограммах, разделенный на квадрат роста в метрах. Значение ИМТ меньше 20 означает, что вес тела ниже нормы, а значения выше 30 указывают на избыточный вес. Это крайне неточная метрика для оценки физического состояния, но для примера по программированию она подходит.) Для описания человека нам понадобится значение `float`, содержащее рост в метрах, и значение типа `int`, в котором хранится вес в килограммах.

Для хранения этих данных мы создадим свой собственный тип `Person`. Переменная типа `Person` представляет собой структуру с двумя полями: `heightInMeters` типа `float` и `weightInKilos` тина `int`.

Создайте новый проект: программу командной строки C с именем `BMICalc`. Включите в *main.c* команды создания структуры, содержащей всю необходимую информацию о человеке:

```
#include <stdio.h>
//Обьяснение структуры Person
struct Person {
    float heightInMeters;
    int weightInKilos;
};

int main(int argc, const char * argv[])
{
    struct Person person;
    person.weightInKilos = 96;
    person.heightInMeters = 1.8;
    printf("person weighs %i kilograms\n", person.weightInKilos);
    printf("person is %.2f meters tall\n", person.heightInMeters);
    return 0;
}
```

Обратите внимание на синтаксис обращения к полям структуры - имя структуры отделяется от имени поля точкой.

На рис. 10.1 показано, как выглядит кадр стека `main()` после присваивания значений полям структуры.

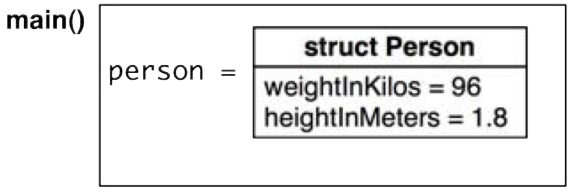


Рис. 10.1. Кадр после присваивания

Как правило, объявление структуры используется в программе многократно, поэтому для типа структур обычно создается `typedef` - псевдоним для объявления типа, позволяющий использовать его как обычный тип данных. Внесите изменения в *main.c*, чтобы в программе создавалось и использовалось определение `typedef` для структуры `Person`:

```
#include <stdio.h>

// объявление типа Person
typedef struct {
      float heightInMeters;
      int weightInKilos;
} Person;

int main(int argc, const {char * argv[])
{
      Person person;
      person.weightInKilos = 96;
      person.heightInMeters = 1.8;
      printf("person weighs %i kilograms\n", person.weightlnKilos);
      printf("person is %.2f meters tall\n", person.heightlnMeters);
      return 0;
}
```

После того как в программе появится определение `typedef`, структуру `Person` можно будет передать другой функции. Включите в программу функцию с именем `bodyMassIndex()`, которая получает структуру `Person` в аргументе и вычисляет ИМТ. Включите вызов этой функции в `main()`:

```
#include <stdio.h>

// объявление типа Person
typedef struct {
      float heightlnMeters;
      int weightlnKilos;
} Person;

float bodyMassIndex(Person p)
{
return p.weightInKilos/(p.heightInMeters * p.heightInMeters);
}

int main(int argc, constchar * argv[])
{
      Person person;
      person.weightlnKilos = 96;
      person.heightlnMeters = 1.8;
      float bmi = bodyMassIndex(person);
      printf("person has a BMI of %.2f\n", bmi);
      return 0;
}
```

Упражнение

Первой структурой, с которой я столкнулся в программировании, была структура `tm`, используемая стандартной библиотекой C для хранения времени с разбивкой на компоненты. Структура определяется следующим образом:

```
struct tm{
      int    tm_sec;    // секунды [0-60]
      int    tm_min;    // минуты [0-59]
      int    tm_hour;   // часы [0-23]
      int    tm_mday;   // день месяца [1-31]
      int    tm_mon;    // месяц [0-11]
      int    tm_year;   // год от 1900
      int    tm_wday;   // день недели с воскресенья [0-6]
      int    tm_yday;   // день года с 1 января [0-365]
      int    tm_isdst;  // флаг летнего времени
      long   tm_gmtoff;  /* смещение в секундах от CUT */
      char   *tm_zone;   /* сокращение часового пояса */
};
```

Функция `time()` возвращает количество секунд от начала 1970 года по Гринвичу. Функция `localtime_r()` читает эту величину и заполняет структуру `tm` соответствующими значениями (точнее, при вызове ей передается адрес количества секунд с 1970 года и адрес структуры `tm`). Таким образом, получение текущего времени в виде структуры `tm` происходит так:

```
long secondsSince1970 = time(NULL);
    printf("It has been %ld seconds since 1970\n", secondsSince1970);
    struct tm now;
    localtime_r(&secondsSince1970, &now);
    printf("The time is %d:%d:%d\n", now.tm_hour, now.tm_min, now.tm_sec);
```

Напишите программу, которая определяет, какая дата (В формате `ММ-ДД-ГГГГ`) соответствует 4 миллионам секунд.
(Подсказка: `tm_mon = 0` соответствует январю, поэтому не забудьте прибавить 1. Также включите заголовочный файл `<time. h>` в начало программы.)

11. Куча

До сих пор в своих программах мы использовали только память, выделяемую в кадрах стека. Эта память автоматически выделяется в начале функции и автоматически освобождается в конце. (Из-за этого удобного поведения локальные переменные часто называют *автоматическими переменными*.)

Однако в некоторых ситуациях требуется зарезервировать длинную последовательность памяти для использования в нескольких функциях. Например, одна функция может прочитать текстовый файл в память, а после нее будет вызвана другая функция, которая подсчитает все гласные буквы в памяти, Как правило, после завершения работы с текстом следует сообщить программе. что занимаемую память можно использовать для других целей.

Длинную последовательность байтов программисты часто называют *буфером*. (Кстати, отсюда и происходит термин «буферизация», который вы видите, пока дожидаетесь загрузки достаточного количества байтов для запуска очередного видеоролика с котом с Youtube.)

Для выделения буфера в памяти используется функция `malloc()`. Буфер выделяется из области памяти, называемой *кучей* (heap), которая существует отдельно от стека. Завершив работу с буфером, следует вызвать функцию `free()` для освобождения памяти и ее возвращения в кучу. Допустим, нам понадобился блок памяти, достаточный для хранения 1000 чисел типа `float`.

```
#include <stdio.h>
#include <stdlib.h> // malloc and free are in stdlib
int main(int argc, const char * argv[])
{
    // объявление указателя
    float *startOfBuffer;

    //  запрос на выделение байтов из кучи
    startOfBuffer = malloc(1000 * sizeof(float));

    // ...работа с буфером...

    // освобождение памяти для повторного использования
    free(startOfBuffer);

    // забываем, с какого адреса начинался выделенный блок
    startOfBuffer = NULL;
return 0;
}
```

Указатель `startOfBuffer` ссылается на первое вещественное число в буфере.

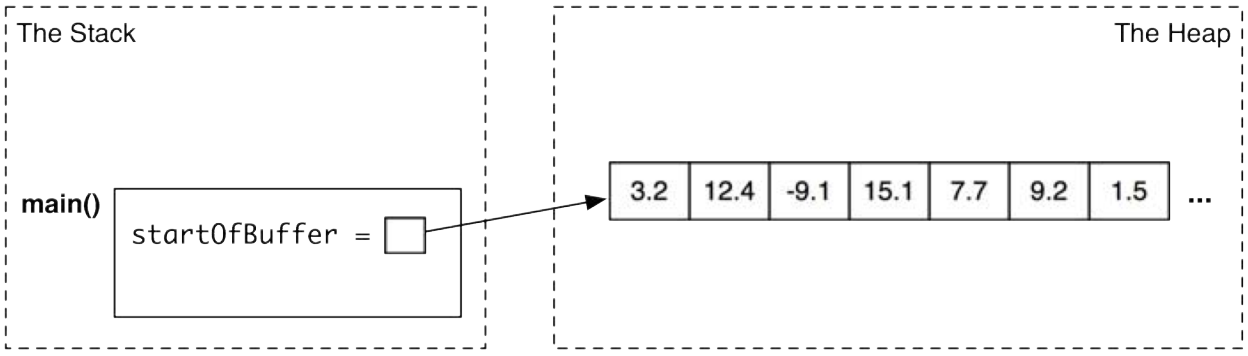


Рис. 11.1. Указатель на буфер, выделенный в куче, хранится в стеке

На этой стадии книги по языку C обычно подолгу объясняют, как происходит чтение и запись данных в различных позициях этого буфера вещественных чисел. Однако эта книга старается как можно быстрее подвести вас к работе с объектами, поэтому массивы C и арифметические операции с указателями откладываются на будущее.

Функция `malloc( )` также может использоваться для выделения памяти под структуру. Например, если вы хотите выделить в куче память для хранения структуры `Person`, программа может выглядеть так:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;
float bodyMassIndex(Person *p)
{
  return p->weightInKilos / (p->heightInMeters * p->heightInMeters);
}
int main(int argc, const char * argv[])
{
    // выделение памяти для одной структуры Person
    Person *x = (Person *)malloc(sizeof(Person));

    // присваивание значений полям структуры
    x->weightInKilos = 81;
    x->heightInMeters = 2.0;

    // вывод ИМТ
    float xBMI = bodyMassIndex(x);
    printf("x has a BMI of = %f\n", xBMI);

    // освобождение памяти
    free(x);

    // забываем начальный адрес
    x = NULL;

return 0;
}
```

Обратите внимание на оператор `->`. Запись `p->weightInKilos` означает: «Разыменовать указатель `p` на структуру и обратиться к полю с именем `weightInKilos`».

Идея хранения структур в куче открывает массу замечательных возможностей. В частности, она закладывает основу для работы с объектами Objective-C. Этой теме будет посвящена следующая часть книги.

III

Objective-C и Foundation

Теперь, когда вы понимаете основные концепции программ. функций, переменных и типов данных, можно переходить к изучению языка Objective-C. Пока мы будем ограничиваться программами командной строки, чтобы не отвлекаться от ключевых аспектов программирования.

Программирование на языке Objective-C неразрывно связано с Foundation - библиотекой классов, используемой для написания программ. Что такое класс? С этого мы и начнем ...

12. Объекты

Во многих языках программирования поддерживается концепция *объектов*. Объекты, как и структуры, содержат некоторые данные. Однако в отличие от структуры, объект также содержит набор функций для работы с этими дан- ными. Чтобы вызвать такую функцию, следует отправить объекту *сообщение*. В объектной терминологии функция, вызываемая при помощи сообщения, называется *методом*.

В начале 1980-х годов Брэд Кокс и Том Лав решили включить объектно-ориентированные возможности в язык C. Они использовали идею выделения памяти в куче и дополнили ее синтаксисом отправки сообщений. Так появился язык Objective-C.

Объекты по своей природе очень «общительны». В ходе своей работы они постоянно принимают и отправляют сообщения, касающиеся всего, что они делают. Сложная программа на языке Objective-C может одновременно хранить в памяти сотни объектов, которые работают и отправляют сообщения друг другу.

Класс описывает конкретную разновидность объектов. В описание включаются методы и *переменные экземпляров*, в которых объекты некоторого типа хранят свои данные. Программист обращается к классу с запросом о создании объекта его типа в куче. Полученный объект называется *экземпляром* этого класса.

Например, на iPhone установлено множество классов, одним из которых является класс `CLLocation`. Вы можете обратиться к классу `CLLocation` с запросом на создание экземпляра `CLLocation`. Объект `CLLocation` содержит несколько переменных для хранения географических данных: широты, долготы и высоты. Объект также содержит набор методов. Например, можно «спросить» у одного экземпляра `CLLocation`, на каком расстоянии он находится от другого объекта `CLLocation`.

Создание и использование объектов

Пришло время написать нашу первую программу на Objective-C. Создайте новый проект командной строки (`Command Line Tool`), но вместо типа `C` выберите тип `Foundation` (рис. 12.1). Присвойте проекту имя `TimeAfterTime`.

Файлы с кодом Objective-C обычно имеют суффикс *.m*. Найдите и откройте файл *main.m*, введите следующие две строки кода:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSDate *now = [NSDate date];
```

```
        NSLog(@"The new date lives at %p", now);
    }

    return 0;

}
```

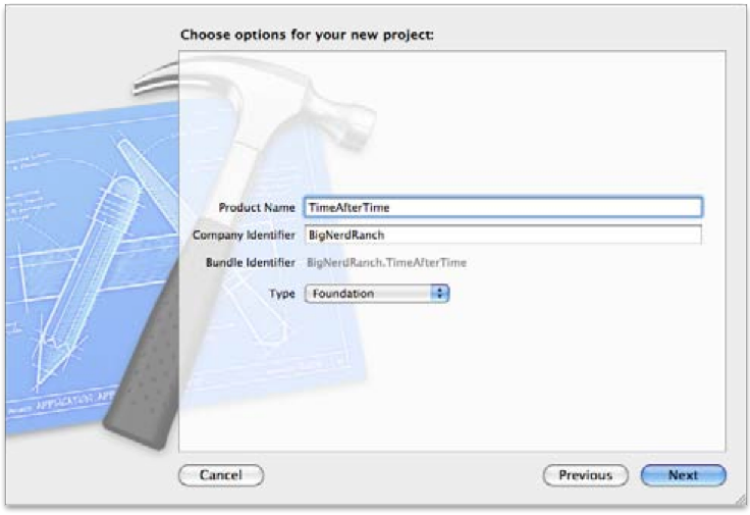


Рис 12.1. Создание программы командной строки `Foundation`

Поздравляю — ваше первое сообщение успешно отправлено! Вы отправили сообщение `date` классу `NSDate`. Метод `date` приказывает классу `NSDate` создать экземпляр `NSDate`, инициализировать его текущей датой/временем и вернуть адрес, с которого размещается в памяти новый объект. Полученный адрес охраняется в переменной `now`. Соответственно переменная является указателем на объект `NSDate`.

`NSLog()` — функция Objective-C, отчасти напоминающая `printf()`; она тоже получает форматную строку, заменяет заполнители `%` фактическими значениями и выводит результат на консоль. Однако ее форматная строка всегда начинается с символа `@` и ее не нужно завершать символом новой строки `\n`.

Постройте и запустите программу. Результат должен выглядеть примерно так:

```
2011-08-05 11:53:54.366 TimeAfterTime[4862:707] The new date lives at
0x100114dc0
```

В отличие от `printf()` функция `NSLog()` выводит перед своими данными дату, время, имя программы и идентификатор процесса.

В дальнейшем, приводя результаты вызова `NSLog()`, я буду опускать эти служебные данные — строки получаются слишком длинными.

В `NSLog()` заполнитель `%p` выводит адрес объекта. Чтобы результат больше напоминал дату, используйте заполнитель `%@`, который приказывает объекту вывести свое описание в виде строки:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
```

```
{
    @autoreleasepool {
        NSDate *now = [NSDate date];
        NSLog(@"The new date lives at %p", now);
      }
      return 0;
}
```

Новый результат будет выглядеть примерно так:

```
The date is 2011-08-05 16:09:14 +0000
```

Анатомия сообщений

Отправка сообщения всегда заключается в квадратные скобки и всегда состоит из минимум двух частей:

- указатель на объект, получающий сообщение;
- имя вызываемого метода.

Отправка сообщения (как и вызов функции) может иметь аргументы. Рассмотрим пример.

Объекты `NSDate` представляют конкретную дату и время. Экземпляр `NSDate` может сообщить разность (в секундах) между датой/временем, которые пред- ставляет он сам, и временем 0:00 (по Гринвичу) 1 января 1970 года. Чтобы получить эту информацию, следует отправить сообщение `timeIntervalSince1970` объекту `NSDate`, на который ссылается указатель `now`.

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

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

    @autoreleasepool {

NSDate *now = [NSDate date];
NSLog(@"The date is %@", now);
double seconds = [now timeIntervalSince1970];
NSLog(@"It has been %f seconds since the start of 1970.", seconds);

      }
      return 0;
}
```

Допустим, вам понадобился новый объект `date` — соответствующий моменту времени на 100 000 секунд позднее уже существующего. Отправив это сообщение

исходному объекту `date`, вы получите новый объект `date`. Метод получает аргумент: величину смещения в секундах. Воспользуемся им для создания нового объекта `date` в функции `main()`:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {

        NSDate *now = [NSDate date];
        NSLog(@"The date is %@", now);

        double seconds = [now timeIntervalSince1970];
        NSLog(@"It has been %f seconds since the start of 1970.", seconds);

        NSDate *later = [now dateByAddingTimeInterval:100000];
        NSLog(@"In 100,000 seconds it will be %@", later);

      }
      return 0;
}
```

В отправляемом сообщении `[now dateByAddingTimeInterval:100000]`:

- `now` — указатель на объект, получающий сообщение («получатель»);
- `dateByAddingTimeInterval` — имя метода («селектор»);
- 100 000 — единственный аргумент.

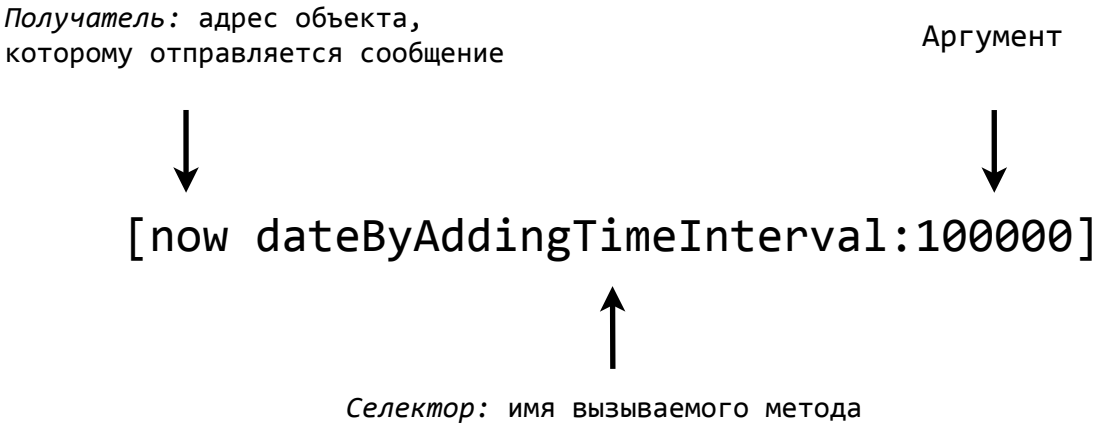


Рис. 12.2. Отправка сообщения

Объекты в памяти

На рис. 12.3 изображена диаграмма объектов. На ней изображены два экземпляра `NSDate` в куче. Две переменные `now` и `later` являются частью кадра функции `main()`. Они указывают на объекты `NSDate` (связи изображены стрелками).

Пока что мы видели только один класс: `NSDate`, но в iOS и Mac OS X входят сотни классов. С самыми распространенными из них мы познакомимся в следующих главах.

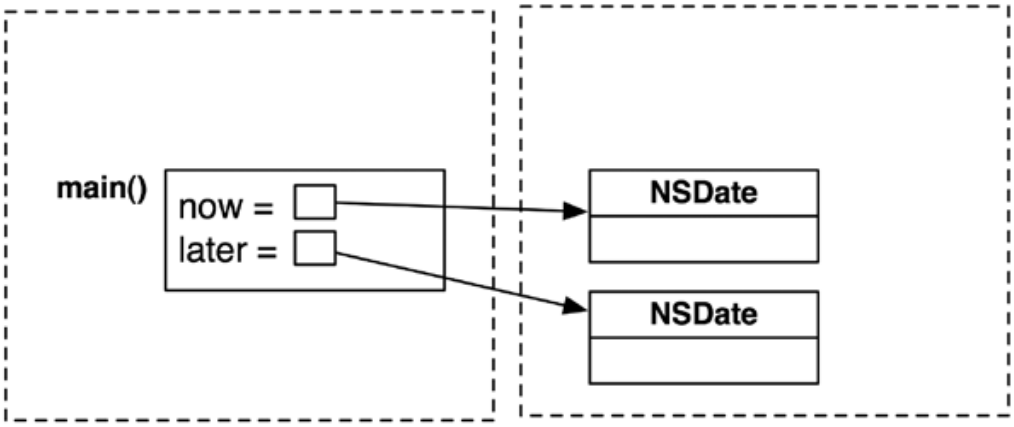


Рис 12.3. Диаграмма объектов приложения `TimeAfterTime`

id

При объявлении указателя на объект обычно указывается класс объекта, на который будет ссылаться этот указатель:

```
NSDate *expiration;
```

Но время от времени бывает нужно создать указатель, не зная точно, на какой объект он будет ссылаться. В таких случаях используется тип `id`, который означает «указатель на какой-либо объект Objective-C». Пример использования `id`:

```
id delegate;
```

Обратите внимание на отсутствие звездочки (*) в этом объявлении — она подразумевается при наличии `id`.

На первых порах довольно трудно усвоить концепции классов, объектов, сообщений и методов. Не огорчайтесь, если вы немного неуверенно чувствуете себя при работе с объектами — ведь это только начало. Мы будем использовать эти конструкции снова и снова, и с каждым разом они будут выглядеть все более осмысленно.

Упражнение

Используйте два экземпляра `NSDate` для вычисления продолжительности вашей жизни в секундах. Подсказка: новый объект `date` по заданному году, месяцу и т. д. создается следующим образом:

```
NSDateComponents *comps = [[NSDateComponents alloc] init];
[comps setYear:1969];
[comps setMonth:4];
[comps setDay:30];
[comps setHour:13];
[comps setMinute:10];
[comps setSecond:0];
NSCalendar *g = [[NSCalendar alloc]
                 initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *dateOfBirth = [g dateFromComponents:comps];
```

Для получения количества секунд между двумя экземплярами `NSDate` следует использовать метод `timeIntervalSinceDate`:

```
double d = [laterDate timeIntervalSinceDate:earlierDate];
```

13. Сообщения

В главе 12 мы отправили несколько сообщений. Еще раз посмотрим на строки, в которых отправлялись эти сообщения с вызовами соответствующих методов:

```
NSDate *now = [NSDate date];
```

Метод `date` является *методам класса*. Иначе говоря, для выполнения этого метода сообщение отправляется самому классу `NSDate`. Метод `date` возвращает указатель на экземпляр `NSDate`.

```
double seconds = [now timeIntervalSince1970];
```

Метод `timeIntervalSince1970` относится к *методам экземпляров*. Для его выполнения сообщение отправляется конкретному экземпляру класса. Метод `timeIntervalSince1970` возвращает значение типа `double`.

```
NSDate *later = [now dateByAddingTimeInterval:100000];
```

Метод `dateByAddingTimeInterval:` также является методом экземпляра. Он получает один аргумент, на что указывает двоеточие после имени метода. Метал также возвращает указатель на экземпляр `NSDate`.

Вложенная отправка сообщений

Метод класса `alloc` возвращает указатель на новый объект, который необходимо инициализировать. Этот указатель используется для отправки новому объекту сообщения `init`. Использование `alloc` и `init` - самый распространенный способ создания объектов в Objective-C. Правила хорошего Стиля программирования рекомендуют отправлять оба сообщения в одной строке кода посредством вложения:

```
[[NSDate alloc] init];
```

Система сначала выполняет внутренние сообщения, а затем переходит
к внешним. Таким образом, сначала классу `NSDate` будет отправлено сообщение `alloc`, а затем результат отправки (указатель на созданный экземпляр)
получит сообщение `init`.

Метод `init` возвращает указатель на новый объект (который почти всегда совпадает с указателем, возвращенным методом `alloc`), что позволяет использовать

возвращенное `init` в команде присваивания. Проверьте, как работают вложенные сообщения - замените строку

```
NSDate *now = [NSDate date];
```

строкой

```
NSDate *now = [[NSDate alloc] init];
```

Множественные аргументы

Некоторые методы получают несколько аргументов. Например, объект `NSDate` не знает, какому дню месяца он соответствует. Для получения этой информации следует использовать объект `NSCa1endar`. Метод `NSCa1endar`, который сообщает день месяца, получает три аргумента. Создайте объект `NSCa1endar` и используйте его в *main.m*:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSDate *now = [[NSDate alloc] init];
        NSLog(@"The date is %@", now);

        double seconds = [now timeIntervalSince1970];
        NSLog(@"It has been %f seconds since the start of 1970.", seconds);

        NSDate *later = [now dateByAddingTimeInterval:100000];
        NSLog(@"In 100,000 seconds it will be %@", later);

        NSCalendar *cal = [NSCalendar currentCalendar];
        NSUInteger day = [cal ordinalityOfUnit:NSDayCalendarUnit
                                        inUnit:NSMonthCalendarUnit
                                       forDate:now];
        NSLog(@"This is day %lu of the month", day);

      }
      return 0;
}
```

Сообщение называется `ordinalityOfUnit:inUnit:forDate:` и получает три аргумента. На это указывают три двоеточия в имени сообщения. Обратите внимание на разбивку отправляемого сообщения по трем строкам. Компилятору это нисколько не мешает - он игнорирует лишние пробелы. Программисты Objective-C обычно выравнивают двоеточия так, чтобы части имени метода можно было легко отличить от аргументов. (И Xcode вам в этом поможет: каждый раз, когда вы начинаете новую строку, предыдущая строка будет снабжаться правильным отступом. Если это не происходит, проверьте настройки отступов в Xcode.)

Первый и второй аргументы метода содержат константы NSDayCalendarUnit и NSMonthCalendarUnit. Эти константы определяются в классе NSCa1endar. Они сообщают методу, что вас интересует день месяца. Третий аргумент определяет дату, к которой относится запрашиваемая информация.

Если бы вместо дня месяца вы захотели узнать час года, то при вызове следовало бы использовать следующие константы:

```
NSUInteger hour = [cal ordinalityOfUnit:NSHourCalendarUnit
                                 inUnit:NSYearCalendarUnit
                                forDate:now];
```

Отправка сообщений nil

Почти во всех объектно-ориентированных языках существует концепция nil - указателя, не ссылающегося ни на какой объект. В Objective-C nil представляет собой нулевой указатель (эквивалент NULL из главы 8).

В большинстве объектно-ориентированных языков отправка сообщений nil недопустима, поэтому перед обращением к объекту следует выполнить проверку на nil. По этой причине в программах часто встречаются конструкции следующего вида:

```
if (fido != nil) {
    [fido goGetTheNewspaper];
}
```

В ходе разработки Objective-C было решено, что отправка сообщения nil будет допустима; просто при этом ничего не будет происходить. Соответственно, следующий код вполне допустим:

```
Dog *fido = nil;
[fido goGetTheNewspaper];
```

Важное замечание #1: если при отправке сообщения ничего не происходит, убедитесь в том, что сообщения не отправляются указателю, которому была задано значение nil.

Важное замечание #2: если вы отправляете сообщение nil, возвращаемое значение не несет полезной информации.

```
Dog *fido = nil;
Newspaper *daily = [fido goGetTheNewspaper];
```

В этом случае значение daily будет равно нулю. (В общем случае, если результатом является число или указатель, при отправке сообщения получателю nil

возвращается нуль. Но для других типов - например, для структур - возможно странные и неожиданные возвращаемые значения.)

Упражнение

На вашем Mac имеется класс с именем `NSTimeZone`. В число его методов входит метод класса `systemTimeZone`, который возвращает часовой пояс компьютера (в виде экземпляра класса `NSTimeZone`), и метод экземпляра `isDaylightSavingTime`, который возвращает YES, если в часовом поясе в настоящее время действует летнее время.

Напишите программу командной строки Foundation, которая проверяет, действует ли в настоящий момент летнее время.

14. NSString

NSString - еще один готовый класс, как и NSDate. В экземплярах NSString хранятся последовательности символов. В программном коде для создания экземпляра NSString можно использовать запись вида

```
NSString *lament = @"Why me!?";
```

Вспомните, по в проекте TimeAfterTime вводился следующий код:

```
NSLog(@"The date is %@", now);
```

NSLog( ) - функция Objective-C (не метод!), внешне похожая на printf(). Однако в NSLog( ) форматная строка в действительности является экземпляром NSString.

Экземпляры NSString можно создавать на программлом уровне при помощи метода класса stringWithFormat:

```
NSString *x = [NSString stringWithFormat:@"The best number is %d", 5];
```

Для получения количества символов встроке используется метод length:

```
NSUInteger charCount = [x length];
```

Метод isEqual: проверяет на равенство содержимое двух строк:.

```
if ([lament isEqual:x])
      NSLog(@"%@ and %@ are equal", lament, x);
```

В языке C тоже существует конструкция для работы со строками символов. Вот как предыдущий пример выглядел бы на языке C:

```
char *lament = "Why me!?";
char *x;
asprintf(&x, "The best number is %d", 5);
size_t charCount = strlen(x);
if (strcmp(lament, x) == 0)
printf("%s and %s are equal\n", lament, x);
free(x);
```

Строки C будут рассмотрены в главе 34. Там, где это возможно, используйте объекты NSString вместо строк C. Класс NSString содержит много методов, упрощающих вашу работу. Кроме того, класс NSString (базирующийся на стандарте

Юникод) отлично справляется с хранением текста на любом из языков нашей планеты. Незнакомые символы? Нет проблем. Текст читается справа
налево? Запросто.

Упражнение

Класс `NSHost` содержит информацию о вашем компьютере. Для получения экземпляра `NSHost` используется метод класса `NSHost` с именем `currentHost`. Этот метод возвращает указатель на объект `NSHost`:

```
+ (NSHost *)currentHost
```

Для получения локализованного имени компьютера используется метод экземпляра NSHost с именем localizedName, возвращающий указатель на объект `NSString`:

```
- (NSString *)localizedName
```

Напишите программу командной строки Foundation, которая выводит имя вашего компьютера. (Программа выполняется на удивление долго.)

15. NSArray

Класс NSArray, как и NSString, часто используется программистами Objective-C. В экземпляре NSArray хранится список указателей на другие объекты. Предположим, вы хотите создать список из трех объектов NSDate, а потом перебрать их и вы вести каждую из хранящихся в них дат.

Создайте новый проект: программу командной строки Foundation с именем DateList. Откройте файл *main.m* и отредактируйте функцию main():

```
#import <Foundation/Foundation.h>
int main(int argc, const char * argv[])
{
    @autoreleasepool {

      // создание трех объектов NSDate
      NSDate *now = [NSDate date];
      NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
      NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

      // создание массива, содержащего все три объекта
      NSArray *dateList = [NSArray arrayWithObjects:now, tomorrow, yesterday,
nil];

      // Сколько объектов содержит список?
      NSLog(@"There are %lu dates", [dateList count]);

      // вывод содержимого пары объектов
      NSLog(@"The first date is %@", [dateList objectAtIndex:0]);
      NSLog(@"The third date is %@", [dateList objectAtIndex:2]);
      }
      return 0;
}
```

Класс NSArray содержит два метода, представленных в этом примере, которыми вы будете постоянно пользоваться:

• count: возвращает количество элементов в массиве:
• objectAtlndex: возвращает указатель, хранящийся в массиве по индексу, заданному в аргументе.

Элементы массива хранится в заданном порядке, к ним можно обращаться по индексу. Нумерация индексов начинается с нуля: первый элемент хранится с индексом 0, второй - с индексом 1 и т. д. Таким образом, если метод count сообщает, что массив содержит 100 элементов, вы можете использовать objectAtlndex: для запроса объектов с индексами от 0 до 99.

На рис. 15.1 изображена диаграмма объектов программы. Обратите внимание: экземпляр NSArray содержит указатели па объекты NSDate.

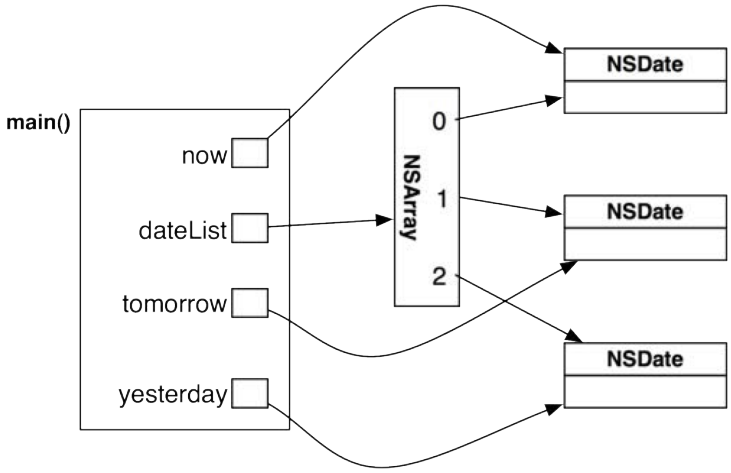


Рис. 15.1. Диаграмма объектов приложения DateList

Для перебора и поочередной обработки всех элементов массива используется цикл `for`. Отредактируйте файл *main.m*:

```
#import <Foundation/Foundation.h>
int main(int argc, const char * argv[])
{
    @autoreleasepool {

      // создание трех объектов NSDate
      NSDate *now = [NSDate date];
      NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
      NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

      // создание массива, содержащего все три объекта
      // nil - признак конца списка
      NSArray *dateList = [NSArray arrayWithObjects:now, tomorrow, yesterday,
nil];

      NSUInteger dateCount = [dateList count];
      for (int i = 0; i < dateCount; i++) {
             NSDate *d = [dateList objectAtIndex:i];
             NSLog(@"Here is a date: %@", d);
              }
      }
      return 0;
}
```

Программистам так часто приходится решать задачу перебора массивов, ЧТО для нее было создано специальное расширение цикла `for` (синтаксис *быстрого перечисления*) Внесите в код следующие изменения:

```
#import <Foundation/Foundation.h>
int main(int argc, const char * argv[])
{
    @autoreleasepool {

      // создание трех объектов NSDate
      NSDate *now = [NSDate date];
```

```
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // создание массива, содержащего все три объекта
        // nil - признак конца списка
        NSArray *dateList = [NSArray arrayWithObjects:now, tomorrow, yesterday,
nil];

        for (NSDate *d in dateList) {
        NSLog(@"Here is a date: %@", d);
              }
        }
        return 0;
}
```

Подобные циклы являются исключительно эффективным способом перебора элементов массива. Существует единственное ограничение: в процессе перебора запрещено добавлять или удалять элементы из массива.

NSMutableArray

Массивы делятся на две разновидности:

• Экземпляр `NSArray` создастся с конкретным списком указателей. Добавление и удаление указателей в таких массивах запрещено.
• Экземпляр `NSMutableArray` похож на `NSArray`, но он поддерживает добавление и удаление указателей. (`NSMutableArray` является *субклассом* `NSArray`. Субклассы более подробно рассматриваются в главе 18.)

Внесите изменения в программу так, чтобы в ней использовался экземпляр `NSMutableArray` и методы класса `NSMutableArray` :

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

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

    @autoreleasepool {

        // создание трех объектов NSDate
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // создание пустого массива
        NSMutableArray *dateList = [NSMutableArray array];

        // включение дат в массив
        [dateList addObject:now];
        [dateList addObject:tomorrow];
```

```
        // Put yesterday at the beginning of the list
        [dateList insertObject:yesterday atIndex:0];

        for (NSDate *d in dateList) {
            NSLog(@"Here is a date: %@", d);
        }

        // Remove yesterday
        [dateList removeObjectAtIndex:0];
        NSLog(@"Now the first date is %@", [dateList objectAtIndex:0]);

    }
    return 0;
}
```

Упражнения

Создайте новую программу командной строки Foundation с именем `Groceries`. Создайте пустой объект `NSMutableArray`, добавьте в него несколько элементов, описывающих покупки в магазине. (Элементы тоже необходимо создать.) Наконец, используйте синтаксис быстрого перечисления для вывода списка покупок.

Следующая задача будет посложнее. Прочитайте следующую программу для поиска распространенных имен собственных, содержащих две смежные буквы A:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
      @autoreleasepool {

      // файл читается в виде одной большой строки
      // без обработки возможных ошибок
      NSString *nameString = [NSString stringWithContentsOfFile:@"/usr/share/
dict/propernames"  encoding:NSUTF8StringEncoding  error:NULL];

      // разбиваем файл на массив строк
      NSArray *names = [nameString componentsSeparatedByString:@"\n"];

      // перебор строк, содержащихся в массиве
      for (NSString *n in names) {

      // поиск подстроки "aa" без учета регистра символов
      NSRange r = [n rangeOfString: @"AA" options:NSCaseInsensitive Search];

      // нашли?
      if (r.location != NSNotFound) {
         NSLog (@"%@", n);
      }}}
return 0;
}
```

Файл `/usr/share/dict/propernames` содержит список распространенных имен собственных, файл `/usr/share/dict/words` - список слов (то есть имен

собственных и нарицательных). Напишите на базе приведенной программы другую - для поиска имен собственных, которые одновременно являются нарицательными («Glen» - мужское имя, и «glen» - узкая лощина). В файле words имена собственные записаны с прописной буквы.

При упорядочении строк компьютер обычно считает, что символы верхнего регистра предшествуют символам нижнего регистра. Для сравнения без учета регистра используется метод caselnsensitiveCompare:

```
NSString *a = @"ABC";
NSString *b = @"abc";

if ([a caselnsensitiveCompare:b] == NSOrderedSame) {
      NSLog(@"a and b are equal");
}

if ([a caselnsensitiveCompare:b] == NSOrderedAscending){
      NSLog(@"a comes before b");
}

if ([a caselnsensitiveCompare:b] == NSOrderedDescending){
      NSLog(@"b comes before a");
}
```

16. Документация разработчика

В предыдущей главе я привел некоторые полезные сведения о классе `NSArray`. Когда вы начнете программировать без моей помощи (хотя бы при выполнении упражнений из этой книги), вам придется искать информацию самостоятельно. В этом нам поможет электронная документация от Apple.

Чтобы просмотреть документацию в Xcode, вызовите панель `Organizer` (щелкните на кнопке  в правой верхней части окна Xcode) и откройте вкладку `Documentation`.

Рис 16.1. Электронная документация

Фирма Apple предоставляет документацию пяти видов:

- *Справочники*: краткие описания всех классов Objective-C и всех функций C.
- *Руководства и пособия для начинающих*: руководства содержат более подробную информацию с концептуальной группировкой идей. Например, в руководстве программиста по обработке ошибок описаны многочисленные способы, которыми разработчики Mac и iOS могут получать информацию о возникших проблемах.

- *Примеры кода*: небольшие законченные проекты, которые демонстрируют, как фирма Apple рекомендует использовать свои технологии. Например, проект `WeatherMap` показывает, как создавать пользовательские аннотации при отображении карт в iOS.

- *Замечания к текущей версии*: при выходе каждой новой версии Mac OS или iOS прилагается документация, в которой описаны изменения текущей версии по сравнению с предыдущей.

- *Технические комментарии, статьи, советы по программированию, технические вопросы и ответы*: небольшие документы с описанием конкретных проблем.

Документацию можно просматривать, но ее объем очень велик. Чаще документация используется для поиска ответов на конкретные вопросы.

Справочные страницы

Щелкните на кнопке в верхней части левой панели `Organizer`, чтобы вызвать панель поиска. Введите условие поиска `NSArray`; ниже появляются результаты поиска. В разделе `Reference` щелкните на ссылке `NSArray`, чтобы просмотреть *справочную страницу* класса `NSArray`.

Трудно переоценить важность справочных страниц для вас и для программистов всех уровней. Фирма Apple основательно потрудилась над созданием огромных библиотек программного кода, упрощающих вашу работу. Основным источником информации об этих библиотеках являются справочные страницы. Не жалейте времени на просмотр справочных страниц новых классов, с которыми вы познакомитесь в этой книге, и ознакомьтесь с их возможностями. Также возможен поиск по именам методов, функций, констант и свойств (о том, что такое свойства, будет рассказано в главе 17). Чем увереннее вы будете работать со справочными страницами, тем быстрее пойдет ваша работа.

В начале справочного описания класса `NSArray` при водятся некоторые общие сведения о классе. В частности, класс является производным от `NSObject` (подробности также при водятся в главе 18). Он поддерживает некоторые протоколы (эта тема будет рассмотрена в главе 25). Ои является частью библиотеки Foundation и входит в поставку всех версий Mac OS X и всех версий iOS.

В конце заголовка приводится список руководств с описаниями класса `NSArray` и примеров кода, демонстрирующих его. Щелкните на ссылке *Collections Programming Topics*, чтобы открыть руководство по работе с коллекциями.

В этом конкретном руководстве обсуждается работа с классами коллекций (включая `NSArray` и `NSMutableArray`). В отличие от справочника по классам, руководство легко читается, а материал излагается логично и последовательно.

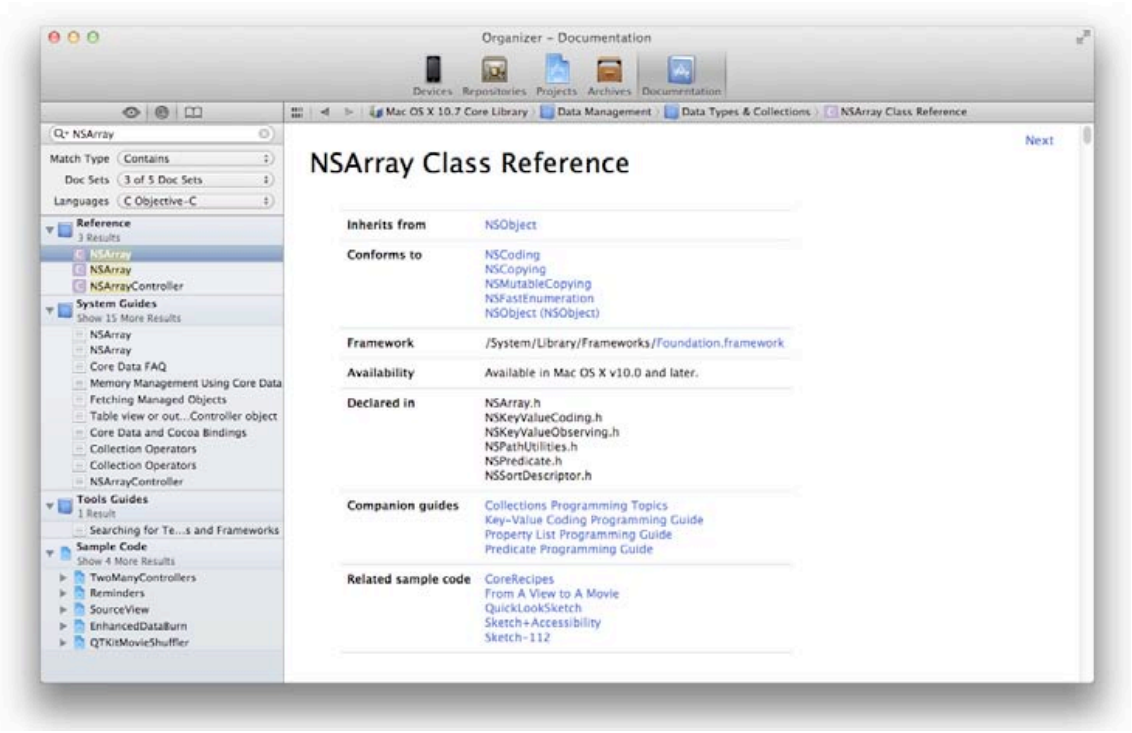


Рис. 16.2. Справочная страница NSArray

Нажмите кнопку возврата в верхней части панели документа, чтобы вернуться к справочной странице `NSArray`. А какие еще сообщения можно отправить `NSArray`? Прокрутите справочную страницу и найдите список методов, реализуемых `NSArray`. Допустим, как узнать, содержит ли массив заданный элемент? Найдите метод `containsObject`: и прочитайте его описание.

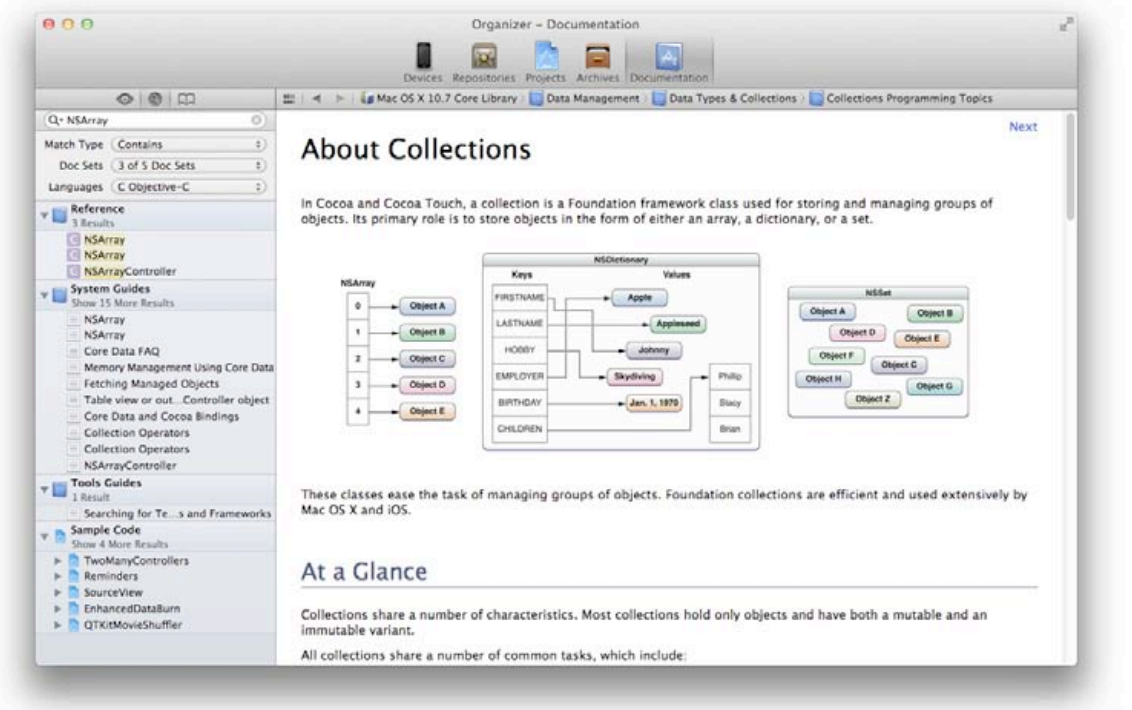
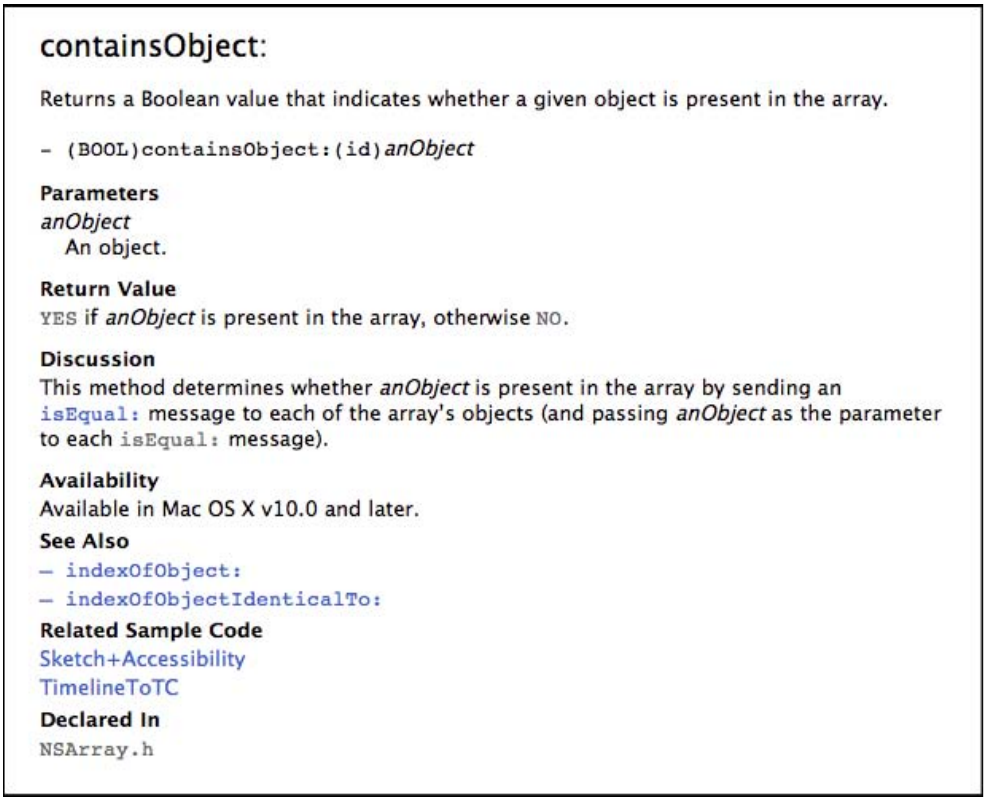


Рис. 16.3. Документация по работе с коллекциями



containsObject:

Returns a Boolean value that indicates whether a given object is present in the array.

- (BOOL)containsObject:(id)*anObject*

Parameters
anObject
An object.

Return Value
YES if *anObject* is present in the array, otherwise NO.

Discussion
This method determines whether *anObject* is present in the array by sending an isEqual: message to each of the array's objects (and passing *anObject* as the parameter to each isEqual: message).

Availability
Available in Mac OS X v10.0 and later.

See Also
– indexOfObject:
– indexOfObjectIdenticalTo:

Related Sample Code
Sketch+Accessibility
TimelineToTC

Declared In
NSArray.h

Рис 16.4. Справочная информация о containsObject:

И теперь вы знаете все необходимое для отправки этого сообщения в своем программном коде.

Быстрая справка

Однако существует и другой, более простой способ перехода из редактора, в котором вы пишете код, к источнику знаний в документации. Закройте панель `Organizer` и вернитесь к проекту `DateList`. Найдите в файле *main.m* строку с фрагментом `[dateList addObject:now]`. Удерживая нажатой клавишу `Option`, щелкните на `addObject:`. На экране появляется окно быстрой справки с информацией об этом методе.

Обратите внимание на ссылки в окне быстрой справки. Если щелкнуть на ссылке, соответствующая документация открывается в `Organizer`. Удобно, правда?

Если вы хотите, чтобы окно быстрой справки оставалось на экране постоянно, откройте его в виде панели Xcode. в правом верхнем углу окна Xcode найдите сегментированный элемент управления `View`, который выглядит так:
он представляет левую, нижнюю и правую панели Xcode. Включение/отключение этих кнопок приводит к сокрытию/отображению соответствующих панелей.

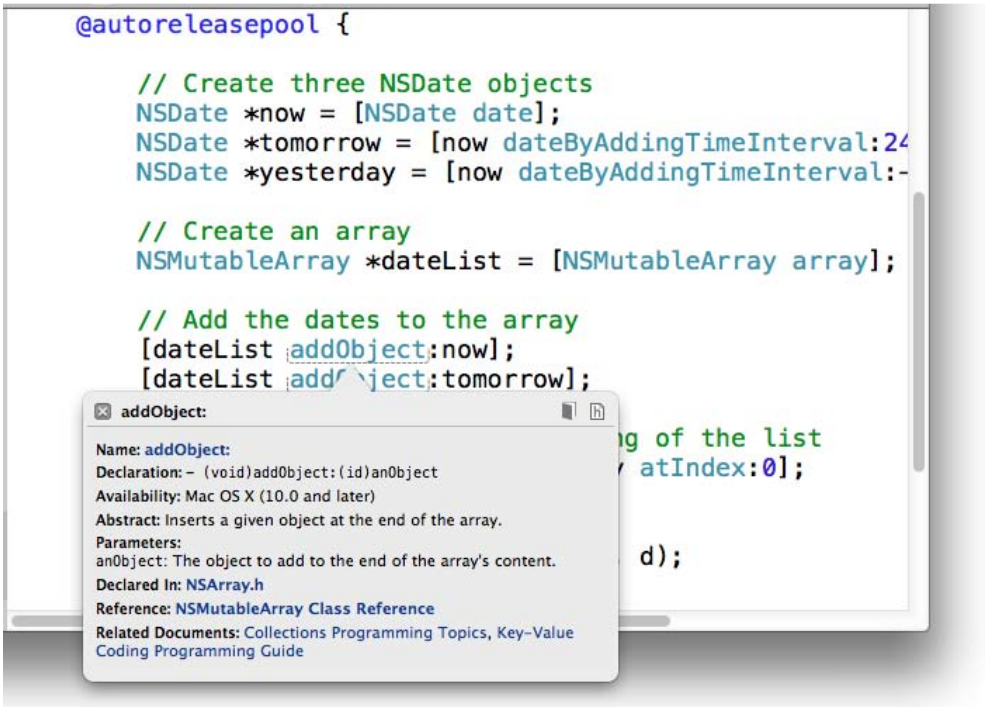


Рис. 16.5. Окно быстрой справки

Левая панель, с которой мы уже неоднократно работали, называется навигатором. Нижняя панель, которая тоже нам попадалась, содержит консоль и называется областью отладки. Правая панель называется служебной панелью (`Utilities`). Щелкните на правой кнопке элемента `View`, чтобы открыть панель `Utilities`. В верхней части панели щелкните на кнопке , чтобы открыть панель быстрой справки.

На панели будет выводиться справка для текста, выбранного в редакторе. Попробуйте сами: просто выберите в редакторе слово `NSMutableArray`:

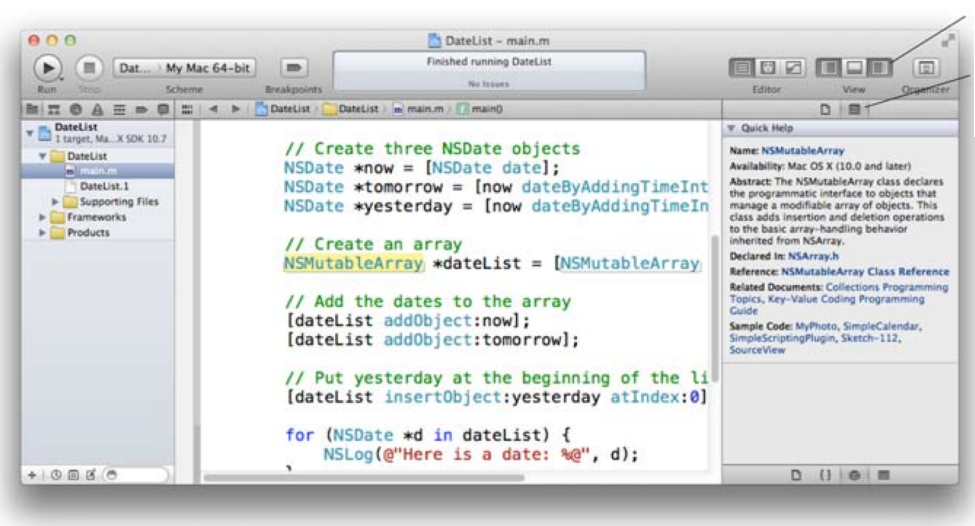


Рис. 16.6. Панель быстрой справки

Если выделить в редакторе другой текст, панель быстрой справки немедленно обновится, и на ней появится документация, относящаяся к вашему новому выбору.

Другие возможности и ресурсы

Если сделать двойной щелчок на любом тексте с нажатой клавишей `Option`, `Organizer` проведен поиск этой строки.

Если щелкнуть на любом классе, функции или методе с нажатой клавишей `Command`, Xcode откроет файл, в котором находится соответствующее объявление. (Попробуйте щелкнуть с нажатой клавишей `Command` на слове `NSMutableArray`).

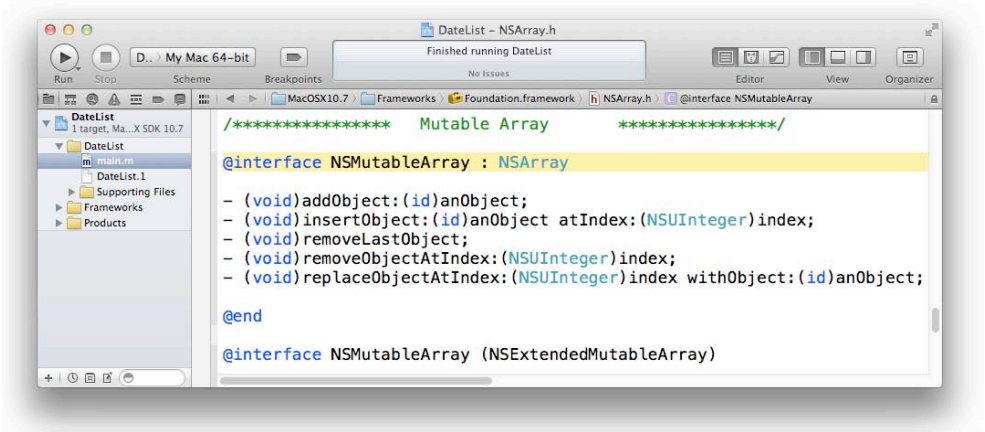


Рис. 16.7. Объявление `NSMutableArray`

Если выбранному тексту соответствует несколько объявлений, ври щелчке с нажатой клавишей `Command` откроется временное окно. Например, попробуйте сделать щелчок с нажатой клавишей `Command` на `addObject:`.

Время от времени Xcode проверяет, не появилась ли обновленная документация. Все обновления автоматически загружаются и индексируются для поиска. Также можно выполнить немедленную проверку обновлений настроек Xcode.

Фирма Apple ведет несколько дискуссионных рассылок по адресу *http//lists.apple.com/*; некоторые из них посвящены темам разработки. Если вам не удастся найти ответ на интересующий вас вопрос в документации, попробуйте провести поиск в архивах этих рассылок (Из уважения к тысячам участников каждого списка, пожалуйста, поищите ответ в архивах перед тем, как публиковать свой вопрос.)

Также следует упомянуть еще об одном ресурсе: *stackouerflow.com*. Это весьма популярный сайт, на котором программисты задают вопросы и отвечают на них. Если вам удастся подобрать несколько ключевых слов, описывающих возникшую проблему, скорее всего, вы найдете здесь ответ.

17. Наш первый класс

До сих пор мы использовали только готовые классы, созданные фирмой Apple. Пришло время заняться написанием собственных классов. Вспомните, что класс описывает две «стороны» объектов:

• методы (экземпляров и класса), реализуемые классом;
• переменные экземпляров, содержащиеся в каждом экземпляре класса.

Сейчас мы создадим класс `Person`, похожий на структуру `Person`, которую мы создали в главе 10. Класс будет определяться в двух файлах: *Person.h* и *Person.m*. Файл *Person.h*, называемый заголовочным или интерфейсным файлом, содержит объявления переменных и методов экземпляров. Файл *Person.m* называется *файлом реализации*. В нем программируются действия, необходимые для выполнения каждого метода.

Создайте новый проект: программу командной строки Foundation с именем `BMITime`.

Чтобы создать новый класс, выберите команду File→New→New File.... из раздела Mac OS X на левой панели, выберите Cocoa и шаблон Objective-C class. Присвойте классу имя `Person` и назначьте его субклассом `NSObject`. Наконец, убедитесь в том, что в окне выбрана цель `BMITime`, и щелкните на кнопке `Save`.

Теперь в навигаторе проекта отображаются файлы нового класса, *Person.h* и *Person.m*. Откройте файл *Person.h*, включите в него объявления двух переменных экземпляров и трех методов экземпляров:

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

// класс Person наследует все переменные экземпляров
// и методы, определенные в классе NSObject
@interface Person : NSObject
{
    // класс содержит две переменные экземпляров
    float heightInMeters;
    int weightInKilos;
}

// методы, которые будут использоваться для присваивания
// значений переменных экземпляров
- (void)setHeightInMeters:(float)h;
- (void)setWeightInKilos:(int)w;

// метод для вычисления ИМТ
- (float)bodyMassIndex;

@end
```

Обратите внимание: переменные экземпляров объявляются в фигурных скобках, а методы - после переменных и вне фигурных скобок.

С точки зрения компилятора этот код говорит: «Я определяю новый класс с именем `Person`, и который входят все методы и переменные экземпляров класса `NSObject`. К ним я добавляю две переменные экземпляров: переменную типа float с именем `heightInMeters` и переменную типа `int` с именем `weightInKilos`. Также я собираюсь добавить три метода экземпляров, которые будут реализованы в *Person.m*».

Метод `setHeightInMeters`: получает аргумент типа float и не возвращает значения, метод `setWeightInKilos`: получает аргумент тина `int` и не возвращает значения, а метод `bodyMassIndex` не получает аргументов и возвращает значение типа float.

Откройте файл *Person.m*. Удалите все существующие методы и реализуйте методы, которые были объявлены:

```
#import "Person.h"
@implementation Person
- (void)setHeightInMeters:(float)h
{
    heightInMeters = h;
}
- (void)setWeightInKilos:(int)w
{
    weightInKilos = w;
}
- (float)bodyMassIndex
{
    return weightInKilos / (heightInMeters * heightInMeters);
}
@end
```

Как видите, среда Xcode импортировала файл *Person.h* за вас. Также обратите внимание на то, что имена реализованных методов должны в точности совпадать с именами, объявленными в заголовочном файле. В Xcode это делается просто; когда вы начинаете вводить метод в файле реализации, Xcode предлагает выбрать одно из имен уже объявленых методов.

Итак, все методы, объявленые в *Person.h*, реализованы; наш класс готов. Отредактируйте файл *main.m*, чтобы немного по экспериментировать с ним:

```
#import <Foundation/Foundation.h>
#import "Person.h"
int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // создание экземпляра Person
        Person *person = [[Person alloc] init];

        // присваивание значение переменным экземпляра
        [person setWeightInKilos:96];
        [person setHeightInMeters:1.8];

        // вызов метода bodyMassIndex
```

```
        float bmi = [person bodyMassIndex];
        NSLog(@"person has a BMI of %f", bmi);
}
return 0;
}
```

Постройте и запустите программу. Файл *Person.h* импортируется для того, чтобы компилятор знал, какие методы были объявлены до их использования в `main`().

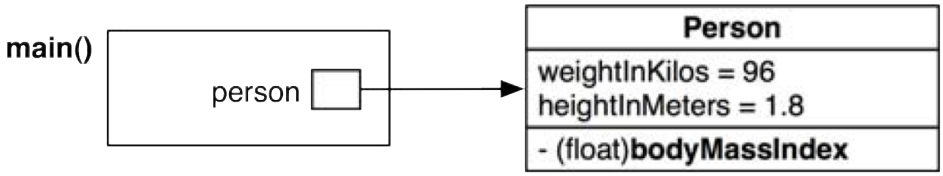


Рис. 17.1. Диаграмма объектов BMITime

Методы доступа

Выполняя это же упражнение со структурами вместо объектов, мы напрямую обращались к полям данных структуры из `main` ():

```
person.weightInKilos = 96;
person.heightInMeters = 1.8;
```

В объектно-ориентированных программах следует по возможности скрыть переменные экземпляров, чтобы они были известны и доступны только для самого объекта. А чтобы функции и методы за пределами `Person` могли задать значения веса и роста, мы создали методы `setWeightlnKilos`: и `setHeightInMeters`:. Такие методы называются `set`-методами (или методами записи).

`Set`-метод позволяет другим методам задать значение переменной экземпляра. `Get`-метод (метод чтения) позволяет другим методам прочитать значение переменной экземпляра. `Set`- и `get`- методы также называются *методами доступа*.

Добавьте в *Person.h* объявления `get`-методов:

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    float heightInMeters;
    int weightInKilos;
}
// вы сможете задавать значения следующих переменных экземпляров
- (float)heightInMeters;
- (void)setHeightInMeters:(float)h;
- (int)weightInKilos;
- (void)setWeightInKilos:(int)w;
- (float)bodyMassIndex;
```

```
@end
```

Почему имена `get`-методов не включают префикс `get` по аналогии с именами `set`-методов? Потому что такое соглашение используется в Objective-C. Имя метода для чтения переменной экземпляра просто совпадает с именем этой переменной.

Теперь вернитесь к файлу *Person.m* и добавьте реализации `get`-методов:

```
@implementation Person
- (float)heightInMeters
{
    return heightInMeters;
}
- (void)setHeightInMeters:(float)h
{
    heightInMeters = h;
}
- (int)weightInKilos
{
    return weightInKilos;
}
- (void)setWeightInKilos:(int)w
{
    weightInKilos = w;
}
- (float)bodyMassIndex
{
    return weightInKilos / (heightInMeters * heightInMeters);
}
@end
```

Остается использовать эти методы в `main.m`:

```
#import "Person.h"
int main(int argc, const char * argv[])
{
@autoreleasepool {
// создание экземпляра person
Person *person = [[Person alloc] init];

// присваивание значений переменным экземпляра
[person setWeightInKilos:96];
[person setHeightInMeters:1.8];
// вызов метода bodyMassIndex
float bmi = [person bodyMassIndex]; NSLog(@"person (%d, %f) has a BMI of %f",
[person weightInKilos], [person heightInMeters], bmi);

            }
      return 0;
}
```

Постройте и запустите программу.

Точечная запись

Программисты Objective-C часто используют методы доступа. Фирма Apple решила создать сокращенную запись для вызова методов доступа. Следующие две строки кода эквивалентны:

```
p = [x fido];
p = x.fido;
```

и эти две строки тоже работают одинаково:

```
[x setFido:3];
x.fido = 3;
```

Форма, в которой вместо квадратных скобок используется символ «точка», называется точечной записью.

Я стараюсь обходиться без использования точечной записи. Мне кажется, что она маскирует факт отправки сообщения и не соответствует соглашениям отправки всех остальных сообщений в системе. При желании вы можете использовать точечную завись в своих программах, но в этой книге она не встречается.

Свойства

Большая часть кода класса `Person` относится к методам доступа. Фирма Apple создала удобный механизм, упрощающий написание методов доступа, - так называемые *свойства*. При использовании свойств `set`- и `get`- методы можно объявлять в одной строке.

В файле *Person.h* замените объявление `set`- и `get`- методов конструкцией `@property`:

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    float heightInMeters;
    int weightInKilos;
}
@property float heightInMeters;
@property int weightInKilos;
- (float)bodyMassIndex;
@end
```

Класс от этой правки не изменяется - просто вы используете более компактный синтаксис.

Теперь загляните в файл *Person.m*. В нем полно простых, предсказуемых методов доступа. Если ваши методы доступа не делают ничего особенного, можно

просто приказать компилятору синтезировать методы доступа по умолчанию на основании объявления `@property`. Удалите методы доступа и замените их вызовом `@synthesize`:

```
#import "Person.h"
@implementation Person
@synthesize heightInMeters, weightInKilos;
- (float)bodyMassIndex
{
    return weightInKilos / (heightInMeters * heightInMeters);
}
@end
```

Постройте и запустите программу. Все работает точно так же, как прежде, но код стал куда более простым и удобным в сопровождении.

self

В каждом методе доступна локальная переменная `self`. Она содержит указатель на объект, для которого был вызван метод. Переменная `self` используется тогда, когда объект хочет отправить сообщение самому себе. Например, многие программисты Objective-C крайне ревностно относятся к использованию методов доступа; они никогда не читают и не записывают значения переменных экземпляров напрямую, а только через методы доступа. Измените метод `bodyMasslndex`, чтобы ваш метод устроил даже самых ревностных блюстителей чистоты языка:

```
- (float)bodyMassIndex
{
    float h = [self heightInMeters];
    return [self weightInKilos] / (h * h);
}
```

Экземпляр `Person` отправляет самому себе два сообщения, `heightlnMeters` и `weightlnKilos`, для получения значений переменных экземпляров.

Значение `self` также можно передать в аргументе, чтобы другие объекты знали адрес текущего объекта. Например, ваш класс `Person` может содержать метод `addYourselfToArray`: следующего вида:

```
- (void)addYourselfToArray:(NSMutableArray *)theArray
{
    [theArray addObject:self];
}
```

Здесь при помощи `self` мы сообщаем массиву, где находится экземпляр `Person` - то есть указываем его адрес.

Проекты из нескольких файлов

Теперь в нашем проекте исполняемый код хранится в двух файлах: *main.m* и *Person.m*. (Файл *Person.h* содержит объявление класса, исполняемого кода в нем нет.) При построении проекта эти файлы компилируются по отдельности, а затем связываются в единое целое. Реальные проекты нередко состоят из сотен файлов кода C и Objective-C:

К проектам также можно подключать библиотеки программного кода. Допустим, вы нашли отличную библиотеку для обработки данных, полученных с цифрового телескопа. Если вашей программе необходима такая функциональность, вы компилируете библиотеку и добавляете ее в ваш проект. При построении программы Xcode подключает к ней все функции и классы, определенные в библиотеке.

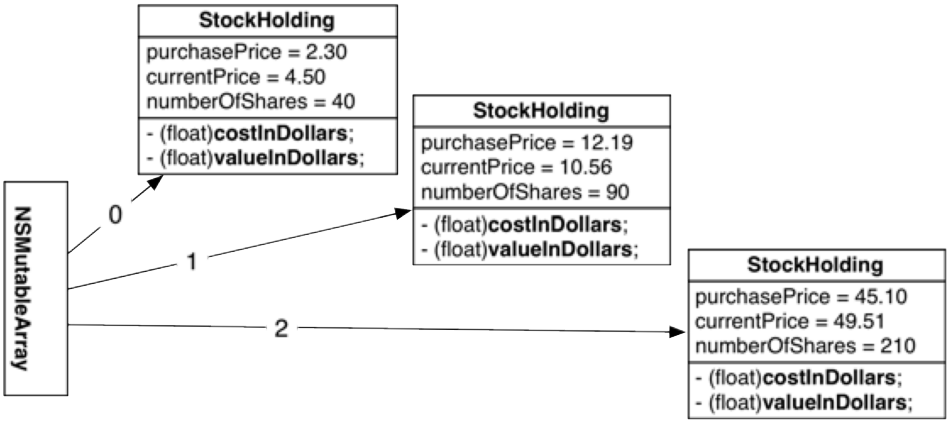


Рис. 17.2. Массив объектов StockHolding

Упражнение

Создайте новую программу командной строки Foundation с именем `Stocks`. Затем создайте класс `StockHolding` для представления купленных акций. Этот класс должен быть субклассом `NSObject`. Он содержит две переменные экземпляра типа float с именами `purchaseSharePrice` и `currentSharePrice` и одну переменную экземпляра типа `int` с именами `numberOfShares`. Создайте методы доступа для переменных экземпляров. Определите два метода экземпляра:

```
- (float)costInDollars;  // purchaseSharePrice * numberOfShares
- (float)valueInDollars; // currentSharePrice * numberOfShares
```

В функции `main`() заполните массив тремя экземплярами `StockHolding`. Переберите элементы массива и выведите значения каждого из них.

18. Наследование

При создании класса `Person` мы объявили его субклассом `NSObject`. Это означает, что каждый экземпляр `Person` будет содержать методы и переменные экземпляров, определенные в `NSObject`, а также методы и переменные экземпляров, определенные в `Person`. Говорят, что `Person` *наследует* методы и переменные экземпляров от `NSObject`. В этом разделе мы изучим наследование более подробно.

Откройте проект `BMITime` и создайте новый файл: класс Objective-C с именем `Employee`, являющийся субклассом `NSObject`. Вскоре мы изменим класс `Employee` и сделаем его субклассом `Person`. Вполне логично, верно? Работник (`Employee`) является разновидностью человека (`Person`). У них есть рост и вес. Тем не менее не каждый человек является работником. Мы также включим в наш класс переменную экземпляра, относящуюся к нашему конкретному классу, - табельный номер работника.

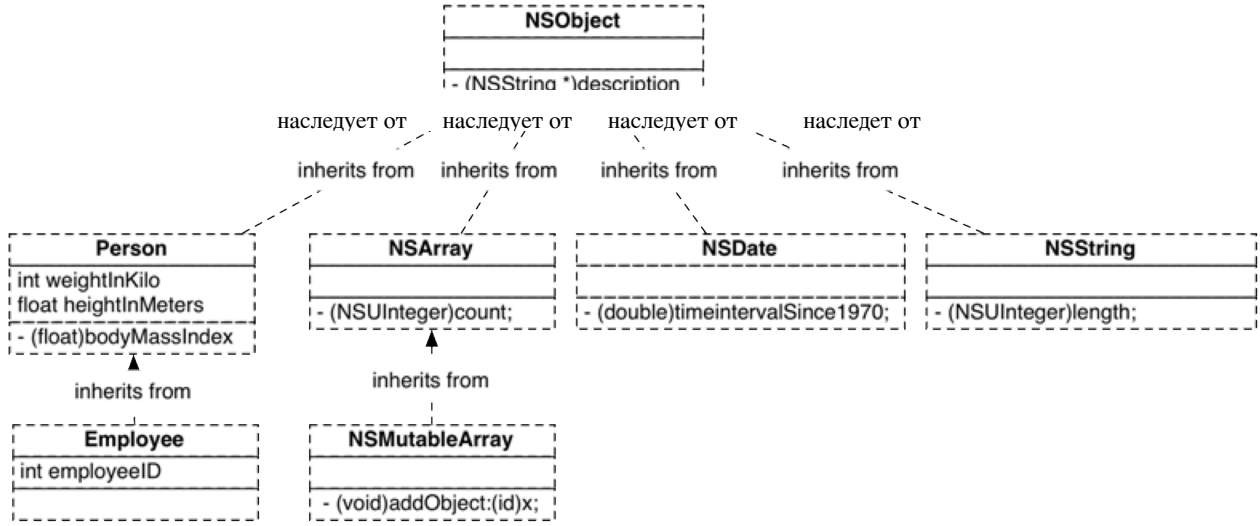


Рис. 18.1. Диаграмма наследования некоторых знакомых классов

Откройте файл *Employee.h*. Импортируйте *Person.h*, замените суперкласс на `Person` и добавьте переменную экземпляра для хранения табельного номера работника:

```
#import "Person.h"
@interface Employee : Person
{
    int employeeID;
}
@property int employeeID;
@end
```

Откройте *Employee.m* и синтезируйте методы доступа:

```
#import "Employee.h"
@implementation Employee
@synthesize employeeID;
@end
```

Теперь у нас имеется новый класс `Employee`, содержащий все переменные экземпляров `Person` и новую переменную экземпляра с именем `employeeID`. Экземпляры `Employee` отвечают на те же сообщения, что и экземпляр `Person`. Кроме того, экземпляры Employee отвечают на сообщения `setEmployeeID`: и `employeeID`.

Так как класс `Person` является субклассом `NSObject`, `Employee` наследует все методы и переменные экземпляров `NSObject`. Все объекты наследуют (прямо или косвенно) от `NSObject`.

Класс `NSObject` содержит много методов, но только одну переменную экземпляра: указатель `isa`. Указатель `isa` каждого объекта ссылается на класс, его создавший.

Допустим, вы отправили объекту сообщение `fido`. Чтобы ответить на это сообщение, объект при помощи указателя `isa` определяет свой класс и спрашивает: "Есть ли у тебя метод экземпляра с именем `fido`? Если в классе существует метод экземпляра `fido`, то он выполняется. Если у класса такого метода нет, он спрашивает у своего суперкласса: «Есть ли у тебя метод экземпляра с именем `fido`? Так поиск метода с именем fido продолжается вверх 1\0 самого начала цепочки. Поиск прекращается при обнаружении метода `fido` или достижении последнего звена. В начале цепочки класс `NSObject` говорит: «Нет, метода `fido` у меня нет», а вы получаете сообщение об ошибке - что-то типа «Экземпляр `Employee` не отвечает на селектор `fido`».

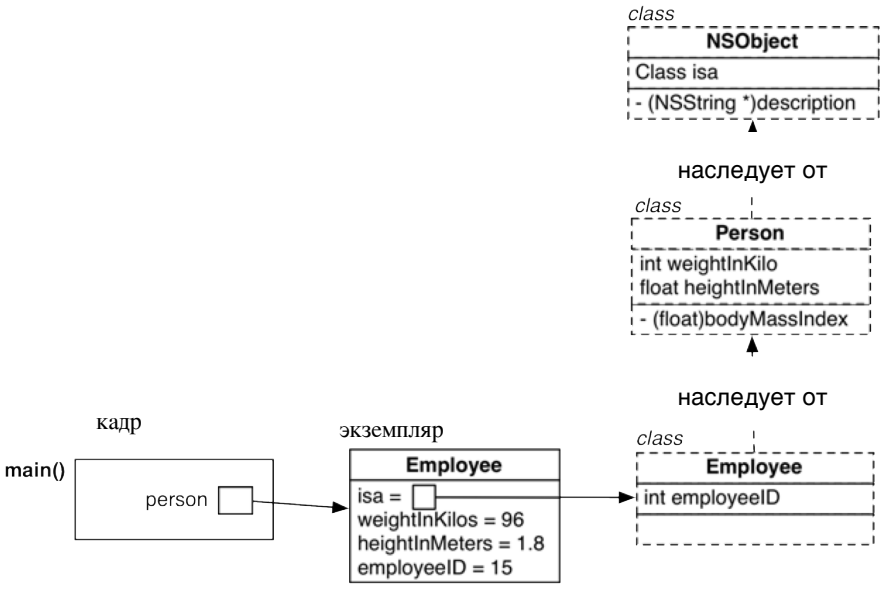


Рис. 18.2. Диаграмма объектов BMITime

Попробуйте сами - откройте файл *main.m* и отправьте экземпляру `Person` сообщение, которое он не поймет:

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

        // создание экземпляра Person
        id person = [[Person alloc] init];
        // присваивание значений переменным экземпляров
        [person setWeightInKilos:96];
        [person setHeightInMeters:1.8];

        // вызов метода bodyMassIndex
        float bmi = [person bodyMassIndex];
        NSLog(@"person (%d, %f) has a BMI of %f",
              [person weightInKilos], [person heightInMeters], bmi);

        [person count];
            }
      return 0;
}
```

Постройте и запустите программу. (На предупреждение компилятора не обращайте внимания, потому что мы делаем это сознательно. В большинстве случаев на предупреждение обязательно стоит отреагировать!) На консоль выводится сообщение об исключении времени выполнения:

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
   reason: '-[Person count]: unrecognized selector sent to instance 0x100108de0'
```

Просмотрите описание исключения и удалите проблемную строку, прежде чем продолжать.

Мы создали новый класс `Employee`, но еще не использовали его. Измените файл *main.m*, чтобы в нем использовался класс `Employee`:

```
#import <Foundation/Foundation.h>
#import "Employee.h"
int main(int argc, const char * argv[])
{
    @autoreleasepool {

        // создание экземпляра Person
        Person * person = [[Employee alloc] init];
        // присваивание значений переменным экземпляра
        [person setWeightInKilos:96];
        [person setHeightInMeters:1.8];

        // вызов метода bodyMassIndex
        float bmi = [person bodyMassIndex];
        NSLog(@"person (%d, %f) has a BMI of %f",
            }
      return 0;
}
```

Обратите внимание: переменная person по-прежнему объявляется как указатель на Person. Не создаст ли это проблем? Постройте и запустите программу - вы увидите, что она работает нормально. Дело в том, что работник (`Employee`) является частным случаем человека (`Person`) - он может сделать все, что может человек Таким образом, мы можем использовать экземпляр Employee везде, где программа ожидает получить экземпляр `Person`.

Но сейчас мы используем метод, уникальный для `Employee`, поэтому тип переменной-указателя придется изменить:

```
#import <Foundation/Foundation.h>
#import "Employee.h"
int main(int argc, const char * argv[])
{
    @autoreleasepool {

// создание экземпляра Person
Employee *person = [[Employee alloc] init];

// присваивание значений переменным экземпляра
[person setWeightInKilos:96];
[person setHeightInMeters:1.8];
[person setEmployeeID:15];

// вызов метода bodyMassIndex
float bmi = [person bodyMassIndex];
NSLog(@"Employee %d has a BMI of %f", [person employeeID], bmi);
            }
      return 0;
}
```

Переопределение методов

Итак, при отправке сообщения поиск метода С указанным именем начинается с класса объекта и переходит вверх по цепочке наследования. Выполняется первая найденная реализация; это означает, что унаследованные методы можно переопределять пользовательским реализациями. Допустим, вы решили, что индекс массы тела работников всегда должен быть равен 19. для этого метод `bodyMassIndex` можно переопределить в классе `Employee`. Откройте файл *Employee.m* и внесите изменения:

```
#import "Employee.h"
@implementation Employee
@synthesize employeeID;
- (float)bodyMassIndex
{
    return 19.0;
}
@end
```

Постройте и запустите программу. Обратите внимание: выполняется наша новая реализация bodyMassIndex, а не реализация класса Person.

Метод bodyMassIndex реализован в Employee. т, но он не объявляется в файле Employee.h. Объявление метода в заголовочном файле публикует информацию о методе, чтобы он мог вызываться экземплярами других классов. Но поскольку Employee наследует от Person, все классы уже знают, что экземпляры Employee содержат метод bodyMassIndex. Сообщать об этом заново не нужно.

super

А если вы решили, что у всех работников ИМТ, вычисленный в реализации Person, должен автоматически уменьшаться на 10%? Конечно, можно продублировать код в реализации Employee, но гораздо удобнее было бы вызвать версию bodyMassIndex класса Person и умножить результат на 0,9, прежде чем возвращать его. Для решения этой задачи используется директива super. Попробуем использовать ее в Employee.m:

```
#import "Employee.h"
@implementation Employee
@synthesize employeeID;

- (float)bodyMassIndex
{
    float normalBMI = [super bodyMassIndex];
    return normalBMI * 0.9;
}

@end
```

Постройте и запустите программу.

Точнее говоря, директива super означает: «Выполнитъ этот метод, но начать поиск реализации с моего супер класса».

Упражнение

Это упражнение строится на базе упражнения из предыдущей главы. Создайте класс ForeignStockHolding, наследующий от StockHolding. Включите в ForeignStockHolding дополнительную переменную экземпляра conversionRate, относящуюся к типу float. (Умножая курс перевода conversionRate на местную цену, мы получаем цену в долларах США. Будем считать, что purchasePrice и currentPrice задаются в местной валюте.) Переопределите costlnDollars и valuelnDollars, чтобы они правильно работали.

В функции `main()` включите в свой массив несколько экземпляров `ForeignStockHolding`.

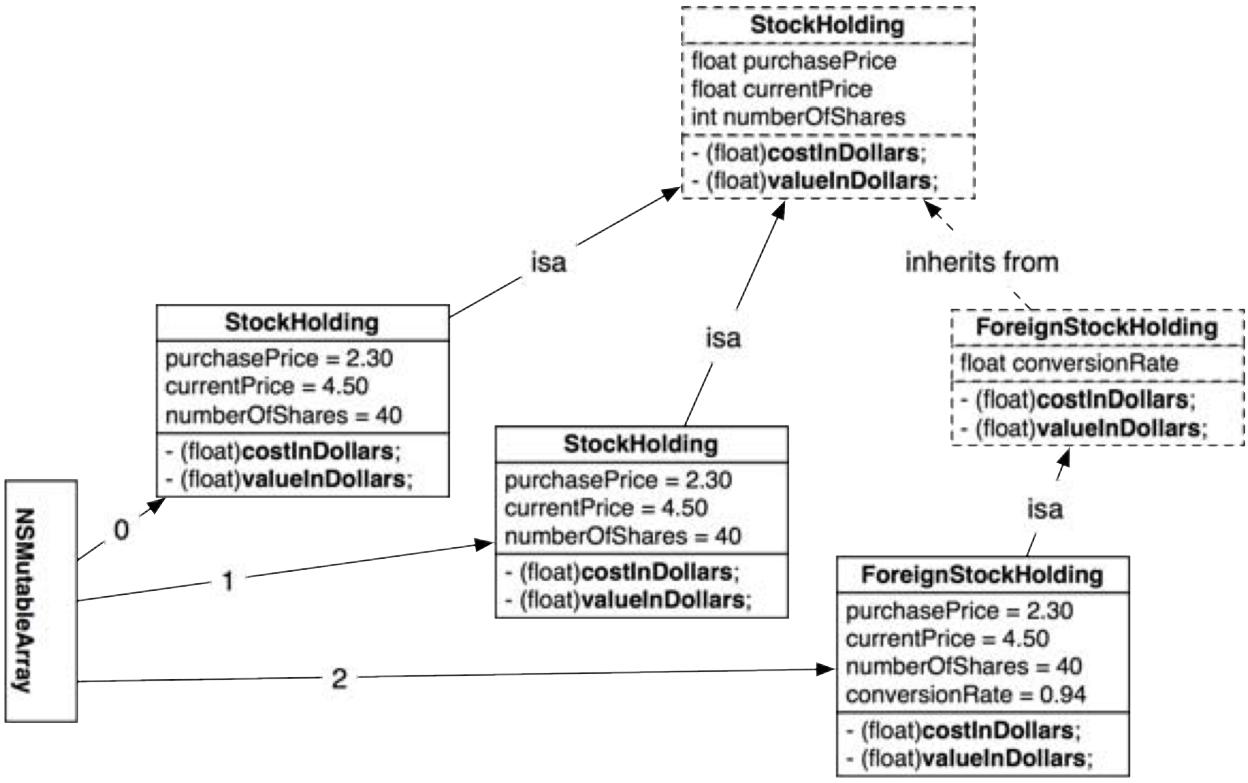


Рис. 18.3. Объекты `StockHolding` и `ForeignStockHolding`

19. Объектные переменные экземпляров

До настоящего момента переменные экземпляров, объявлявшиеся в наших классах, относились к простым типам C - таким, как `int` или `float`. На практике переменные экземпляров гораздо чаще содержат указатели на другие объекты. Объектная переменная экземпляра указывает на другой объект и описывает связь между двумя объектами. Как правило, объектные переменные экземпляров относятся к одной из трех категорий:

- *Атрибуты объектного типа*: указатель на простой объект-значение - такой, как `NSString` или `NSNumber`. Например, фамилия работника из класса `Employee` будет храниться в объекте `NSString`. Соответственно экземпляр `Employee` содержит переменную экземпляра, в которой хранится указатель на экземпляр `NSString`.
- *Отношения типа «один к одному»*: указатель на один составной объект. Например, у работника может быть жена; в экземпляр `Employee` включается переменная экземпляра, которая содержит указатель на один экземпляр `Person`.
- *Отношения типа «один ко многим»*: указатель на экземпляр класса коллекции - например, `NSMutableArray`. (Другие примеры коллекций встретятся нам в главе 21.) Например, у работника могут быть дети; в экземпляр включается переменная экземпляра, которая содержит указатель на экземпляр `NSMutableArray` со списком указателей на один или несколько объектов `Person`.

В этом списке под `NSString` понимается «*экземпляр* класса `NSString`». Возможно, на первый взгляд такое сокращение выглядит непривычно, но оно очень широко распространено.

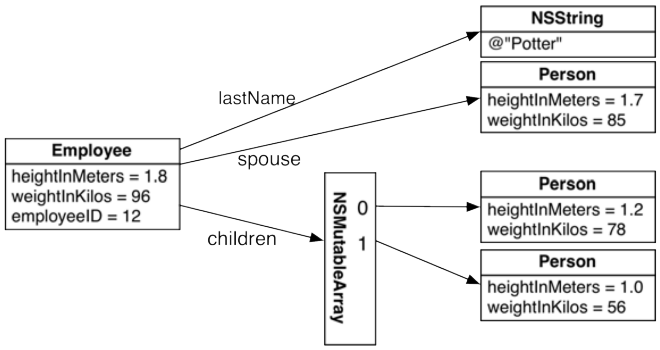


Рис. 19.1. Класс Employee с объектными переменными экземпляров

Как и на других диаграммах, указатели изображены стрелками (а также помечены именами переменных). Таким образом, класс `Employee` содержит три

переменные экземпляров: `lastName`, spouse и `children`. Объявление переменных экземпляров `Employee` будет выглядеть так:

```
@interface Employee : Person
{
    int employeeID;
    NSString *lastName;
    Person *spouse;
    NSMutableArray *children;
}
```

Все эти переменные, за исключением `employeeID`, являются указателями. Объектные переменные экземпляров всегда Являются указателями. Например, переменная `spouse` содержит указатель на другой объект, «живущий» в куче. Указатель `spouse` хранится в объекте `Employee`, а объект `Person`, на который он ссылается - нет. Объекты не хранятся внутри других объектов. Объект работника содержит свой табельный номер (переменная и само значение), но он только знает адрес, по которому в памяти хранится объект его жены.

У хранения в объектах указателей на объекты (вместо самих объектов) есть два важных побочных эффекта:

- Один объект может принимать несколько ролей. Например, может оказаться, что объект жены работника также указан в качестве контактного лица в объектах детей.

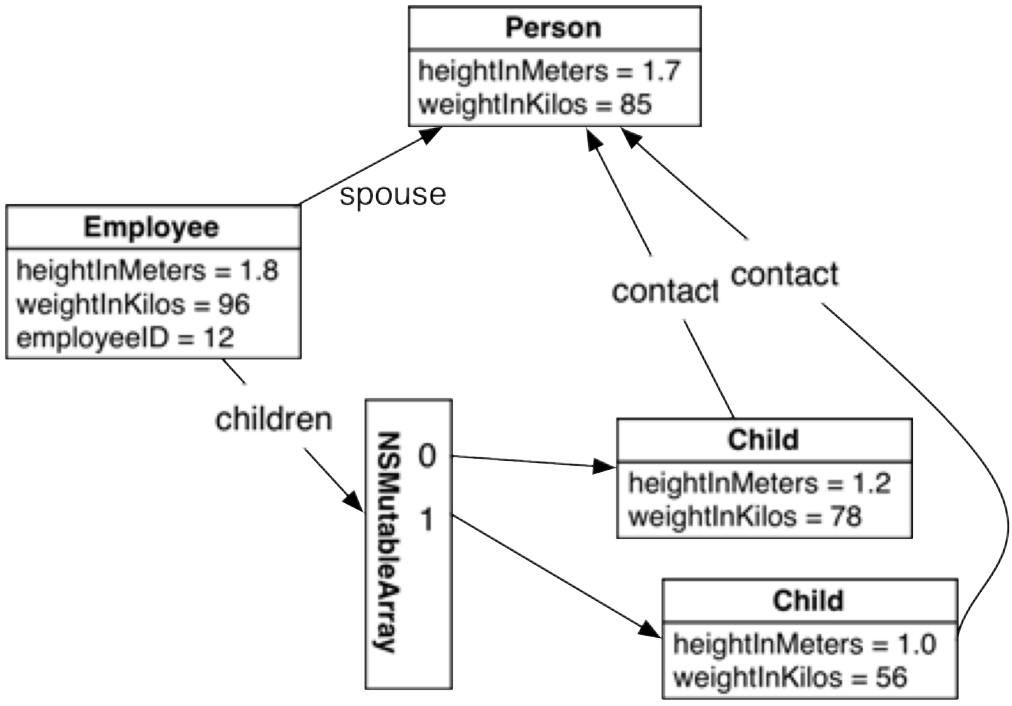


Рис. 19.2. Один объект в нескольких ролях

- В памяти программы хранится большое количество всевозможных объектов. Используемые объекты должны оставаться в памяти, но ненужные объекты должны уничтожаться (их память должна возвращаться в кучу для повторного

использования). Повторное использование памяти сводит к минимуму затраты памяти на выполнение программы, а компьютер быстрее реагирует на действия пользователя. На мобильных устройствах (например, iPhone) операционная система уничтожает программу, занимающую слишком много памяти.

Владельцы объектов и ARC

Для решения этой проблемы была разработана концепция *владения* объектами. Когда объект является объектной переменной экземпляра, говорят, что он является владельцем объекта, на который указывает переменная.

С другой стороны, объект знает, сколько владельцев он имеет в настоящий момент. Например, на приведенной выше диаграмме экземпляр `Person` имеет трех владельцев: объект `Employee` и два объекта `Child`. Когда количество владельцев объекта уменьшается до нуля, объект считает, что он больше никому не нужен, и удаляет себя из памяти.

Подсчетом владельцев каждого объекта занимается механизм `ARC` (`Automated Reference Counting`) - в Objective-C эта функция появилась недавно.

До выхода Xcode 4.2 управлять подсчетом владельцев приходилось вручную, на что уходило много времени и усилий. (О ручном подсчете владельцев о том, как он работал, рассказано в последнем разделе главы 20. Впрочем, во всем коде, приведенном в книге, предполагается, что вы используете `ARC`.)

Давайте немного доработаем проект `BMITime` и посмотрим, как подсчет владельцев работает на практике. Компании довольно часто следят за тем, какие материальные ресурсы были выделены тому или иному работнику. Мы создадим класс `Asset`, а в каждом объекте `Employee` будет храниться массив с описанием ресурсов, доверенных данному работнику.

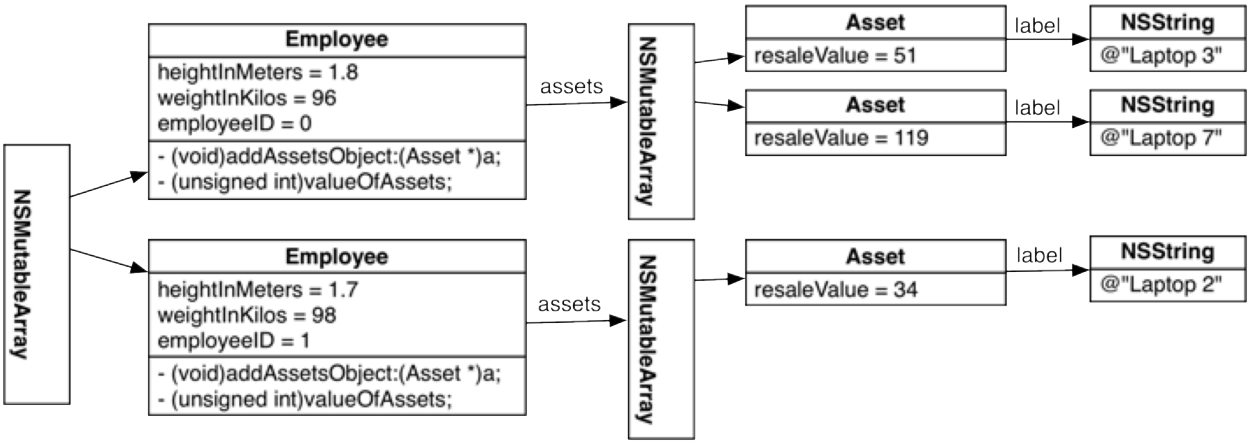


Рис 19.3. Работники и ресурсы

Подобные иерархии часто называются отношениями «родитель/потомок»: родитель (экземпляр `Employee`) содержит коллекцию потомков (коллекция `NSMutableArray` с объектами `Asset`).

Создание класса Asset

Создайте новый файл для субкласса `NSObject`. Присвойте ему имя `Asset`. Откройте файл *Asset.h*, объявите две переменные экземпляров и два свойства:

```
#import <Foundation/Foundation.h>
@interface Asset : NSObject
{
    NSString *label;
    unsigned int resaleValue;
}
@property (strong) NSString *label;
@property unsigned int resaleValue;
@end
```

Обратите внимание на модификатор `strong` у объявления `@property` для `label`. Он означает: «Это указатель на объект, владельцем которого я становлюсь». (Другие возможные значения этого атрибута рассматриваются в главе 20.)

Напомню, что объект, у которого нет ни одного владельца, уничтожается. При уничтожении объекту отправляется сообщению `dealloc`. (Все объекты наследуют метод `dealloc` от `NSObject`.) Переопределение метода `dealloc` позволит нам понаблюдать за уничтожением экземпляров `Asset`.

Чтобы было понятно, какой именно экземпляр `Asset` уничтожается, также необходимо реализовать другой метод `NSObject`, который называется `description`. Метод возвращает строку с содержательным описанием экземпляра класса. Для экземпляров `Asset` метод `description` будет возвращать строку, включающую значения `label` и `resaleValue` текущего экземпляра.

Откройте файл *Asset.m*. Синтезируйте методы доступа для переменных экземпляров, а затем переопределите `description` и `dealloc`.

```
#import "Asset.h"
@implementation Asset
@synthesize label, resaleValue;

- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@: $%d >",
                [self label], [self resaleValue]];
}

- (void)dealloc
{
    NSLog(@"deallocating %@", self);
}
@end
```

Обратите внимание на заполнитель %@в форматных строках приведенного выше кода. Этот заполнитель заменяется результатом отправки сообщения соответствующей переменной (которая должна содержать указатель на объект, получающий сообщение).

Попробуйте построить программу. чтобы посмотреть, не были ли допущены ошибки при вводе. Программу можно построить и без запуска - для этой цели используется комбинация клавиш Command+B. Данная возможность может быть полезна для проверки кода без запуска программы - например, если вы знаете, что программа к запуску еще не готова.

Также всегда рекомендуется строить программу после внесения изменений, чтобы любые синтаксические ошибки обнаруживались и исправлялись немедленно. Потом будет неочевидно, какие изменения отвечают за появление «новой» ошибки.

Добавление отношения «один ко многим» В Employee

А сейчас мы добавим в класс Employee отношение «один ко многим». Напомню, что в этом отношении задействован объект коллекции (например, массива) и объекты, содержащиеся в коллекции.

Два важных факта, которые необходимо знать о коллекциях и владельцах объектов:

- При добавлении объекта коллекция сохраняет указатель на объект, и у объекта появляется владелец.
- При удалении объекта коллекция освобождает указатель на объект, а у объекта теряется владелец.

Чтобы реализовать отношение «один ко многим». В Employee, нам понадобится новая переменная экземпляра, в которой будет храниться указатель на изменяемый массив ресурсов. Также понадобится пара дополнительных методов. Откройте файл *Employee.h* и добавьте их:

```
#import "Person.h"
@class Asset;
@interface Employee : Person
{
    int employeeID;
    NSMutableArray *assets;
}
@property int employeeID;
- (void)addAssetsObject:(Asset *)a; - (unsigned int)valueOfAssets;
@end
```

Обратите внимание на строку @class Asset;. В процессе чтения файла компилятор встречает имя класса Asset. Если у него нет никакой информации об этом классе, то компилятор выдает ошибку. Строка @class Asset; сообщает компилятору:

«Класс с именем `Asset` существует. Если он встретится в этом файле - без паники. И это все, что тебе пока необходимо знать».

Использование `@class` вместо `#import` предоставляет компилятору меньше информации, но ускоряет обработку этого конкретного файла. Мы можем использовать директиву `@class` в *Employee.h* и других заголовочных файлах, потому что для обработки файла с объявлениями компилятору не требуется много информации.

Теперь обратимся к файлу *Employee.m*. В отношениях типа «один ко многим». необходимо создать объект коллекции (массив в нашем случае), прежде чем что-либо сохранять в нем. Это можно сделать при создании исходного объекта (экземпляр `Employee`) или же не торопиться и подождать первой операции с коллекцией. В нашей программе будет использован второй вариант.

```
#import "Employee.h"
#import "Asset.h"
@implementation Employee
@synthesize employeeID;

- (void)addAssetsObject:(Asset *)a
{
    // Is assets nil?
    if (!assets) {
        // Create the array
        assets = [[NSMutableArray alloc] init];
    }
    [assets addObject:a];
}

- (unsigned int)valueOfAssets
{
    // Sum up the resale value of the assets
    unsigned int sum = 0;
    for (Asset *a in assets) {
        sum += [a resaleValue];
    }
return sum;
}

- (float)bodyMassIndex
{
    float normalBMI = [super bodyMassIndex];
    return normalBMI * 0.9;
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"<Employee %d: $%d in assets>",
            [self employeeID], [self valueOfAssets]];
}

- (void)dealloc
{
```

```
    NSLog(@"deallocating %@", self);
}
@end
```

Для обработки файла *Employee.m* компилятору необходима подробная информация о классе `Asset`, поэтому мы импортируем *Asset.h* вместо использования `@class`.

Также обратите внимание на переопределение `description` и `dealloc` для отслеживания удаления экземпляров `Employee`.

Постройте проект и убедитесь, что он не содержит ошибок.

Теперь необходимо создать объекты ресурсов и закрепить их за работниками. Отредактируйте файл *main.m*:

```
#import <Foundation/Foundation.h> #import "Employee.h"
#import "Asset.h"
int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // создание массива объектов Employee
        NSMutableArray *employees = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {

            // создание экземпляра Employee
            Employee *person = [[Employee alloc] init];

            // присваивание значений переменным экземпляра
            [person setWeightInKilos:90 + i];
            [person setHeightInMeters:1.8 - i/10.0];
            [person setEmployeeID:i];

            // включение экземпляра в массив employees
            [employees addObject:person];
        }

        // создание 10 экземпляров assets
        for (int i = 0; i < 10; i++) {

            // создание экземпляра asset
            Asset *asset = [[Asset alloc] init];

// присваивание метки
      NSString *currentLabel = [NSString stringWithFormat:@"Laptop %d", i];
            [asset setLabel:currentLabel];
            [asset setResaleValue:i * 17];

            // получение случайного числа от 0 до 9 включительно
            NSUInteger randomIndex = random() % [employees count];

            // получение соответствующего работника
            Employee *randomEmployee = [employees objectAtIndex:randomIndex];

            // назначение ресурса работнику
```

```
            [randomEmployee addAssetsObject:asset];
        }
        NSLog(@"Employees: %@", employees);
        NSLog(@"Giving up ownership of one employee");
        [employees removeObjectAtIndex:5];
        NSLog(@"Giving up ownership of array");
        employees = nil;
            }
      return 0;
}
```

Постройте и запустите программу Результат должен выглядеть примерно так:

```
Employees: (
    "<Employee 0: $0 in assets>",
    "<Employee 1: $153 in assets>",
    "<Employee 2: $119 in assets>",
    "<Employee 3: $68 in assets>",
    "<Employee 4: $0 in assets>",
    "<Employee 5: $136 in assets>",
    "<Employee 6: $119 in assets>",
    "<Employee 7: $34 in assets>",
    "<Employee 8: $0 in assets>",
    "<Employee 9: $136 in assets>"
)
Giving up ownership of one employee
deallocating <Employee 5: $136 in assets>
deallocating <Laptop 3: $51 >
deallocating <Laptop 5: $85  >
Giving up ownership of array
deallocating <Employee 0: $0 in assets>
deallocating <Employee 1: $153 in assets>
deallocating <Laptop 9: $153 >
...
deallocating <Employee 9: $136 in assets>
deallocing <Laptop 8: $136 >
```

При удалении из массива экземпляра `Employee` с номером 5 экземпляр уничтожается, потому что у него не остается владельцев. После этого уничтожаются его ресурсы, потому что у них тоже не осталось владельцев. (Поверьте на слово: метки, то есть экземпляры `NSString`, уничтоженных ресурсов тоже будут уничтожены, потому что у них тоже не остается владельцев.)

Когда `employees` задается значение `nil`, у массива не остается владельца. Соответственно, он уничтожается, что приводит к цепной реакции освобождений памяти и уничтожения объектов, так как все экземпляры `Employee` лишаются последнего владельца.

Удобно, правда? Как только объект становится ненужным, он уничтожается. Если ненужные объекты почему-либо не уничтожаются, говорят, что в программе возникает *утечка памяти*. Объекты задерживаются в памяти сверх положенного времени, приложение начинает поглощать все больше и больше памяти. В iOS операционная система в конечном итоге аварийно завершает приложение. В Mac OS X

утечка памяти снижает общую производительность системы, так как компьютер начинает тратить все больше времени на выгрузку данных из памяти на диск.

Упражнение

Используя класс `StockHolding` из предыдущего упражнения, напишите программу, которая создает экземпляр класса `Portfolio` (портфель акций) И заполняет его объектами `StockHolding` из упражнения предыдущей главы. Объект `Portfolio` должен уметь вычислять свою текущую стоимость.

20. Предотвращение утечки памяти

В программировании часто встречаются двусторонние отношения. Допустим, объект ресурса должен иметь возможность узнать, за кем из работников он закреплен. Давайте включим это отношение в нашу программу. Новая диаграмма объектов будет выглядеть как на рис. 20.1.

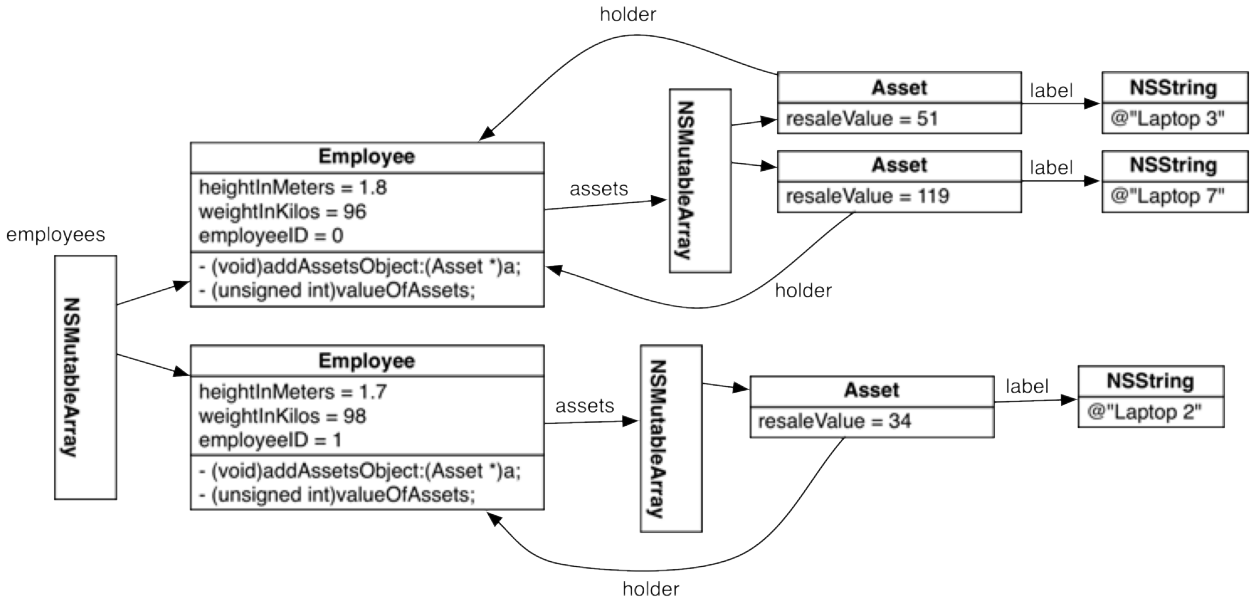


Рис. 20.1. Добавление отношения

С точки зрения архитектуры эта связь реализуется добавлением указателя от потомка (экземпляр `Asset`) к родителю (экземпляр `Employee`).

Добавьте в *Asset.h* переменную экземпляра для хранения указателя на объект работника:

```
#import <Foundation/Foundation.h>
@class Employee;
@interface Asset : NSObject
{
    NSString *label;
    Employee *holder;
    unsigned int resaleValue;
}

@property (strong) NSString *label;
@property (strong) Employee *holder;
@property unsigned int resaleValue;
@end
```

В файле *Asset.m* сгенерируйте методы доступа и расширьте метод `description`, чтобы он выводил информацию о держателе ресурса:

```
#import "Asset.h"
#import "Employee.h"
@implementation Asset
@synthesize label, resaleValue, holder;
- (NSString *)description
{
    // держатель ресурса отличен от nil?
    if ([self holder]) {
        return [NSString stringWithFormat:@"<%@: $%d, assigned to %@>",
                [self label], [self resaleValue], [self holder]];
    } else {
        return [NSString stringWithFormat:@"<%@: $%d unassigned>",
[self label], [self resaleValue]];
}
}
- (void)dealloc
{
    NSLog(@"deallocing %@", self);
}
@end
```

Возникает важный вопрос: как при совместном использовании классов `Asset` и `Employee` обеспечить согласованность двух отношений? Иначе говоря, ресурс должен присутствовать в массиве ресурсов работника в том и только в том случае, если работник является держателем ресурса. Возможны три варианта:

- Явно задать оба отношения:

```
[vicePresident addAssetsObject:townCar];
[townCar setHolder:vicePresident];
```

- В методе, задающем значение указателя потомка, добавить потомка в коллекцию родителя:

```
- (void)setHolder:(Employee *)e
  {
      holder = e;
      [e addAssetsObject:self];
  }
```

Такое решение применяется относительно редко.

- В методе, добавляющем потомка в коллекцию родителя, задать указатель потомка.

В этом упражнении мы воспользуемся последним вариантом. В файле *Employee.m* расширьте метод `addAssetsObject:`, значение чтобы он также присваивал `holder`:

```
- (void)addAssetsObject:(Asset *)a
{
    // Is assets nil?
    if (!assets) {
        // Create the array
        assets = [[NSMutableArray alloc] init];
    }
    [assets addObject:a];
    [a setHolder:self];
}
```

(Одна из моих любимых ошибок: два метода доступа автоматически вызывают друг друга. В программе возникает бесконечный цикл: `addAssetsObject:` вызывает `setHolder:`, который вызывает `addAssetsObject:`, который вызывает `setHolder:`, который...)

Постройте и запустите программу. Результат должен выглядеть примерно так:

```
Employees: (
    "<Employee 0: $0 in assets>",
    "<Employee 1: $153 in assets>",
    "<Employee 2: $119 in assets>",
    "<Employee 3: $68 in assets>",
    "<Employee 4: $0 in assets>",
    "<Employee 5: $136 in assets>",
    "<Employee 6: $119 in assets>",
    "<Employee 7: $34 in assets>",
    "<Employee 8: $0 in assets>",
    "<Employee 9: $136 in assets>"
)
Giving up ownership of one employee
Giving up ownership of array
deallocating <Employee 0: $0 in assets>
deallocating <Employee 4: $0 in assets>
deallocating <Employee 8: $0 in assets>
```

Обратите внимание: ни один из объектов работников, которым назначены ресурсы, не уничтожается. То же происходит и с ресурсами. Почему?

Циклическое владение

Ресурс является владельцем работника, работник является владельцем массива ресурсов, а массив ресурсов является владельцем ресурса. Подобные циклические отношения приводят к хранению «мусора» в памяти. Эти объекты должны быть уничтожены для освобождения памяти, но этого не происходит. Возникает ситуация *циклического владения* - очень распространенный источник утечки памяти.

Для выявления циклического владения в программах можно воспользоваться программой - профилировшиком Apple, которая называется Instruments. Программа запускается под управлением профилировщика для анализа событий, происходящих

«на заднем плане» вашего кода и системы. Однако наша программа запускается и завершает работу очень, очень быстро. Чтобы получить время для профилирования, включите в конец функции `main`() вызов `sleep`() с паузой на сотню секунд:

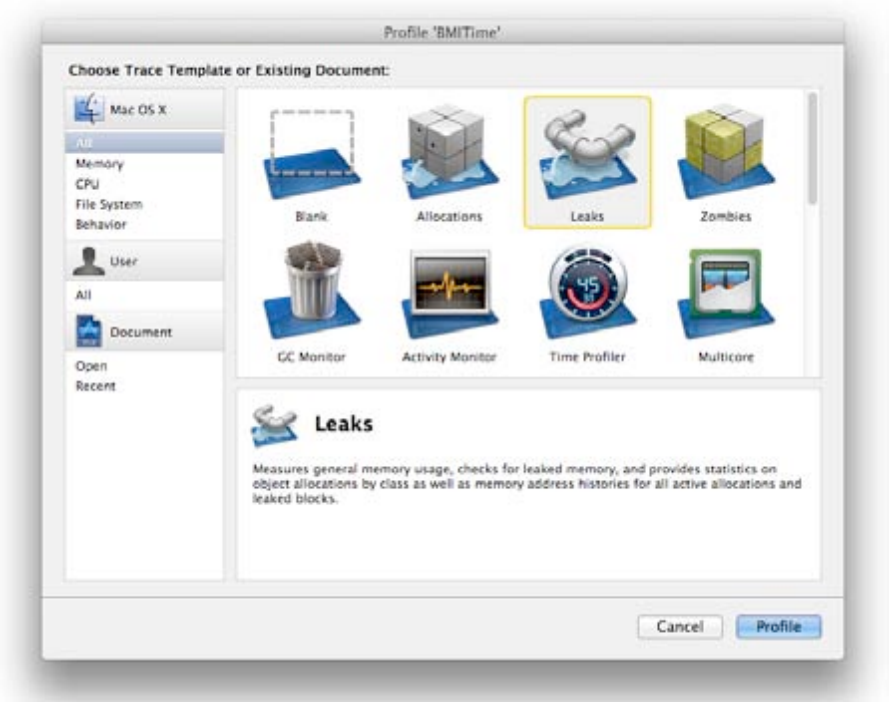


Рис 20.2. Выбор профилировщика

```
}
sleep(100);
return 0;
}
```

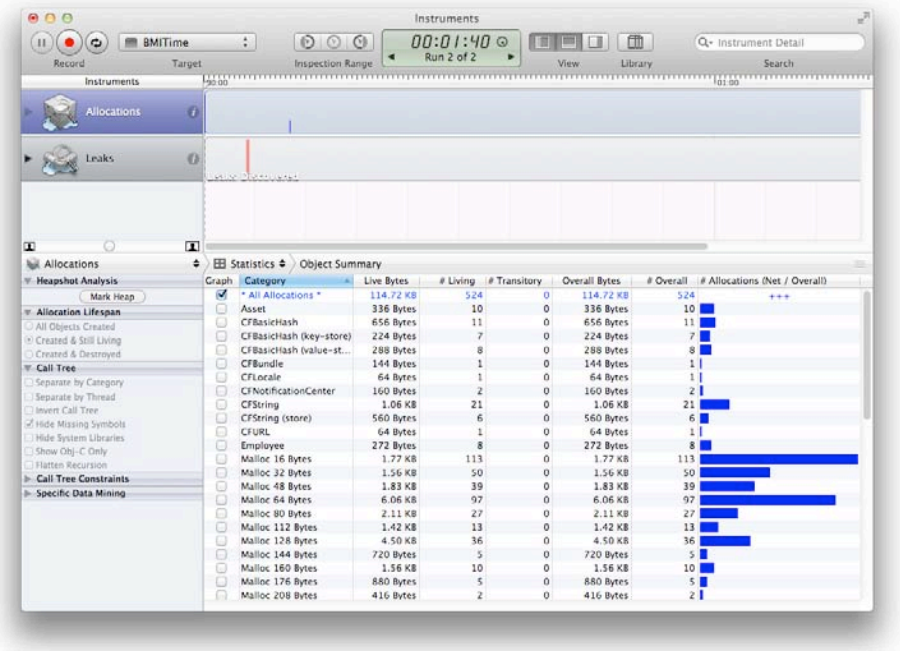


Рис. 20.3. Инструмент Allocations

Выполните в меню Xcode команду `Product→Profile`; запускается программа Instruments. Выберите в списке инструментов профилирования режим `Leaks`.

В ходе выполнения программы вы наблюдаете за текущим положением дел. На левой панели находятся два инструмента (рис. 20.3). Щелчок на инструменте `Allocations` отображает гистограмму памяти, выделенной в куче.

Например, из рисунка видно, что в куче остались 10 экземпляров `Asset`.

Чтобы найти в программе циклическое владение, переключитесь на инструмент `Leaks` и выберите представление `Cycles` из строки меню над таблицей. При выборе конкретного цикла отображается его диаграмма объектов (рис. 20.4).

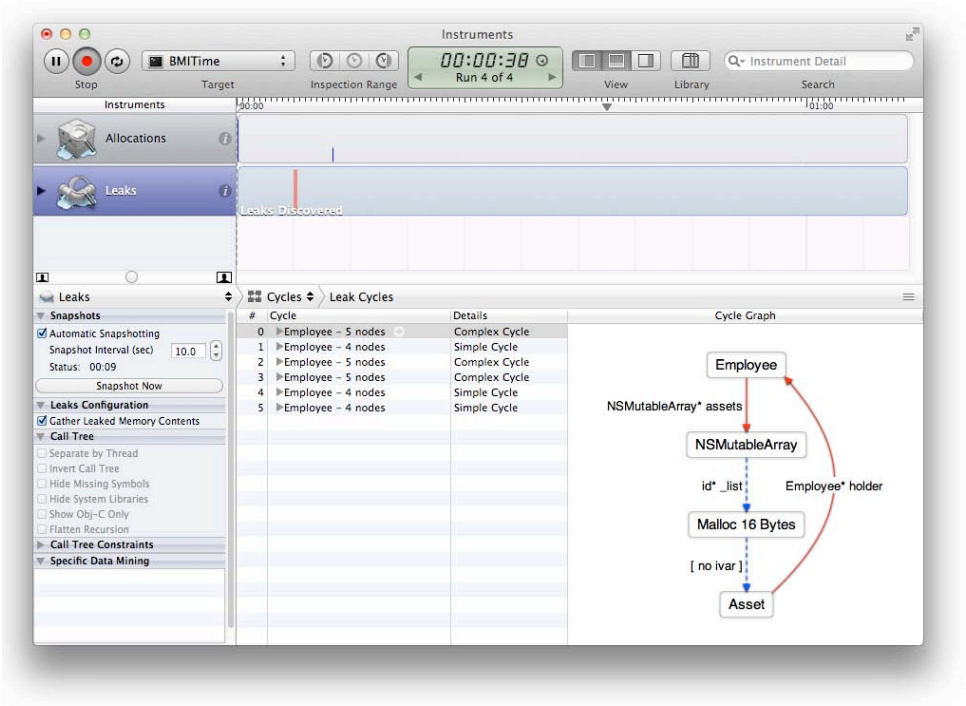


Рис. 20.4. Инструмент Leaks

Слабые ссылки

Как устранить проблему с циклическим владением? Используйте слабые ссылки. *Слабая ссылка* представляет собой указатель, не подразумевающий владения. В нашей программе для разрыва циклического владения ресурс не должен быть владельцем своего держателя. Отредактируйте файл *Asset.h*, чтобы преобразовать держателя в слабую ссылку:

```
#import <Foundation/Foundation.h>
@class Employee;
@interface Asset : NSObject
{
NSString *label;
__weak Employee *holder; unsigned int resaleValue;
}
@property (strong) NSString *label;
```

```
@property (weak) Employee *holder;
@property unsigned int resaleValue;
@end
```

Постройте и запустите программу. Теперь все объекты уничтожаются так, как положено.

В отношениях «родитель/потомок» для предотвращения циклов владения обычно применяется простое правило: родитель является владельцем потомка, но потомок не должен быть владельцем родителя.

Обнуление слабых ссылок

Чтобы понаблюдать за работой слабых ссылок, давайте включим в нашу картину еще один массив. Допустим, нам понадобился массив всех ресурсов - даже тех, которые не были назначены ни одному конкретному работнику.
Ресурсы добавляются в массив по мере создания. Включите несколько строк кода в main.m:

```
#import <Foundation/Foundation.h>
#import "Employee.h"
#import "Asset.h"
int main(int argc, const char * argv[])
{
    @autoreleasepool {

        // создание массива объектов Employee
        NSMutableArray *employees = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {

            // создание экземпляра Employee
            Employee *person = [[Employee alloc] init];

            // присваивание значений переменным экземпляров
            [person setWeightInKilos:90 + i];
            [person setHeightInMeters:1.8 - i/10.0];
            [person setEmployeeID:i];

            // включение работника в массив employees
            [employees addObject:person];
        }

        NSMutableArray *allAssets = [[NSMutableArray alloc] init];

        // создание 10 экземпляров assets
        for (int i = 0; i < 10; i++) {

          // создание экземпляра asset
            Asset *asset = [[Asset alloc] init];
```

```
            // присваивание метки
              NSString *currentLabel = [NSString stringWithFormat:@"Laptop %d",
i];
            [asset setLabel:currentLabel];
            [asset setResaleValue:i * 17];

            // получение случайного числа от 0 до 9 включительно
            NSUInteger randomIndex = random() % [employees count];

            // получение соответсвующего работника
            Employee *randomEmployee = [employees objectAtIndex:randomIndex];

            // назначение ресурса работнику
            [randomEmployee addAssetsObject:asset];

            [allAssets addObject:asset];
        }

        NSLog(@"Employees: %@", employees);

        NSLog(@"Giving up ownership of one employee");

        [employees removeObjectAtIndex:5];

        NSLog(@"allAssets: %@", allAssets);

        NSLog(@"Giving up ownership of arrays");

        allAssets = nil;
        employees = nil;
      }
      sleep(100);
      return 0;
}
```

Прежде чем строить и запускать программу, попробуйте предположить, как будет выглядеть вывод. Вы увидите содержимое массива `allAssets` - после уничтожения экземпляра `Employee №5`. В каком состоянии будут находиться его ресурсы на этот момент? Они теряют одного владельца (экземпляр `Employee №5`), но у них по-прежнему остается владелец `allAssets`, поэтому они не будут уничтожены.

Что происходит со значением `holder` для ресурсов, владельцем которых ранее был экземпляр `Employee №5`? При уничтожении объекта, на который указывает слабая ссылка, переменная указателя обнуляется, то есть ей присваивается `nil`. Соответственно ресурсы экземпляра `Employee №5` не будут уничтожены, а их переменной `holder` будут автоматически обнулены.

Теперь постройте и запустите про грамму и проверьте результат:

```
Employees: (
    "<Employee 0: $0 in assets>",
...
    "<Employee 9: $136 in assets>"
```

```
)
Giving up ownership of one employee
deallocating <Employee 5: $136 in assets>
allAssets: (
"<Laptop 0: $0, assigned to <Employee 3: $68 in assets>>",
"<Laptop 1: $17, assigned to <Employee 6: $119 in assets>>",
"<Laptop 2: $34, assigned to <Employee 7: $34 in assets>>",
"<Laptop 3: $51 unassigned>",
"<Laptop 4: $68, assigned to <Employee 3: $68 in assets>>",
"<Laptop 5: $85 unassigned>",
"<Laptop 6: $102, assigned to <Employee 6: $119 in assets>>",
"<Laptop 7: $119, assigned to <Employee 2: $119 in assets>>",
"<Laptop 8: $136, assigned to <Employee 9: $136 in assets>>",
"<Laptop 9: $153, assigned to <Employee 1: $153 in assets>>"
)
Giving up ownership of arrays
deallocing <Laptop 3: $51 unassigned>
...
deallocing <Laptop 8: $136 unassigned>
```

Для любознательных: ручной подсчет ссылок и история ARC

Как упоминалось в начале главы 19, до появления в Objective-C механизма автоматического подсчета ссылок ARC (Automatic Referencе Counting) использовался ручной подсчет ссылок, при котором владельцы изменялись только при явной отправке объекту сообщения, уменьшавшего или увеличивавшего счетчик ссылок.

```
[anObject release]; // anObject теряет владельца
[anObject retain]; // anObject получает владельца
```

Подобные вызовы в основном встречаются в методах доступа (`retain` - при создании нового значения, `release` - при освобождении старого значения) и в методах `dealloc` (при освобождении всех ранее захваченных объектов). Метод `setHolder`: класса `Asset` в этом случае выглядит примерно так:

```
- (void)setHolder:(Employee *)newEmp
{
    [newEmp retain];
    [holder release];
    holder = newEmp;
}
```

А вот как будет выглядеть метод `dealloc`:

```
- (void)dealloc
{
    [label release];
    [holder release];
    [super dealloc];
```

```
}
```

Как насчет метода `description`? Он создает и возвращает строку. Должен ли класс `Asset` заявлять на нее права владельца? Нет, это бессмысленно - ведь ресурс не удерживает созданную строку. Отправляя объекту сообщение `autorelease`, вы помечаете его для отправки `release` в будущем. До появления ARC метод description класса `Asset` выглядел бы так:

```
- (NSString *)description
{
    NSString *result = [[NSString alloc] initWithFormat:@"<%@: $%d >",
                                            [self label], [self resaleValue]];
    [result autorelease];
    return result;
}
```

Когда отправляется сообщение `release`? При сбросе текущего пула `autorelease`:

```
// Создание пула autorelease
NSAutoreleasePool *arp = [[NSAutoreleasePool alloc] init];
Asset *asset = [[Asset alloc] init];

NSString *d = [asset description];
// строка, на которую указывает d, находится в пуле autorelease

[arp drain]; // Строке отправляется сообщение release
```

ARC *использует* пул `autorelease` автоматически, но за создание и сброс пула отвечаете вы. С появлением ARC также появился новый синтаксис создания пула `autorelease`. Приведенный выше код теперь выглядит следующим образом:

```
// Создание пула autorelease
@autoreleasepool {
    Asset *asset = [[Asset alloc] init];

    NSString *d = [asset description];
    // строка, на которую указывает d, находится в пуле autorelease

}// Сброс пула
```

Правила подсчета ссылок

Существует стандартный набор соглашений по работе с памятью, которые соблюдаются всеми программистами Objective-C. Если вы используете ARC, то эти соглашения автоматически выполняются за вас.

В этих правилах словом «вы» я обозначаю «экземпляр класса, с которым вы работаете в настоящий момент». Такая точка зрения весьма полезна: представьте, что вы - объект, над которым вы работаете в настоящий момент. Таким образом, фраза «Если вы становитесь владельцем строки, она не будет уничтожена» в действительности означает «Если экземпляр класса, над которым вы работаете, становится владельцем строки, она не будет уничтожена».

Итак, правила (в круглых скобках описаны подробности реализации).

Если вы создаете объект методом, имя которого начинается с `alloc` или `new` или содержит `copy`, то вы становитесь его владельцем. (То есть предполагается, что у нового объекта счетчик ссылок равен 1 и он не находится в пуле `autorelease`.) Вы отвечаете за уничтожение объекта, когда он станет ненужным. Примеры часто используемых методов, которые делают вас владельцем созданного объекта: alloc (за вызовом которого всегда следует вызов `init`), copy и `mutableCopy`.

- Если объект был создан любыми другими средствами, вы не являегесь его владельцем (То есть следует полагать, что его счетчик ссылок равен 1 и он уже находится в пуле autorelease - а следовательно, обреченна уничтожение, если только у него не появится новый владелец до сброса пула autorelease.)

- Если вы не являетесь владельцем объекта, но хотите, чтобы он продолжая существовать, станьте его владельцем, отправив ему сообщение `retain` (что приводит к увеличению счетчика ссылок).

- Если вы являетесь владельцем объекта и он вам больше не нужен, отправьте ему сообщение `release` или autorelease. (Сообщение `release` уменьшает счетчик ссылок немедленно, autorelease приводит к отправке сообщения `release` при сбросе пула.)

- Пока у объекта остается хотя бы один владелец, он продолжит существовать, (Когда счетчик ссылок уменьшится до нуля, объекту отправляется сообщение `dealloc`.)

Чтобы лучше разобраться в управлении памятью, полезно рассматривать происходящее с локальной точки зрения. Классу `Asset` ничего не нужно знать о других объектах, для которых также существенно содержимое его метки. При условии, что экземпляр `Asset` назначает себя владельцем объектов, которые он хочет удержать в памяти, проблем не будет. Начинающие программисты иногда совершают ошибку, пытаясь самостоятельно вести учет ссылок на все объекты в приложении. Не делайте этого.

Если вы соблюдаете эти правила и всегда мыслите локальными категориями класса, вам никогда не придется беспокоиться о том, как остальные части приложении работают с объектом.

После знакомства с концепцией владения становится ясно, почему необходимо отправлять autorelease в методе description для строки: объект работника создает строку, но не хочет становиться ее владельцем.

21. Классы коллекций

В книге уже использовались два класса коллекций: `NSArray` и его субкласс `NSMutableArray`. Как вы уже знаете, массив содержит набор указателей на другие объекты. Указатели хранятся в определенном порядке, и для обращения к объектам коллекции можно использовать индекс (целочисленный номер). В этой главе мы ближе познакомимся с массивами, а также рассмотрим другие классы коллекций:

NSArray /NSMutableArray

При включении объекта в массив последний становится его владельцем. Когда объект удаляется из массива, массив исключается из числа владельцев. Откройте проект `BMITime` и посмотрите, как в программе используется массив employees. Если не обращать внимания на все остальное, основной код выглядит примерно так:

```
// Создание массива объектов Employee
NSMutableArray *employees = [[NSMutableArray alloc] init];
for (int i = 0; i < 10; i++) {

      // Создание объекта работника
      Employee *person = [[Employee alloc] init];

      // Включение объекта работника в массив employees
      [employees addObject:person];
}
[employees removeObjectAtIndex:5];
employees = nil;
```

Как правило, пустой изменяемый (`mutable`) массив создается методами `alloc/init` или методом класса `array`. Например, команда создания изменяемого массива может выглядеть так:

```
NSMutableArray *employees = [NSMutableArray array];
```

Метод `addObject`: добавляет объект в конец списка. С добавлением новых объектов массив автоматически увеличивается до размера, необходимого для их хранения.

Неизменяемые объекты

При создании экземпляра NSArray указываются все объекты, которые в нем должны храниться. Обычно это выглядит так:

```
NSArray *colors = [NSArray arrayWithObjects:@"Orange", @"Yellow", @"Green",
nil];
```

Значение nil в конце списка показывает, где метод должен прекратить добавление новых элементов. В нашем примере массив colors будет содержать три строки. (Если вы забудете nil, скорее всего, это приведет к аварийному завершению программы.)

Многих начинающих программистов удивляет сам факт существования NSArray. Зачем нужен список, содержимое которого нельзя изменить? Есть две причины:

- Вы не доверяете людям, с которыми работаете. Иначе говоря, вы хотите разрешить им просмотр массива, но не изменение его содержимого. Другое, более деликатное решение - предоставить им NSMutableArray, но выдать его NSArray. Например, представьте себе следующий метод:

```
// Возвращает массив из 30 нечетных чисел
- (NSArray *)odds
  {
    static NSMutableArray *odds = [[NSMutableArray alloc] init];
      int i = 1;
      while ([odds count] < 30) {
      [odds addObject:[NSNumber numberWithInt:i];
      i += 2;
      }
      return odds;
}
```

Каждый, кто вызывает этот метод, считает, что он возвращает неизменяемый массив NSArray. Если вызывающая сторона попытается добавить или удалить элементы из возвращенного объекта, компилятор выдает суровое предупреждение - хотя на самом деле возвращается экземпляр NSMutableArray.

- Другая причина для использования неизменяемых объектов - производительность; неизменяемые объекты не нужно копировать. При работе с из- меняемым объектом иногда приходится создавать приватную копию, чтобы быть уверенным в том, что другой код не сможет модифицировать объект незаметно для вас. Для неизменяемых объектов это не обязательно. Более того, метод copy класса NSMutableArray создает новую копию самого себя и возвращает указатель на новый массив, а метод copy класса NSArray не делает ничего - он просто возвращает указатель на самого себя.

По этим причинам неизменяемые объекты достаточно часто используются в программировании Objective-C. В Foundation неизменяемые экземпляры создаются многими классами: `NSArray`, `NSString`, `NSAttributedString`, `NSData`, `NSCharacterSet`, `NSDictionary`, `NSSet`, `NSIndexSet` и `NSURLRequest`. Все эти классы имеют изменяемые субклассы: `NSMutableArray`, `NSMutablestring`, `NSMutableAttributedString` и. т. д. Классы `NSDate` и `NSNumber` тоже относятся к неизменяемым, но не имеют изменяемых субклассов.

Сортировка

Существует несколько способов сортировки массива, но самый распространенный способ основан на использовании массива дескрипторов сортировки. Класс NSMutableArray содержит следующий метод:

```
- (void)sortUsingDescriptors:(NSArray *)sortDescriptors;
```

Аргумент содержит массив объектов `NSSortDescriptor`. Дескриптор сортировки содержит имя свойства объектов, содержащихся в массиве, и признак направления сортировки (по возрастанию или убыванию). Зачем передавать массив дескрипторов сортировки? Представьте, что вам потребовалось от- сортировать список клиентов по фамилиям в порядке возрастания. А если у двух клиентов окажутся одинаковые фамилии? Тогда приходится использовать более сложный критерий сортировки, например:. Отсортировать по фамилиям в порядке возрастания; если фамилии совпадают - то по именам в порядке возрастания, а если совпадают и имена, и фамилии - то по почтовым индексам».

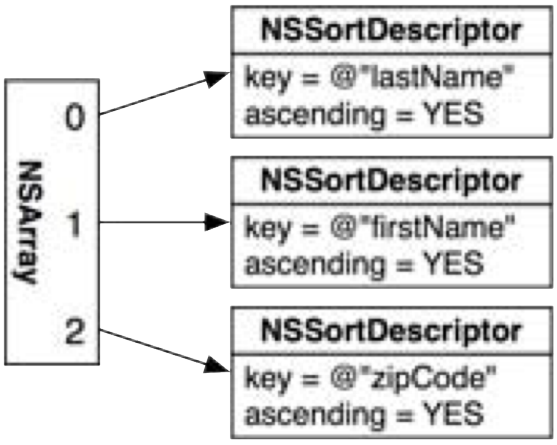


Рис. 21.1. Сортировка осуществляется сначала по lastName, затем по firstName и по zipCode

Свойство, по которому осуществляется сортировка, может быть любой переменной экземпляра или результатом выполнения любого метода объекта. Вернемся к проекту `BMITime` и посмотрим, как сортировка работает на практике. В

функции `main()`, непосредственно перед выводом массива `employees`, отсортируем его по значению `valueOfAssets`. Если за двумя работниками закреплены ресурсы равной стоимости, вторичная сортировка осуществляется по значению `employeeID`. Внесите изменения в файл *main.m*:

```
}
      NSSortDescriptor *voa = [NSSortDescriptor
sortDescriptorWithKey:@"valueOfAssets"
                                                        ascending:YES];
      NSSortDescriptor *ei = [NSSortDescriptor
sortDescriptorWithKey:@"employeeID"
                                                      ascending:YES];
      [employees sortUsingDescriptors:[NSArray arrayWithObjects:voa, ei, nil]];

NSLog(@"Employees: %@", employees);
```

Постройте и запустите программу Вы увидите, что список работников сортируется в правильном порядке:

```
Employees: (
    "<Employee 0: $0 in assets>",
    "<Employee 4: $0 in assets>",
    "<Employee 8: $0 in assets>",
    "<Employee 7: $34 in assets>",
    "<Employee 3: $68 in assets>",
    "<Employee 2: $119 in assets>",
    "<Employee 6: $119 in assets>",
    "<Employee 5: $136 in assets>",
    "<Employee 9: $136 in assets>",
    "<Employee 1: $153 in assets>"
```

Фильтрация

Класс `NSPredicate` предназначен для представления *предикатов* - условий, которые могут быть истинными, например: «Значение `employeeID` больше 75». Класс `NSMutableArray` содержит удобный метод для удаления всех объектов, для которых предикат не выполняется:

```
- (void)filterUsingPredicate:(NSPredicate *)predicate;
```

У NSArray имеется метод, который создает новый массив со всеми объектами, для которых предикат выполняется:

```
- (NSArray *)filteredArrayUsingPredicate:(NSPredicate *)predicate;
```

Допустим, мы хотим отобрать все ресурсы, выделенные работникам, у которых суммарная стоимость выделенных ресурсов превышает $70. Включите в конец *main.m* следующий код:

```
[employees removeObjectAtIndex:5];
NSLog(@"allAssets: %@", allAssets);
NSPredicate *predicate = [NSPredicate predicateWithFormat:
                                                    @"holder.valueOfAssets >
70"];
        NSArray *toBeReclaimed = [allAssets
filteredArrayUsingPredicate:predicate];
        NSLog(@"toBeReclaimed: %@", toBeReclaimed);
        toBeReclaimed = nil;
        NSLog(@"Giving up ownership of arrays");
        allAssets = nil;
        employees = nil;
    }
return 0;
}
```

Постройте и запустите программу Новая версия выводит следующий список ресурсов:

```
toBeReclaimed: (
    "<Laptop 1: $17, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 3: $51, assigned to <Employee 5: $136 in assets>>",
    "<Laptop 5: $85, assigned to <Employee 5: $136 in assets>>",
    "<Laptop 6: $102, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 8: $136, assigned to <Employee 9: $136 in assets>>",
    "<Laptop 9: $153, assigned to <Employee 1: $153 in assets>>"
)
```

Форматная строка в определении предиката может быть очень сложной. Если вам часто приходится фильтровать содержимое коллекций, обязательно прочитайте руководство «Predicate Programming Guide» из документации Apple.

NSSet/NSMutableSet

Множество представляет собой разновидность коллекции, в которой отсутствует понятие порядка следования объектов. Любой конкретный объект может входить в множество только один раз.

Множества, прежде всего, полезны для получения ответа на вопрос: «Присутствует ли объект в заданном наборе?» Например, в вашей программе может храниться набор URL-адресов, которые должны быть недоступны для детей. Прежде чем отображать веб-страницу для ребенка, следует проверить, не присутствует ли она среди URL-адресов множества. С множествами такая проверка выполняется очень быстро.

Выделенные работнику ресурсы не имеют определенного порядка хранения и не могут повторно входить В коллекцию ресурсов одного работника. Измените

программу чтобы вместо NSMutableArray для хранения отношений ресурсов использовался класс NSMutableSet.

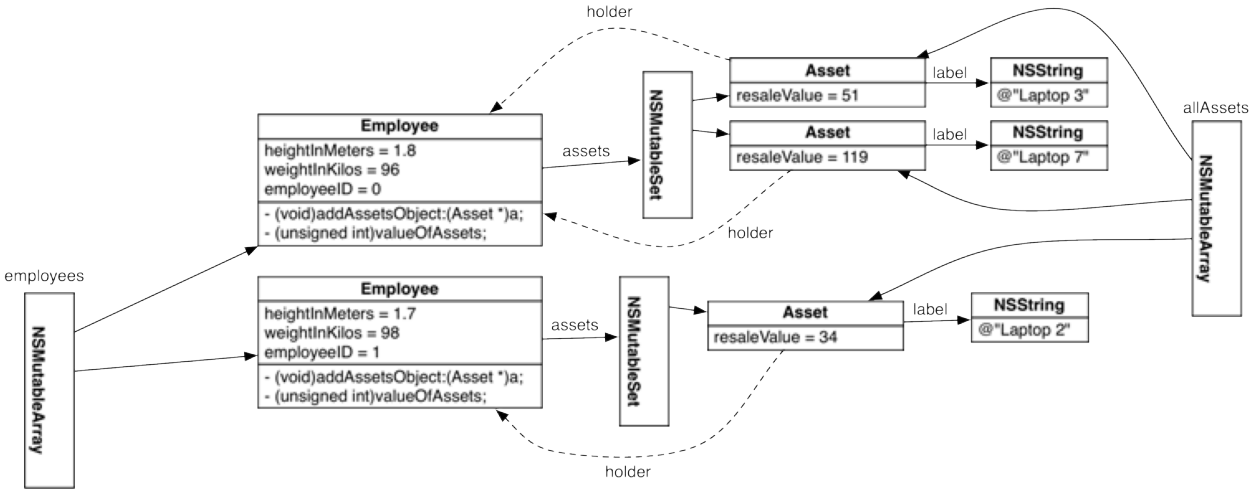


Рис. 21.2. Использование класса NSMutableSet для хранения ресурсов

В файле *Employee.h* измените объявление переменной:

```
#import "Person.h"
@class Asset;
@interface Employee : Person
{
    int employeeID;
NSMutableSet *assets; }
@property int employeeID;
- (void)addAssetsObject:(Asset *)a;
- (unsigned int)valueOfAssets;
@end
In Employee.m, create an instance of the correct class:
- (void)addAssetsObject:(Asset *)a
{
if (!assets) {
assets = [[NSMutableSet alloc] init];
    }
    [assets addObject:a];
    [a setHolder:self];
}
```

Постройте и запустите программу. Она должна работать так же, как прежде.

К объекту множества невозможно обратиться по индексу, потому что элементы множества не упорядочены. Вместо этого можно лишь спросить: «Входит ли объект в множество?» В классе NSSet имеется соответствующий метод:

```
- (BOOL)containsObject:(id)x;
```

При отправке этого сообщения множество перебирает свою коллекцию объектов в поисках объекта, равного x. Если такой объект будет найден, метод возвращает YES; в противном случае возвращается NO.

И тут мы сталкиваемся с достаточно глубоким вопросом: что означает «равного»? Класс `NSObject` определяет метод с именем `isEqual:`, проверяющий равенство двух объектов:

```
if ([myDoctor isEqual:yourTennisPartner]) {
   NSLog(@"my doctor is equal to your tennis partner");
}
```

`NSObject` предоставляет простую реализацию `isEqual:`. Она выглядит так:

```
- (BOOL)isEqual:(id)other
{
    return (self == other);
}
```

Таким образом, без переопределения метода `isEqual:` приведенный фрагмент эквивалентен следующему:

```
if (myDoctor == yourTennisPartner) {
   NSLog(@"my doctor is equal to your tennis partner");
}
```

Некоторые классы переопределяют `isEqual:`. Например, в `NSString` метод `isEqual:` переопределяется для сравнения символов строки. Для этих классов равенство не совпадает с идентичностью. Допустим, у нас имеются два указателя на строки:

```
NSString *x = ...
NSString *y = ...
```

Значения `x` и `y` идентичны, если они содержат одинаковые адреса. Значения `x` и `y` равны, если строки, на которые они указывают, содержат одинаковые символы, следующие в одном порядке.

Таким образом, идентичные объекты всегда равны, но равные объекты не всегда идентичны.

Важна ли эта разница? Да, важна. Например, класс `NSMutableArray` содержит два метода:

```
- (NSUInteger)indexOfObject:(id)anObject;
- (NSUInteger)indexOfObjectIdenticalTo:(id)anObject;
```

Первый метод перебирает коллекцию, проверяя для каждого объекта условие `isEqual: anObject`. Второй метод перебирает коллекцию с проверкой условия `== anObject`.

NSDictionary/NSMutableDictionary

Как вы уже знаете, элементы массивов идентифицируются числовыми индексами; легко и удобно обратиться к массиву: «Дай мне объект с индексом 10». Словари (ассоциативные массивы) индексируются строками; у них запрос на выборку элемента выглядит иначе: «Дай мне объект с ключом `favoriteColor`». Говоря точнее, словарь представляет собой коллекцию пар «ключ/значение». Ключ обычно является строкой, а значение может быть произвольным объектом. Пары «ключ/значение» не хранятся в каком-либо определенном порядке.

Создадим словарь руководителей фирмы. Ключом в нем будет должность, а значением - экземпляр `Employee`. Первый работник заносится в словарь с ключом @"CEO" (исполнительный директор), а второй - с ключом @"CTO" (технический директор). Внесите изменения в *main.m*: сначала программа создает и заполняет `NSMutableDictionary`, после чего выводит содержимое словаря. Наконец, указателю на словарь присваивается `nil`, чтобы было видно, что словарь был уничтожен.

```
// Создание массива объектов Employee
NSMutableArray *employees = [[NSMutableArray alloc] init];
//Создание словаря executives
        NSMutableDictionary *executives = [[NSMutableDictionary alloc] init];

        for (int i = 0; i < 10; i++) {

            // Создание экземпляра Employee
            Employee *person = [[Employee alloc] init];

            // Присваивание значений переменным экземпляра
            [person setWeightInKilos:90 + i];
            [person setHeightInMeters:1.8 - i/10.0];
            [person setEmployeeID:i];

            // Включение объекта Employee в массив employees
            [employees addObject:person];

            // Первый работник?
            if (i == 0) {
                [executives setObject:person forKey:@"CEO"];
            }

            // Второй работник?
            if (i == 1) {
                [executives setObject:person forKey:@"CTO"];
            }
}
...
        NSLog(@"allAssets: %@", allAssets);

        NSLog(@"executives: %@", executives);
        executives = nil;
```

```
        NSPredicate *predicate = [NSPredicate predicateWithFormat:
@"holder.valueOfAssets > 70"];
        NSArray *toBeReclaimed = [allAssets
filteredArrayUsingPredicate:predicate];
        NSLog(@"toBeReclaimed: %@", toBeReclaimed);
        toBeReclaimed = nil;
        NSLog(@"Giving up ownership of arrays");

        allAssets = nil;
        employees = nil;
    }
return 0;
}
```

Постройте и запустите программу Содержимое словаря выводится на консоль:

```
executives = {
    CEO = "<Employee 0: $0 in assets>";
    CTO = "<Employee 1: $153 in assets>";
}
```

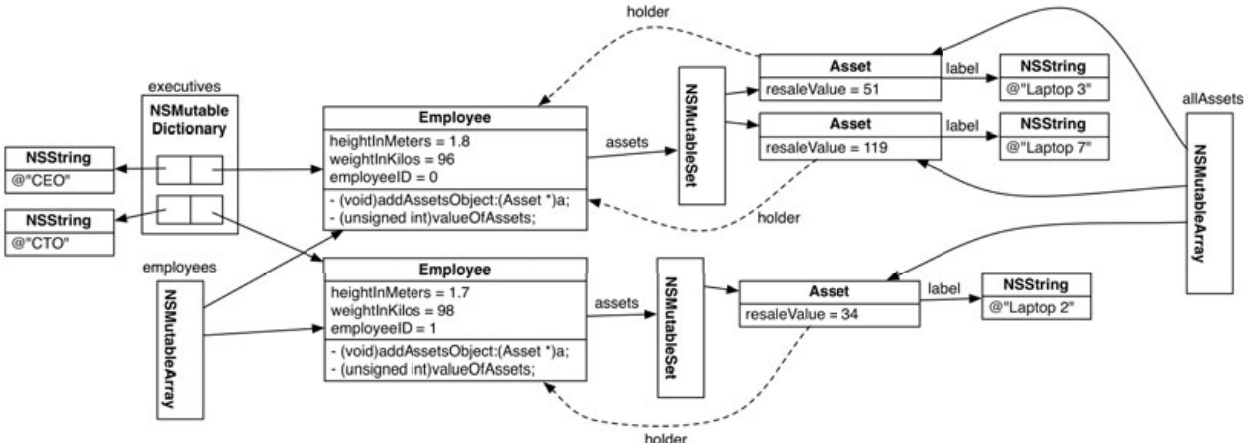


Рис. 21.3. Два экземпляра Employee в NSMutableDictionary

Ключи словаря уникальны. При попытке добавления второго объекта с уже существующим ключом происходит замена первой пары «ключ/значение».

```
// Создание словаря
NSMutableDictionary *friends = [NSMutableDictionary dictionary];
// Включение объекта с ключом "bestFriend"
[friends setObject:betty forKey:@"bestFriend"];
// Замена объекта
[friends setObject:jane forKey:@"bestFriend"];
// friends содержит только одну пару "ключ/значение" (bestFriend => jane)
```

Примитивные типы C

В коллекциях, рассматриваемых в этой главе, могут храниться только объекты. А если вам понадобится коллекция `float`, `int` или указателей на структуры? Примитивные типы C можно упаковать в объектную «обертку». Существует два

класса, предназначенных специально для этой цели. В NSNumber хранятся числовые типы C, а NSVa1ue может содержать указатель и некоторые типы структур.

Например, для сохранения в массиве чисел 4 и 5,6 следует использовать NSNumber:

```
NSMutableArray *numList = [[NSMutableArray alloc] init];
[numList addObject:[NSNumber numberWithInt:4]];
[numList addObject:[NSNumber numberWithFloat:5.6]];
```

Коллекции и nil

В классы коллекций, представленных в этой главе, не допускается хранение nil. А если вам понадобится поместить в коллекцию пустой элемент, «ничто»? Для этого используется специальный класс NSNull. Он существует только в одном экземпляре, и этот объект представляет «ничто». Пример:

```
NSMutableArray *hote1 = [[NSMutableArray a11oc] init];

// На первом этаже вестибюль
[hotel addObject:lobby];
// На втором этаже. бассейн
[hote1 addObject:pool];

// Третий этаж еще не застроен
[hote1 addObject:[NSNull null]];

// На четвертом этаже спальни
[hotel addObject:bedrooms];
```

Упражнение

Просмотрите справочные страницы классов NSArray, NSMutableArray, NSDictionary и NSMutableDictionary.

Вы будете постоянно использовать эти классы в своей работе.

22. Константы

Мы довольно подробно рассмотрели переменные, которые, как следует из самого названия, изменяют свои значения в ходе выполнения программы. Однако существуют и такие данные, значения которых *не* изменяются (как, например, математическая постоянная `pi`), Такие данные называются *константами*. В Objective-C существует два распространенных способа определения констант: `#define` и глобальные переменные.

Создайте в Xcode новую программу командной строки Foundation с именем `Constants`.

В стандартных библиотеках C константы определяются директивой препроцессора `#define`. Математическая часть стандартной библиотеки C объявляется В файле *math.h*. В частности, там определяется константа `M_PI`.

Используем ее в *main.m*:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // в литералах NSString, произвольные символы Юникода определяются символы \u
        NSLog(@"\u03c0 is %f", M_PI);
}
return 0;
}
```

Постройте и запустите программу. Она выводит следующий результат:

```
π is 3.141593
```

Чтобы просмотреть определение константы `M_PI`, щелкните на ней в редакторе, удерживая нажатой клавишу `Command`. Щелчок открывает следующую строку в *math.h*:

```
#define M_PI 3.14159265358979323846264338327950288
```

Директивы препроцессора

Компилирование файла с кодом C, C++ или Objective-C осуществляется за два прохода. Сначала файл обрабатывается препроцессором, после чего результат обработки передается компилятору. Директивы препроцессора начинаются с символа `#`; три наиболее часто используемых директивы - `#include`, `#import` и `#define`.

#include и #import

Директивы `#include` и `#import` делают практически одно и то же: они приказывают препроцессору прочитать файл и включить его в выходные данные. Обычно они используются для включения файлов объявлений (с расширением *.h*); объявления нужны компилятору для более полного понимания компилируемого кода.

Чем `#include` отличается от `#import`? Директива `#import` гарантирует, что препроцессор включит файл только один раз. Директива `#include` позволяет многократно включить один файл. Программисты C чаще используют `#inc1ude`, а программисты Objective-C предпочитают `#import`.

Имя импортируемого файла может быть заключено в кавычки или угловые скобки. Кавычки означают, что заголовочный файл находится в каталоге проекта. Угловые скобки означают, что заголовочный файл находится в одном из стандартных каталогов, известных препроцессору. (Например, `<math.h>` находится в каталоге / Developer/SDKs/MacOSX10.7.sdk/usr/include/math.h.) Два примера директивы `#import`:

```
// Включение заголовков, написанных мной для операций Pet Store
#import "PetStore.h"
// Включение заголовкок для библиотек OpenLDAP
#import <ldap.h>
```

Когда-то в проектах каждый файл C программным кодом начинался с включения набора заголовков. Это приводило к загромождению начала файла и замедлению компиляции. Чтобы программисту было проще работать, а программы быстрее компилировались, в большинстве проектов Xcode имеется файл с перечнем заголовков, которые должны быть предварительно откомпилированы и включены в каждый файл. В проекте `Constants` этот файл называется *Constants-Prefix.pch*.

Итак, как же происходит включение константы из *math.h* во время компилирования *main.m*? Наш файл *main.m* содержит следующую строку:

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

Файл *Foundation.h* содержит строку:

```
#include <CoreFoundation/CoreFoundation.h>
```

Файл *CoreFoundation.h* содержит строку:

```
#include <math.h>
```

#define

Директива `#define` приказывает препропессору: «Каждый раз, когда ты встречаешь `X`, замени его на `У` еще до того, как компилятор увидит код». Еще раз взгляните на строку из *math.h*:

```
#define M_PI            3.14159265358979323846264338327950288
```

В директиве `#define` две части (символическое имя и его замена) разделяются пробелами.

Вообще говоря, с помощью директивы `#define` можно создать некое подобие функции. Включите в *main.m* команду вывода большего из двух чисел:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%d is larger", MAX(10, 12));
}
return 0;
}
```

`MAX` - не функция, а `#define`. Простейшая версия `MAX` на языке C выглядит так;

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

Итак, к тому моменту, когда компилятор увидит добавленную строку, она примет следующий вид:

```
NSLog(@"%d is larger", ((10) > (12) ? (10) : (12)));
```

Используя `#define` для имитации функций (вместо простой замены значений), вы создаете *макроопределение*, или *макрос*.

Глобальные переменные

Вместо `#define` программисты Objective-C обычно используют для хранения постоянных значений глобальные переменные.

Давайте дополним нашу программу. Класс с именем `NSLocale` содержит *локальный контекст*, то есть описание стандартов, действующих в определенном географическом месте. Например, вы можете получить экземпляр текущего локального контекста для пользователя, а затем обращаться к нему за информацией.

Например, сели вы хотите узнать, какая денежная единица используется в локальном контексте пользователя, это делается так:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%d is larger", MAX(10, 12));
        NSLocale *here = [NSLocale currentLocale];
        NSString *currency = [here objectForKey:@"currency"];
        NSLog(@"Money is %@", currency);
}
return 0;
}
```

Постройте и запустите программу. В зависимости от того, где вы находитесь, результат может выглядеть примерно так:

```
Money is USD
```

Но если при вводе ключа будет допущена ошибка (например,`@"Kuruncy"`), вы не получите никаких данных. Для предотвращения подобных проблем в библиотеке Foundation определяется глобальная переменная с именем `NSLoca1eCurrencyCode`. Вряд ли ее проще вводить, но зато если при вводе будет допущена ошибка, компилятор ее заметит. Кроме того, автозавершение кода в Xcode нормально работает для глобальной переменной, но не для строки `@"currency"`. Измените код так, чтобы в нем использовалась константа:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%d is larger", MAX(10, 12));

        NSLocale *here = [NSLocale currentLocale];
        NSString *currency = [here objectForKey:NSLocaleCurrencyCode];
        NSLog(@"Money is %@", currency);
        }
        return 0;
}
```

При написании масса `NSLoca1e` эта глобальная переменная встречалась в двух местах. В файле *NSLocale.h* переменная была объявлена примерно так

```
extern NSString const *NSLocaleCurrencyCode;
```

Ключевое слово `const` говорит о том, что указатель не будет изменяться в течение всего жизненного цикла программы. Ключевое слово `extern` означает: «Я

гарантирую, что это существует, но оно определяется в другом файле». И действительно, файл *NSLocale.m* содержит строку следующего вида:

```
NSString const *NSLocaleCurrencyCode = @"currency";
```

enum

Часто в программе требуется определить набор констант. Допустим, вы программируете блендер с пятью рабочими режимами. Ваш класс `B1ender` содержит метод `setSpeed:`. Было бы хорошо, если бы тип аргумента указывал, что допустимыми являются только пять конкретных значений. Для этого в программе определяется *перечисляемый тип*, или *перечисление*:

```
enum BlenderSpeed {
    BlenderSpeedStir = 1,
    BlenderSpeedChop = 2,
    BlenderSpeedLiquify = 5,
    BlenderSpeedPulse = 9,
    BlenderSpeedIceCrush = 15
};
@interface Blender : NSObject
{
    // пять допустимых значений скорости
    enum BlenderSpeed speed;
}
// setSpeed: получает одно из пяти допустимых значений
- (void)setSpeed:(enum BlenderSpeed)x;
@end
```

Чтобы разработчикам не приходилось постоянно вводить `enum BlenderSpeed`, они часто определяют сокращенную запись с использованием

```
typedef enum  {
    BlenderSpeedStir = 1,
    BlenderSpeedChop = 2,
    BlenderSpeedLiquify = 5,
    BlenderSpeedPulse = 9,
    BlenderSpeedIceCrush = 15
} BlenderSpeed;
@interface Blender : NSObject
{
    // пять допустимых значений скорости
    BlenderSpeed speed;
}
// setSpeed: получает одно из пяти допустимых значений
- (void)setSpeed:(BlenderSpeed)x;
@end
```

Часто для программиста совершенно неважно, какими числами представлены пять скоростей - лишь бы они отличались друг от друга. Значения элементов перечисления можно не указывать, в этом случае компилятор сгенерирует их автоматически:

```
typedef enum  {
    BlenderSpeedStir,
    BlenderSpeedChop,
    BlenderSpeedLiquify,
    BlenderSpeedPulse,
    BlenderSpeedIceCrush
} BlenderSpeed;
```

#define и глобальные переменные

Если для определения констант можно использовать как `#define`, так и глобальные переменные (включая `enum`), почему программисты Objective-C обычно используют глобальные переменные? Иногда глобальные переменные повышают быстродействие программы. Например, при работе с глобальной переменной для сравнения строк можно использовать == вместо `isEqual:` (а математическая операция выполняется быстрее отправки сообщения). Кроме того, с глобальными переменными удобнее работать в отладчике.

Используйте для констант глобальные переменные и `enum`, а не `#define`.

23. Запись в файлы с использованием NSString и NSData

Библиотека Foundation предоставляет разработчику несколько простых механизмов чтения и записи в файлы. Некоторые из них будут рассмотрены в этой главе.

Запись NSString в файл

Начнем с сохранения в файле содержимого `NSString`. При записи строки в файл необходимо указать используемую кодировку, то есть способ представления каждого символа в виде массива байтов. В кодировке `ASCII` буква 'A' хранится в виде `01000001`, а в кодировке `UTF-16` та же буква 'A' представляется в виде `0000000001000001`.

Библиотека Foundation поддерживает около 20 видов кодировки строк. Кодировка `UTF` способна справиться даже с самыми экзотическими алфавитами. Она существует в двух разновидностях: в `UTF-16` каждый символ представляется двумя и более байтами, а в `UTF -8` для первых `128` символов `ASCII` используется один байт, а для всех остальных символов - два и более байта. Для многих практических задач `UTF-8` оказывается достаточно ..

Создайте новый проект: программу командной строки `Foundation` с именем `Stringz`. В функции `main`() методами класса `NSString` мы создадим строку и запишем ее в файловую систему:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])    {
    @autoreleasepool {
        NSMutableString *str = [[NSMutableString alloc] init];
        for (int i = 0; i < 10; i++) {
            [str appendString:@"Aaron is cool!\n"];
        }
        [str writeToFile:@"/tmp/cool.txt"
              atomically:YES
                encoding:NSUTF8StringEncoding
                   error:NULL];
        NSLog(@"done writing /tmp/cool.txt");
}
return 0;
}
```

Программа создает текстовый файл, который можно читать и редактировать в любом текстовом редакторе. Строка `/tmp/cool.txt` содержит путь к файлу. Пути могут быть абсолютными или относительными; абсолютный путь начинается с символа /, представляющего корневой уровень файловой системы, а относительные пути начинаются с рабочего каталога программы. Относительный путь не может начинаться с `/`. В программировании Objective-C почти всегда используются абсолютные пути, потому что мы не знаем рабочий каталог программы. (Чтобы найти каталог `/tmp` в `Finder`, используйте команду меню `Go -> Go to Folder`.)

NSError

Попытка записи строки в файл может оказаться неудачной по многим причинам. Например, пользователю может быть запрещена запись в каталог, в котором находится файл. Или каталог может вообще не существовать. Или в файловой системе не найдется свободного места. для подобных ситуаций, когда завершение операции невозможно, метод должен каким-то образом вернуть описание ошибки кроме логического признака успеха или неудачи.

Как говорилось в главе 9, если функция должна вернуть какую-либо информацию помимо своего возвращаемого значения, следует использовать механизм передачи по ссылке. Функции (или методу) передается ссылка на переменную, в которой сохраняется нужное значение или выполняются другие операции. Ссылка представляет собой адрес переменной в памяти. Для обработки ошибок многим методам передается по ссылке указатель на `NSError`. Включите обработку ошибок в приведенный выше пример:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
@autore1easepoo1 {

      NSMutableString *str = [[NSMutableString a11oc] init];
      for (int i =0; i <10; i++) {
      [str appendString:@"Aaron is coo1!\n"];
      }

// 3десь указатель на объект NSError объявляется без создания
// экземпляра. 3кземпляр NSError будет создан только при возникновении ошибки.
NSError *error = nil;

// Передача по ссылке указателя NSError методу NSString
B00L success = [str writeToFile:@"/tmp/cool.txt"
                            atomically:YES
                            encoding:NSUTF8StringEncoding
                            error:&error]:
// Проверка полученного флага и вывод информации из NSError в случае ошибки
записи
if (succes) { } e1se {
NSLog(@"done writing/tmp/cool.txt");
}
NSLog(@"writing/tmp/cool.txt failed: %@", [error localizedDescription]);
```

```
}
return 0;
}
```

Постройте и запустите программу. Измените код таким образом, чтобы методу записи передавался несуществующий путь к файлу - например, `@"/too/darned/bad.txt"`. Программа должна вывести информативное сообщение об ошибке.

Обратите внимание: в этом коде мы объявляем указатель на экземпляр `NSError`, но не создаем экземпляр `NSError`, который с этим указателем связывается.

Почему? Не стоит создавать объект ошибки, если ошибки не было. Если произойдет ошибка, то сообщение `writeToFile:atomically:encoding:error:` обеспечит создание нового экземпляра `NSError` с последующим изменением указателя error, чтобы он указывал на созданный объект. Далее вы сможете по указателю error получить расширенную информацию об ошибке.

Условное создание `NSError` требует передачи ссылки на error (`&error`), потому что объект, который можно было бы передать, еще не существует. Однако в отличие от передачи по ссылке, которая использовалась в главе 9 для примитивных переменных C, в данном случае передается адрес переменной-указателя. Фактически мы передаем адрес другого адреса (который может стать адресом объекта `NSError`).

Возвращаясь к нашему примеру со шпионами из главы 9, вы можете приказать своему подопечному: «Если что-то пойдет не так, составь полный отчет (который в тайнике не поместится) и заложи его в книгу в библиотеке. А чтобы я узнал, где спрятан отчет, положи в трубу библиотечный шифр книги». То есть вы сообщаете шпиону место, в котором он может спрятать адрес созданного им отчета об ошибке.

Заглянем вовнутрь класса `NSString`, в котором объявляется `writeToFile:atomically:encoding:error:`:

```
- (BOOL)writeToFile:(NSString *)path
         atomically:(BOOL)useAuxiliaryFile
           encoding:(NSStringEncoding)enc
              error:(NSError **)error
```

Обратите внимание на двойную звездочку. При вызове этого метода мы передаем *указатель на указатель* на экземпляр `NSError`.

Методы, передающие `NSError` по ссылке, всегда возвращают значение, по которому можно определить, произошла ошибка или нет. Например, при наличии ошибки этот метод возвращает `NO`. Не пытайтесь обратиться к `NSError`, если возвращаемое значение не указывает на наличие ошибки; если объект `NSError` не существует, то попытка обращения к нему приведет к сбою программы.

Чтение файлов с использованием NSString

Чтение содержимого файла в строку выполняется аналогичным образом:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])    {
    @autoreleasepool {
        NSError *error = nil;
        NSString *str = [[NSString alloc] initWithContentsOfFile:@"/etc/
resolv.conf"

encoding:NSASCIIStringEncoding
                                                        error:&error];
        if (!str) {
            NSLog(@"read failed: %@", [error localizedDescription]);
        } else {
            NSLog(@"resolv.conf looks like this: %@", str);
}
}
return 0;
}
```

Запись объекта NSData в файл

Объект `NSData` представляет данные. Например, при загрузке данных по URL-адресу вы получаете экземпляр `NSData`, которому можно приказать записать себя в файл. Создайте новую программу командной строки `Foundation` с именем `ImageFetch`; эта программа будет загружать изображение с сайта Google в экземпляр `NSData`. Затем прикажите `NSData` записать свой буфер данных в файл:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSURL *url = [NSURL URLWithString:
                          @"http://www.google.com/images/logos/ps_logo2.png"];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        NSError *error = nil;
        NSData *data = [NSURLConnection sendSynchronousRequest:request
                                             returningResponse:NULL
                                                         error:&error];
        if (!data) {
            NSLog(@"fetch failed: %@", [error localizedDescription]);
            return 1;
        }
        NSLog(@"The file is %lu bytes", [data length]);
        BOOL written = [data writeToFile:@"/tmp/google.png"
                                 options:0
                                   error:&error];
        if (!written) {
            NSLog(@"write failed: %@", [error localizedDescription]);
            return 1;
}
        NSLog(@"Success!");
```

```
}
return 0;
}
```

Постройте и запустите программу откройте созданный графический файл в Preview.

Метод `writeToFile:options:error:` получает число, обозначающее настройки процесса сохранения. Чаще всего используется значение `NSDataWritingAtomic`. Допустим, изображение уже было загружено, а сейчас вы загружаете его заново и заменяете новой версией. Во время записи нового изображения внезапно отключается питание. Наполовину записанный файл бесполезен. В тех случаях, когда наполовину записанный файл хуже, чем никакого, можно объявить операцию атомарной. Добавьте соответствующий параметр:

```
NSLog(@"The file is %lu bytes", [data length]);
BOOL written = [data writeToFile:@"/tmp/google.png" options:NSDataWritingAtomic
                                 error:&error];
if (!written) {
        NSLog(@"write failed: %@", [error localizedDescription]);
        return 1;
}
```

Данные записываются во временный файл, которому после завершения записи присваивается правильное имя. Таким образом вы получаете либо весь файл, либо ничего.

Чтение NSData из файла

Экземпляр `NSData` также можно создать на основе содержимого файла. Включите в программу две строки:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSURL *url = [NSURL URLWithString: @"http://www.google.com/images/
logos/ps_logo2.png"];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        NSError *error;

        // этот метод блокируется, пока все данные не будут извлечены
        NSData *data = [NSURLConnection sendSynchronousRequest:request
                                             returningResponse:NULL
                                                         error:&error];
        if (!data) {
            NSLog(@"fetch failed: %@", [error localizedDescription]);
            return 1;
        }
        NSLog(@"The file is %lu bytes", [data length]);
```

```
        BOOL written = [data writeToFile:@"/tmp/google.png"
                                 options:NSDataWritingAtomic
                                   error:&error];
        if (!written) {
            NSLog(@"write failed: %@", [error localizedDescription]);
            return 1;
}
        NSLog(@"Success!");
        NSData *readData = [NSData dataWithContentsOfFile:@"/tmp/google.png"];
        NSLog(@"The file read from the disk has %lu bytes", [readData length]);
}
return 0;
}
```

Постройте и запустите программу.

24. Обратный вызов

До настоящего момента всем распоряжался ваш код. Он отправлял сообщения стандартным объектам `Foundation` (скажем, экземплярам `NSString` и `NSArray`) и отдавал им приказы. До настоящего момента ваши программы выполнялись и завершались за считанные миллисекунды.

В реальном мире приложения могут выполняться часами, а объекты подчиняются потоку событий от пользователя, системных часов, сети и т. д.

В реальном приложении должен существовать объект, который ожидает таких событий, как перемещения мыши, прикосновения к экрану, таймеры и сетевые операции. В Mac OS X·и iOS таким объектом является экземпляр цикла событий `NSRunLoop`. Обычно он сидит и ждет, а когда в системе что-то происходит - отправляет сообщение другому объекту.

Ситуация, в которой код ожидает внешних событий, называется *обратным вызовом*. Для программистов Objective-C существуют три основных формы обратного вызова. (Так как приведенные концепции имеют общий характер, я буду обозначать произошедшее «нечто» буквой «`X`», а подробности будут приведены ниже.)

- *Приемник/действие*: перед началом ожидания вы говорите: «Когда произойдет `X`, отправь это конкретное сообщение этому конкретному объекту». Объект, получающий сообщение, называется приемником. Селектор сообщения и есть действие.

- Вспомогательные объекты: перед началом ожидания вы говорите: «Здесь имеется вспомогательный объект, поддерживающий твой протокол. Когда что-нибудь произойдет, отправляй ему сообщения». (Протоколы подробно рассматриваются в главе 25.) Вспомогательные объекты часто называются *делегатами* или *источниками данных*.

- *Оповещения*: имеется объект, называемый *центром оповещений*. Перед началом ожидания вы говорите ему: «Этот объект ожидает таких-то оповещений. При поступлении одного из них отправь объекту это сообщение». Когда происходит `X`, объект отправляет оповещение центру оповещений, а последний пересылает его вашему объекту.

В этой главе мы реализуем все три типа обратных вызовов, а также разберемся, в каких обстоятельствах применяется каждый из них.

Модель «приемник/действие»

В частности, механизм «приемник/действие. используется таймерами. При создании таймера указывается задержка, приемник и действие. После истечения заданной задержки таймер отправляет сообщение действия своему приемнику.

Мы создадим программу с циклом событий и таймером. Каждые две секунды таймер будет отправлять своему приемнику сообщение действия. Мы создадим класс, экземпляр которого и будет использоваться в качестве приемника.

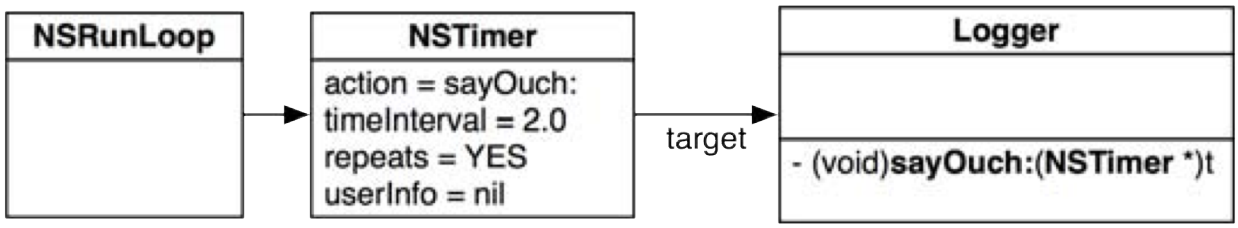


Рис. 24.1. Logger является приемником для NSTimer

Создайте в Xcode новый проект: программу командной строки `Foundation` с именем `Callbacks`. Начнем с простого создания работоспособного цикла событий. Внесите изменения в *main.m*:

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        [[NSRunLoop currentRunLoop] run];
}
return 0;
}
```

Постройте и запустите программу. Обратите внимание: метод `run` никогда не возвращает управление. Он выполняется в бесконечном цикле, ожидая, пока что-нибудь произойдет. Чтобы его прервать, необходимо завершить программу. (Выберите команду `Product → Stop`.)

Теперь мы создадим класс, который будет использоваться в качестве приемника таймера. Создайте новый файл: класс Objective-C с именем `Logger`, который является субклассом `NSObject`. (Шаблон класса вызывается командой `File→New→NewFile...`) В файле *Logger.h* объявите метод действия:

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

@interface Logger : NSObject
- (void)sayOuch:(NSTimer *)t;
@end
```

Метод действия получает один аргумент - объект, отправляющий сообщение действия. В нашем случае это объект таймера.

Реализуйте в *Logger.m* простой метод `sayOuch:`:

```
#import "Logger.h"
@implementation Logger
- (void)sayOuch:(NSTimer *)t
{
    NSLog(@"Ouch!");
}
@end
```

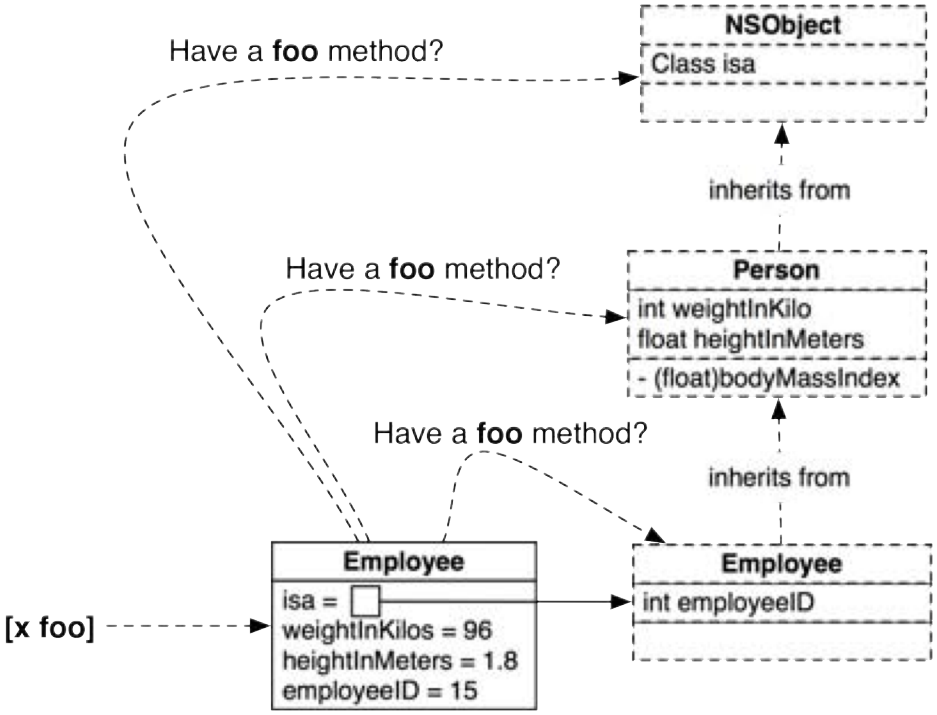


Рис. 24.2. Поиск метода с указанным именем

На этой стадии необходимо слегка отвлечься и поговорить о селекторах. Вспомните, что при отправке сообщения объекту класс объекта проверяет, содержит ли он метод с указаным именем. Поиск переходит вверх по цепочке наследования, пока какой-либо класс не ответит: «Да,у меня есть метод с таким именем».

Естественно, такая проверка должна выполняться очень, очень быстро. При использовании фактического имени метода (которое может быть очень длинным) поиск будет относительно медленным. Для ускорения проверки компилятор присваивает каждому имени метода в программе уникальный идентификатор. На стадии выполнения вместо имени метода используется его идентификатор.

Уникальное число, представляющее конкретное имя метода, называется *селектором*. Чтобы создать таймер, отправляющий сообщение `sayOuch:` классу `Logger`, необходимо приказать компилятору определить его селектор. Для решения этой задачи используется директива компилятора `@selector`.

В файле main.m создайте экземпляр `Logger` и сделайте его приемником `NSTimer`. В качестве действия укажите селектор `sayOuch:`.

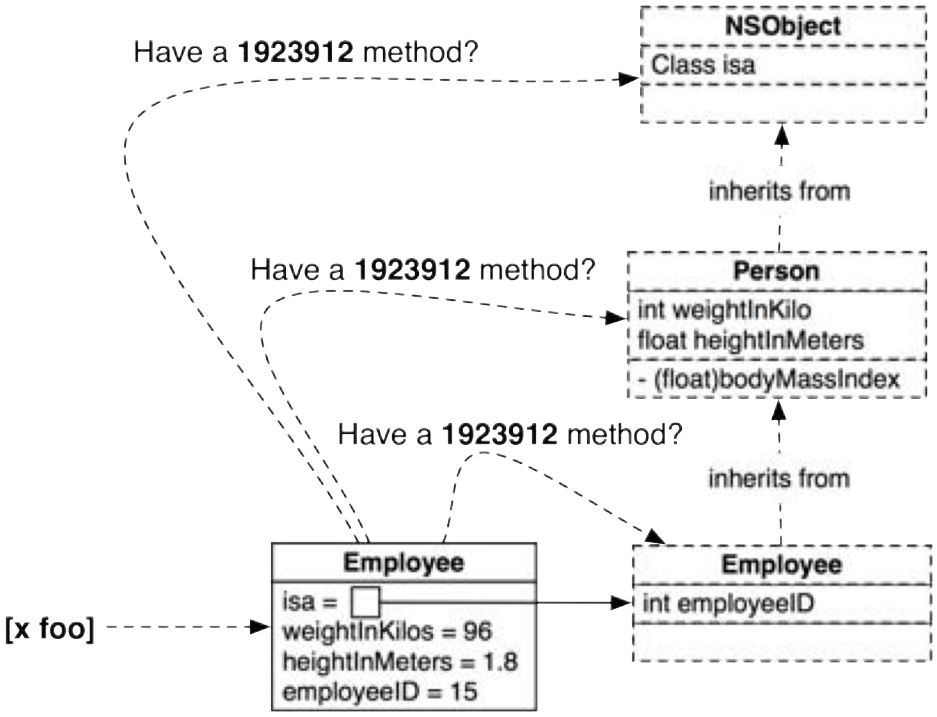


Рис. 24.3. Что происходит в действительности

```
#import <Foundation/Foundation.h>
#import "Logger.h"
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        Logger *logger = [[Logger alloc] init];
        NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:2.0
                                                          target:logger

selector:@selector(sayOuch:)
                                                        userInfo:nil
                                                         repeats:YES];
        [[NSRunLoop currentRunLoop] run];
}
return 0;
}
```

Постройте и запустите программу. (Компилятор выдает сообщение о не используемой переменной, пока не обращайте на него внимания.) Сообщение `Ouch!` должно выводиться на консоль каждые две секунды.

Обратите внимание: двоеточие является частью селектора. Запись `@selector(sayOuch)` не эквивалентна `@selector(sayOuch:)`.

А теперь снова взгляните на предупреждение компилятора. Фактически он говорит: «Вы создали переменную `timer`, но не используете ее». В некоторых ситуациях (таких, как эта) для подавления предупреждений необходимо пометить переменную и указать, что она не используется намеренно. Для этой цели используется модификатор `_unused`. Добавьте следующий фрагмент.

```
__unused NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:2.0
                                    target:logger
                                    selector:@selector(sayOuch:)
                                    userInfo:nil
                                    repeats:YES];
```

Постройте программу снова. На этот раз предупреждение исчезает.

Вспомогательные объекты

Таймеры просты. Они умеют делать только одно: срабатывать в нужный момент. Поэтому модель «приемник/действие» хорошо подходит для них. Многие простые элементы пользовательского интерфейса (такие, как кнопки и шкалы) тоже используют механизм «приемник/действие». А как насчет чего-то более сложного?

В главе 23 мы использовали объект `NSURLConnection` для загрузки данных с веб-сервера. Программа работала, но у такого решения есть два недостатка:

- Главный программный поток блокируется в ожидании поступления данных. Если бы этот метод использовался в настоящем приложении, пользовательский интерфейс переставал бы реагировать на действия пользователя до момента получения данных.

- Метод не предусматривает возможность обратного вызова - например, на случай, если веб-сервер запросит имя пользователя и пароль.

По этим причинам `NSURLConnection` обычно используется асинхронно. Иначе говоря, мы начинаем выборку данных, а затем ждем обратных вызовов при завершении загрузки, запросе веб-сервером удостоверений пользователя или ошибке загрузки.

Как получить эти обратные вызовы? Для этого `NSURLConnection` передается вспомогательный объект. Когда в ходе загрузки что-то происходит, подключение передает сообщения вспомогательному объекту. Какие сообщения? Создатель класса `NSURLConnection` определил протокол (список объявлений методов), которые могут быть реализованы вспомогательным объектом. Некоторые методы этого протокола:

```
- (NSURLRequest *)connection:(NSURLConnection *)c
             willSendRequest:(NSURLRequest *)req
            redirectResponse:(NSURLResponse *)res;
- (void)connection:(NSURLConnection *)sender
       didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)ch;

- (void)connection:(NSURLConnection *)sender
    didReceiveData:(NSData *)data;

- (void)connectionDidFinishLoading:(NSURLConnection *)sender;

- (void)connection:(NSURLConnection *)sender didFailWithError:(NSError *)error;
```

```
- (NSCachedURLResponse *)connection:(NSURLConnection *)sender
                  willCacheResponse:(NSCachedURLResponse *)cachedResponse;
```

Как видите, NSURLConnection живет куда более интересной жизнью, чем NSTimer. Сейчас мы создадим объект, который реализует часть методов этого протокола, и представим его NSURLConnection как вспомогательный объект. Для этой цели у NSURLConnection имеется делегат, на который ссылается указатель с именем delegate.

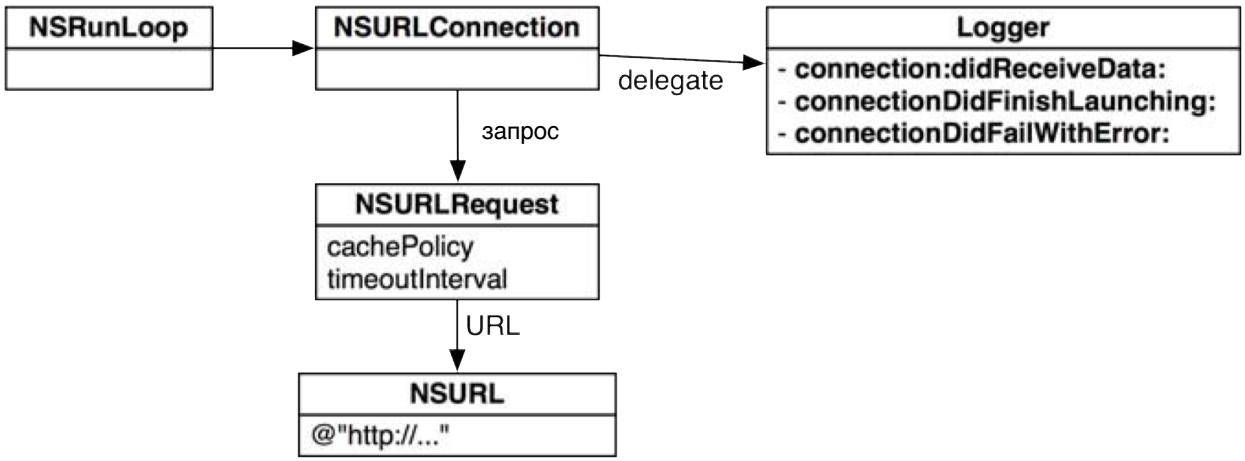


Рис. 2.4.4. Logger - делегат объекта NSUrlConnection

в main() создайте объект NSURLConnection и назначьте экземпляр Logger его делегатом:

```
#import <Foundation/Foundation.h>
#import "Logger.h"
int main (int argc, const char * argv[])
{
@autoreleasepool {
Logger *logger = [[Logger alloc] init]; NSURL *url = [NSURL URLWithString:
@"http://www.gutenberg.org/cache/epub/205/pg205.txt"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
__unused NSURLConnection *fetchConn
                          = [[NSURLConnection alloc] initWithRequest:request
                                  delegate:logger
                                  startImmediately:YES];
__unused NSTimer *timer
               = [NSTimer scheduledTimerWithTimeInterval:2.0
                                  target:logger
                                  selector:@selector(sayOuch:)
                                  userInfo:nil
                                  repeats:YES];

[[NSRunLoop currentRunLoop] run];
}
return 0;
}
```

Экземпляру Logger понадобится экземпляр NSMutableData для хранения поступающих байтов. Включите соответствующую переменную экземпляра в *Logger.h*:

```
#import <Foundation/Foundation.h>
@interface Logger : NSObject {
      NSMutableData *incomingData;
}
- (void)sayOuch:(NSTimer *)t;
@end
```

Теперь реализуйте некоторые методы делагата в *Logger.m*:

```
#import "Logger.h"
@implementation Logger
- (void)sayOuch:(NSTimer *)t
{
    NSLog(@"Ouch!");
}
// вызывается при каждом получении блока данных
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data
{
    NSLog(@"received %lu bytes", [data length]);
    // если изменяемый объект данных не существует, создать его
    if (!incomingData) {
        incomingData = [[NSMutableData alloc] init];
    }
    [incomingData appendData:data];
}

// вызывается при обработке последнего блока
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Got it all!");
    NSString *string = [[NSString alloc] initWithData:incomingData
                                             encoding:NSUTF8StringEncoding];
    incomingData = nil;
    NSLog(@"string has %lu characters", [string length]);
    // раскоментируйте следующую строку, чтобы увидеть весь файл
    // NSLog(@"The whole string is %@", string);
}
// вызывается в случае ошибки при загрузке данных
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error
{
    NSLog(@"connection failed: %@", [error localizedDescription]);
    incomingData = nil;
}

@end
```

Обратите внимание: мы реализуем не все методы протокола, а только те, которые важны для нашей программы.

Постройте и запустите программу. Вы увидите, что данные загружаются с веб-сервера по блокам. В конце загрузки делегат уведомляется о ее завершении.

Итак, до настоящего момента мы видим следующие правила обратного вызова: при отправке одного обратного вызова одному объекту Apple использует механизм «приемник/действие». При отправке различных вызовов одному объекту Apple использует вспомогательный объект с протоколом. (Протоколы более подробно рассматриваются в следующей главе.) Вспомогательные объекты обычно называются делегатами или источниками данных.

А если обратный вызов должен передаваться нескольким объектам?

Оповещения

Когда пользователь изменяет на Mac часовой пояс, информация об этом изменении может представлять интерес для многих объектов вашей программы. Каждый такой объект регистрируется в качестве наблюдателя в центре оповещений. При смене часового пояса центру отправляется оповещение `NSSystemTimeZoneDidChangeNotification`, а он передает его всем наблюдателям, заинтересованным в этом событии.

Внесите изменения в *main.m*, чтобы зарегистрировать экземпляр `Logger` для получений оповещений об изменениях часового пояса:

```
#import <Foundation/Foundation.h>
#import "Logger.h"
int main (int argc, const char * argv[])
{
@autoreleasepool {
Logger *logger = [[Logger alloc] init]; [[NSNotificationCenter defaultCenter]
                                    addObserver:logger
                                       selector:@selector(zoneChange:)

name:NSSystemTimeZoneDidChangeNotification
                                         object:nil];
NSURL *url = [NSURL URLWithString:
                       @"http://www.gutenberg.org/cache/epub/205/pg205.txt"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
__unused NSURLConnection *fetchConn
                           = [[NSURLConnection alloc] initWithRequest:request
                                                             delegate:logger
                                                     startImmediately:YES];
__unused NSTimer *timer
                = [NSTimer scheduledTimerWithTimeInterval:2.0
[[NSRunLoop currentRunLoop] run];
}
return 0;
}
```

Вызываемый метод реализуется *Logger.m*:

```
- (void)zoneChange:(NSNotification *)note
{
```

```
    NSLog(@"The system time zone has changed!");
}
```

Постройте и запустите программу. Во время выполнения откройте раздел системных настроек и измените часовой пояс вашей системы. Вы увидите, что в вашей программе будет вызван метод `zoneChange:`. (В некоторых системах он вызывается дважды; никаких проблем это не создает.)

Что использовать?

В этой главе были представлены три разновидности обратных вызовов. Как фирма Apple предлагает выбрать наиболее подходящую разновидность в каждой конкретной ситуации?

- Объекты, которые делают что-то одно (как `NSTimer`), используют механизм «приемник/действие».

- Объекты с более сложной «внутренней жизнью» (как `NSURLConnection`) используют вспомогательные объекты. Самым распространенным типом вспомогательного объекта является делегат.

- Объекты, которые должны передавать обратные вызовы нескольким другим объектам (таким, как `NSTimeZone`), используют оповещения.

Обратные вызовы и владение объектами

При использовании обратных вызовов возникает опасность того, что объекты, ожидающий обратных вызовов, не будут уничтожены положенным образом. Из-за этого было решено, что:

- *Центры оповещений не владеют своими наблюдателями*. Если объект является наблюдателем, он обычно удаляется из центра оповещений в его методе dealloc:

```
- (void)dealloc
 {
     [[NSNotificationCenter defaultCenter] removeObserver:self];
 }
```

- *Объекты не владеют своими делегатами и источниками данных*. Если вы создаете объект, который является делегатом или источником данных, он должен «попрощаться» в своем методе dealloc:

```
- (void)dealloc
 {
     [windowThatBossesMeAround setDelegate:nil];
     [tableViewThatBegsForData setDataSource:nil];
 }
```

- *Объектные владеют своими приемниками*. Если вы создаете объект, который является приемником. он должен обнулить указатель на приемник в своем методе dealloc:

```
- (void)dealloc
 {
     [buttonThatKeepsSendingMeMessages setTarget:nil];
 }
```

В нашей программе все эти проблемы неактуальны, потому что объект `Logger` не будет уничтожен до завершения программы. (Кроме того, в этом упражнении я использовал два документированных исключения из правил: `NSURLConnection` владеет своим делегатом, а `NSTimer` владеет своим приемником.)

25. Протоколы

Пришло время обсудить одну, отчасти абстрактную концепцию. Однажды кто-то сказал: «То, что ты есть, и то, что ты делаешь - не одно и то же». Это утверждение справедливо и для объектов: класс объекта не всегда совпадает с его ролью в работающей системе. Например, объект может быть экземпляром `NSMutableArray`, тогда как в приложении он может обеспечивать управление очередью заданий печати.

Действительно хорошо написанный класс имеет более общую природу, которая не ограничивается его ролью в одном конкретном приложении. Это позволяет по разному использовать экземпляры данного класса.

Мы уже говорили о том, как определить класс. Возможно ли определить роль? В определенной степени для определения роли можно использовать конструкцию @protocol.

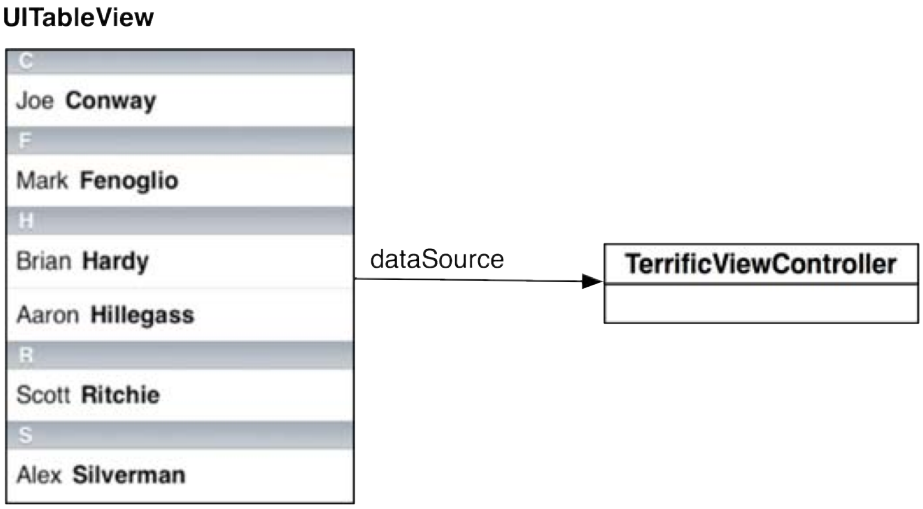


Рис. 25.1. Источник данных UITableView

Например, в приложениях iOS данные часто отображаются в экземпляре класса `UITableView`. Однако объект `UITableView` не содержит данные, которые в нем отображаются; он должен получить данные из другого источника. Ему необходимо сообщить: «Вот объект, который будет выполнять функции твоего источника данных».

Как разработчик, создавший класс `UITableView`, определил роль источника данных `UITableView`? Он создал протокол - список объявлений методов. Одни методы являются обязательными, другие относятся к дополнительным. Если ваш объект собирается выполнять функции некоторой роли, он должен реализовать обязательные методы и по желанию реализует дополнительные.

Протокол источника данных для `UITableView` называется `UITableViewDataSource` и выглядит следующим образом (с моими комментариями):

```
// Протоколы, как и классы, могут наследовать от других протоколов.
// Этот протокол наследует от протокола NSObject
@protocol UITableViewDataSource <NSObject>

// следующие методы обязательно должны быть реализованы каждым
// источником данных табличного представления
@required

// табличное представление делится на секции,
// каждая секция может состоять из нескольких строк
- (NSInteger)tableView:(UITableView *)tv numberOfRowsInSection:
(NSInteger)section;

// индексный путь состоит из двух целых чисел (секция и строка)
- (UITableViewCell *)tableView:(UITableView *)tv
         cellForRowAtIndexPath:(NSIndexPath *)ip;

// а эти методы могут быть (а могут и не быть) реализованы
// источником данных табличного представления
@optional

// если источник данных не реализует этот метод,
// табличное представление состоит только из одной секции
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tv;

// Строки могут удаляться и перемещаться
- (BOOL)tableView:(UITableView *)tv canEditRowAtIndexPath:(NSIndexPath *)ip;
- (BOOL)tableView:(UITableView *)tv canMoveRowAtIndexPath:(NSIndexPath *)ip;
- (void)tableView:(UITableView *)tv
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
 forRowAtIndexPath:(NSIndexPath *)ip;
- (void)tableView:(UITableView *)tv
moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath
       toIndexPath:(NSIndexPath *)destinationIndexPath;

// Для экономии места я опускаю объявления
// нескольких дополнительных методов.
@end
```

Протоколы, как и классы, тоже имеют свои справочные страницы в документации разработчика Apple, с поддержкой поиска и вывода списка методов протокола.

Когда вы создаете класс, который должен использоваться в качестве источника данных `UITableView`, вы должны явно указать в заголовочном файле: «Этот класс поддерживает протокол `UITableViewDataSource`». Это выглядит примерно так:

```
@interface TerrificViewController : UIViewController <UITableViewDataSource>
...
@end
```

Иначе говоря, «`TerrificViewController` - субкласс `UIViewController`, который поддерживает протокол `UITableViewDataSource`».

Если класс поддерживает несколько протоколов, они перечисляются в угловых скобках:

```
@interface TerrificViewController : UIViewController
      <UITableViewDataSource, UITableViewDelegate, UITextFieldDelegate>
```

Файл *TerrificController.m* должен содержать реализации обязательных методов в каждом протоколе. Если вы забудете реализовать один из обязательных методов, компилятор выдаст предупреждение.

Также просмотрите список дополнительных методов и выберите те из них, которые вам хотелось бы реализовать. Реализованные методы будут автоматически вызваны в нужный момент времени.

И последнее замечание: в программе `Callbacks` из главы 24 экземпляр `Logger` становится делегатом объекта `NSURLConnection`. Однако в файле *Logger.h* не указано, что `Logger` поддерживает соответствующий протокол.

На момент написания книги формального протокола для делегатов `NSURLConnection` еще не существует. Я не удивлюсь, если это произойдет. (А если при построении программы `Callbacks` вы получили предупреждение вида «Данный объект не поддерживает протокол `NSURLConnectionDelegate`», значит, уже произошло.)

26. Списки свойств

Иногда требуется создать файл в формате, удобном как для компьютеров, так и для людей. Допустим, вы хотите сохранить в файле описание своего портфеля акций. Добавление новых акций было бы удобно осуществлять вручную, но также было бы хорошо, чтобы содержимое файла нормально читалось вашими программами. Для решения подобных проблем многие программисты Objective-C используют списки свойств.

Список свойств представляет собой комбинацию следующих элементов:

- NSArray
- NSOictionary
- NSString
- NSOata
- NSOate
- NSNumber (целое, вещественное или логическое значение)

Например, массив словарей со строковыми ключами и объектами даты представляет собой список свойств.

Операции чтения и записи списка свойств в файл очень просты. В Xcode создайте новый проект: программу командной строки Foundation с именем stockz. Включите в нее следующий код:

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

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

    @autoreleasepool {

        NSMutableArray *stocks = [[NSMutableArray alloc] init];
        NSMutableDictionary *stock;

        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"AAPL"

                  forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:200]

                  forKey:@"shares"];
        [stocks addObject:stock];

stock = [NSMutableDictionary dictionary];
[stock setObject:@"GOOG"
          forKey:@"symbol"];
[stock setObject:[NSNumber numberWithInt:160]
          forKey:@"shares"];
[stocks addObject:stock];
```

```
[stocks writeToFile:@"/tmp/stocks.plist"
         atomically:YES];
}
return 0;
}
```

(Обратите внимание на повторное использование указателя `stock`. Сначала он указывает на первый словарь, а потом на второй.)

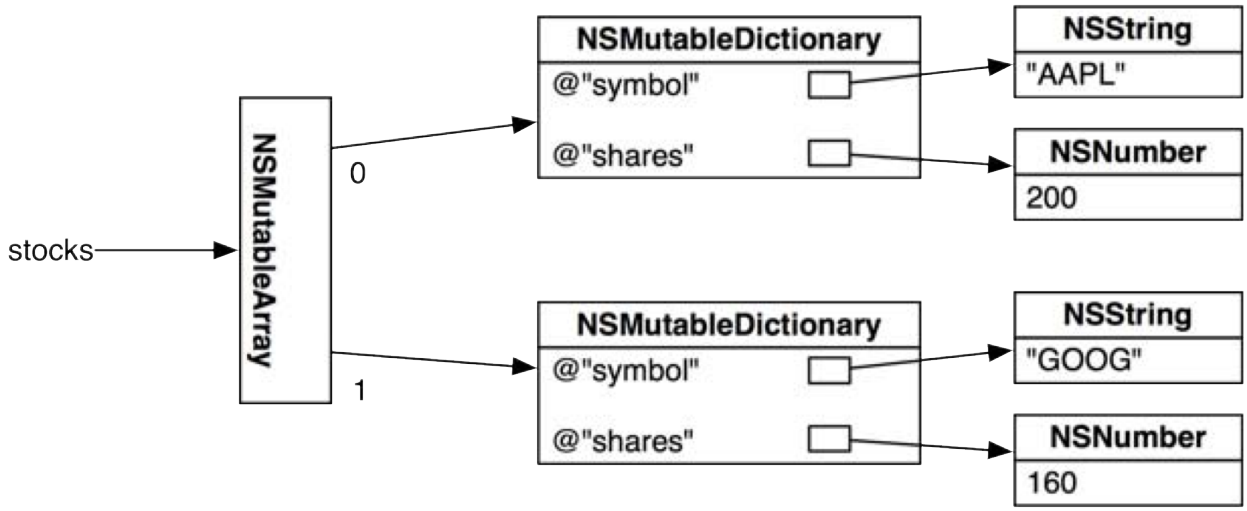


Рис. 26.1. Массив словарей

Запущенная программа создает файл *stocks.plist*. Открыв его в текстовом редакторе, вы увидите примерно следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
  "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <dict>
        <key>shares</key>
        <integer>200</integer>
        <key>symbol</key>
        <string>AAPL</string>
    </dict>
    <dict>
        <key>shares</key>
        <integer>160</integer>
        <key>symbol</key>
        <string>GOOG</string>
    </dict>
</array>
</plist>
```

Неплохо, верно? XML. Хорошо читается. И всего одна строка кода.

Если вам придется создавать списки свойств вручную, учтите, что в Xcode для работы со списками свойств имеется специальный встроенный редактор.

Теперь добавьте строку для чтения файла:

```
int main(int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *stocks = [[NSMutableArray alloc] init];
        NSMutableDictionary *stock;
        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"AAPL"
                  forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:200]
                  forKey:@"shares"];
        [stocks addObject:stock];
        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"GOOG"
                  forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:160]
                  forKey:@"shares"];
        [stocks addObject:stock];
        [stocks writeToFile:@"/tmp/stocks.plist"
                 atomically:YES];
        NSArray *stockList = [NSArray arrayWithContentsOfFile:@"/tmp/
stocks.plist"];
        for (NSDictionary *d in stockList) {
            NSLog(@"I have %@ shares of %@",
      [d objectForKey:@"shares"], [d objectForKey:@"symbol"]);
}
}
return 0;
}
```

Постройте и запустите программу

Упражнение

Напишите программу для создания списка свойств, содержащего все 8 типов: массив, словарь, строка, данные, дата, целое число, вещественное число, логическое значение.

IV

Событийное программирование

Мы подошли к тому, ради чего вы начали читать эту книгу - к написанию приложений iOS и Cocoa. В следующих двух главах вы получите представление о разработке приложений. Ваши приложения будут иметь графический интерфейс (GUI, Grapllical User Interface), и они будут управляться событиями.

Как работают приложения командной строки? Вы запускаете программу, а она начинает делать свое дело, пока не закончит. С приложениями, управляемыми событиями, дело обстоит иначе. Приложение запускает цикл событий, который ожидает поступления событий. Когда событие происходит, приложение обрабатывает его: выполняет методы, отправляет сообщения и т. д.

Сначала мы напишем приложение iOS, а затем создадим похожее приложение Cocoa. Cocoa - набор библиотек, разработанных Apple для программирования для Mac. Вы уже знакомы с одной из этих библиотек - Foundation. Cocoa и Cocoa Touch включают некоторые общие библиотеки - такие, как Foundation. Другие библиотеки предназначены для одной конкретной платформы.

27. Первое приложение iOS

В этой главе мы создадим свое первое приложение iOS: простой список текущих задач `iTahDoodle`, который будет хранить свои данные в списке свойств. Вот как будет выглядеть `iTahDoodle` после завершения работы.

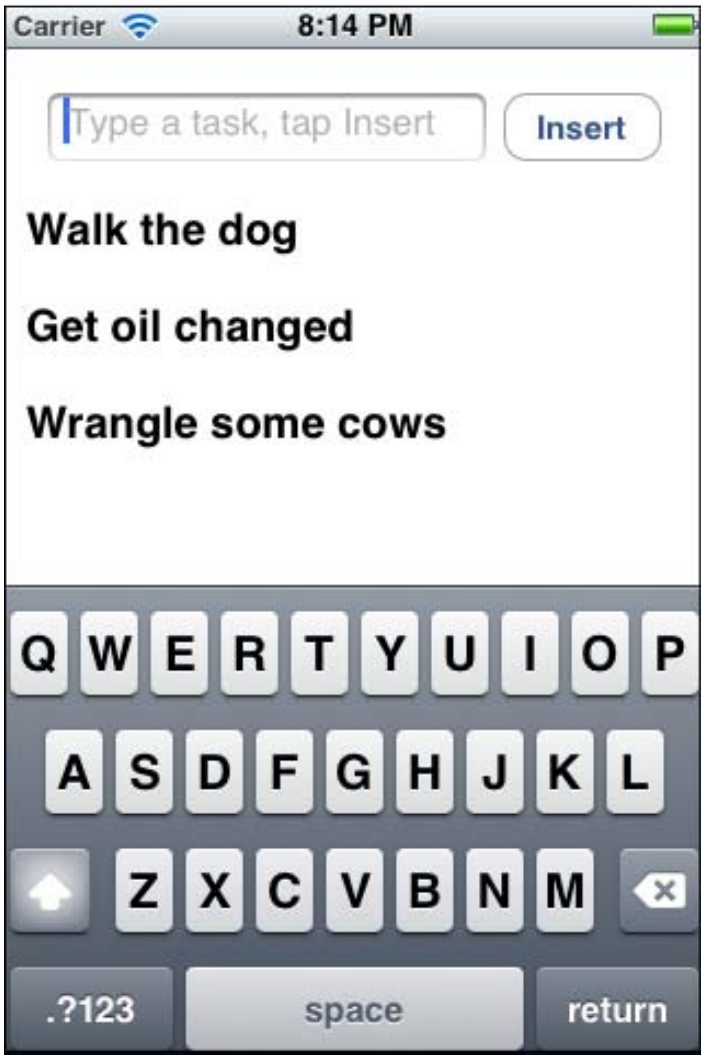


Рис. 27.1. ПриложениеiTahDoodle

Все приложения iOS относятся к категории приложений, управляемых событиями. В них цикл событий ожидает, пока что-нибудь произойдет. Далее ожидающее приложение реагирует на события, сгенерированные пользователем (такие, как нажатие кнопки) или системой (например, предупреждение о нехватке памяти).

Начинаем работу над iTahDoodle

В Xcode выполните команду `File→New→New Project...`. В разделе iOS (не в разделе Mac OS X!) выберите строку Application. В открывшемся наборе шаблонов выберите шаблон пустого приложения Empty Application.

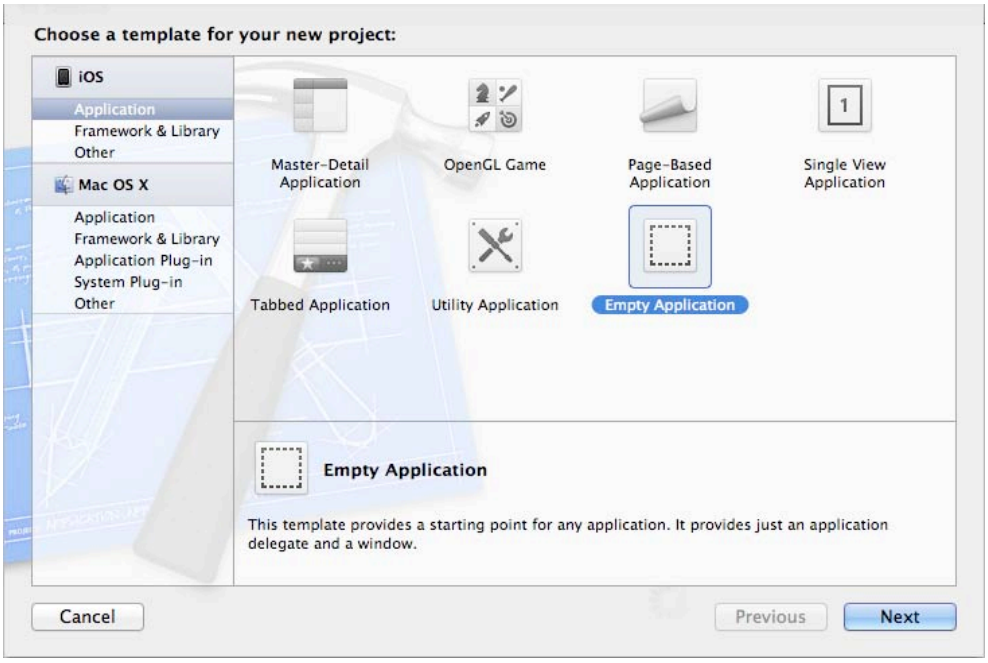


Рис. 27.2. Создание нового приложения iOS

Шаблоны проектов Xcode призваны упростить вашу работу. Они содержат заготовки кода, которые могут ускорить разработку. Однако мы намеренно выбрали шаблон пустого приложения. Если Xcode сгенерирует слишком много готового кода, это только помешает вам понять, как работает программа.

Имена шаблонов часто изменяются в новых версиях Xcode; не удивляйтесь, если в вашей версии шаблоны не будут полностью совпадать с тем, что вы видите на рис. 27.2. Найдите шаблон с самым простым названием и отредактируйте код, приведя его в соответствие с кодом книги. Если у вас возникнут трудности с согласованием кода или шаблонов проектов, обращайтесь за помощью на форум Big Nerd Ranch по адресу *forums.bignerdranch.com*.

После выбора шаблона пустого приложения щелкните на кнопке Next и присвойте проекту имя `iTahDoodle`. Поля Company Identifier и Bundle Identifier обеспечивают уникальность приложений в App Store. Оба значения представляют собой строки в обратной доменной записи. Так, Big Nerd Ranch вводит в поле Company Identifier строку com.bignerdranch.

Содержимое поля Class Prefix будет вставлено перед именем исходного класса, сгенерированного шаблоном за вас. Этот префикс (состоящий из двух-трех букв) нужен для того, чтобы имена ваших классов отличались от имен классов Apple или других разработчиков. (Введите префикс BNR, чтобы ваш код не отличался от приведенного в книге.)

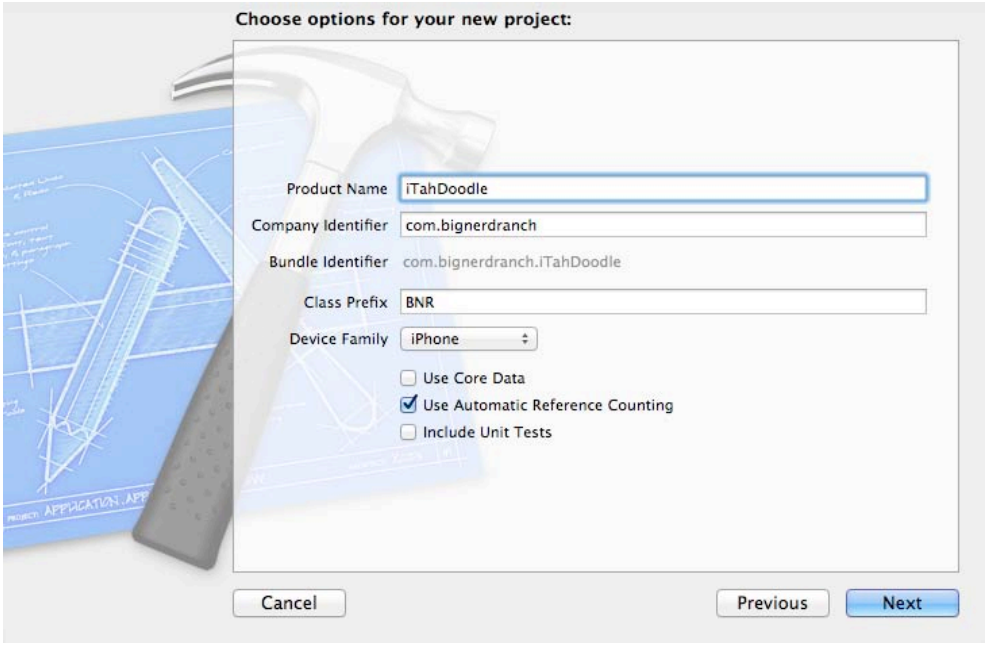


Рис. 27.3. Настройка параметров проекта iTahDoodle

Обратите внимание на использование префиксов в коде Apple. Все классы Apple, встречавшиеся нам до сих пор, начинались с префикса NS-сокращения от «NeXTSTEP» (платформа, для которой изначально проектировалась библиотека Foundation). В этой главе мы также будем использовать классы из библиотеки `UIKit`, имена которых будут начинаться с префикса UI.

Напоследок укажите, что приложение `iTahDoodle` является приложением для iPhone (а не приложением для iPad или универсальным приложением). В `iTahDoodle` будет использоваться автоматический подсчет ссылок, но поддержка Core Data и модульные тесты не понадобятся.

BNRAppDelegate

Создавая ваш проект по шаблону Empty Application, Xcode создает за вас всего один класс: `BNRAppDelegate`. Класс «делегата приложения» является отправной точкой для создания приложения; такой класс присутствует в каждом приложении для iOS. `BNRAppDelegate` существует в одном экземпляре, который отвечает за обработку событий и координацию работы других объектов в приложении.

Откройте файл *BNRAppDelegate.h*. Добавьте четыре переменных и метод экземпляра. Первые три переменные экземпляра представляют собой указатели на объекты, которые пользователь видит и с которыми может взаимодействовать, - табличное представление для отображения списка задач, текстовое поле для ввода новой задачи и кнопка для включения введенной задачи в таблицу. Четвертый объект - изменяемый массив. В нем задачи будут храниться в виде строк.

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

```
@interface BNRAppDelegate : UIResponder <UIApplicationDelegate>
{
    UITableView *taskTable;
    UITextField *taskField;
    UIButton *insertButton;
    NSMutableArray *tasks;
}
- (void)addTask:(id)sender;
@property (strong, nonatomic) UIWindow *window;
@end
```

Обратите внимание: шаблон импортирует файл *UIKit.h*. Библиотека UIKit содержит большинство классов для iOS, включая `UITableView`, `UITextField` и `UIButton`. Кроме того, `BNRAppDelegate` поддерживает протокол `UIApplicationDelegate`.

Заметьте, что мы не импортируем *Foundation.h*. Как тогда использовать `NSMutableArray`? В этом шаблоне заголовок библиотеки Foundation входит в предварительно откомпилированный файл заголовка проекта, так что классы Foundation автоматически доступны для использования. (Не верите? Щелкните на файле *iTahDoodle-Prejix.pch* в разделе Supporting Files навигатора проекта и убедитесь сами.)

Добавление вспомогательной функции C

По объявлениям переменных экземпляров можно понять, что приложение `iTahDoodle` будет содержать минимум четыре дополнительных объекта. На прежде чем браться за эти объекты, мы напишем функцию C. В Objective-C для выполнения основной работы обычно используются методы вместо функций, поэтому функции C в приложениях Objective-C часта называются «вспомогательными» функциями.

Приложение `iTahDoodle` хранит задачи пользователя в списке свойств - то есть в файле XML. Это означает, что вам понадобится как-та определить местонахождение файла во время выполнения приложения. Мы напишем функцию C, которая возвращает путь к файлу в формате `NSString`.

Чтобы добавить вспомогательную функцию в приложение, необходимо сначала объявить ее в файле *BNRAppDelegate.h*.

```
#import <UIKit/UIKit.h>
// объявление вспомогательной функции для получения пути
// к каталогу на диске, который будет использоваться
// для сохранения списка задач
NSString *docPath(void);
@interface BNRAppDelegate : UIResponder
<UIApplicationDelegate>
{
    UITableView *taskTable;
    UITextField *taskField;
    UIButton *insertButton;
```

```
    NSMutableArray *tasks;
}
- (void)addTask:(id)sender;
@property (strong, nonatomic) UIWindow *window;
@end
```

Обратите внимание: объявление `docPath()` находится да объявления класса. Эта связана с тем, что хотя функция `docPath()` объявляется в файле *BNRAppDelegate.h*, она не является его частью. Вообще говоря, для нее можно было бы создать пару отдельных файлов в проекте `iTahDoodle`. На поскольку `iTahDoodle` содержит всего одну вспомогательную функцию, мы для простаты размещаем ее в файлах класса делегата приложения.

Теперь откройте *BNRAppDelegate.m* и реализуйте вспомогательную функцию. И снова, так как `docPath()` не является частью класса, реализация должна располагаться после `#import`, но до строки `@implementation` (в которой начинается реализация класса).

```
#import "BNRAppDelegate.h"
// Вспомогательная функция для получения пути к списку задач
// хранящемуся на диске
NSString *docPath()
{
    NSArray *pathList =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                    NSUserDomainMask,
YES);
    return [[pathList objectAtIndex:0]
stringByAppendingPathComponent:@"data.td"];
}

@implementation
```

Функция `docPath()` вызывает другую функцию C, `NSSearchPathForDirectoriesInDomains()`. Эта функция ищет каталоги, удовлетворяющие заданному критерию, н возвращает их в массиве. Пока не обращайте внимания на аргументы: почти во всех приложениях iOS, которые вы когда-либо напишете, вы будете передавать этой функции эти три аргумента и получать массив с ровно одним элементом. (Если вас интересует, как работает функция `NSSearchPathForDirectoriesInDomains()`, найдите ее описание в справочнике «Foundation Functions Reference» в документации разработчика.)

Объекты в приложении iTahDoodle

Вернемся к нашим объектам. Вам уже известны пять объектов, образующих приложение `iTahDoodle`: прежде всего экземпляр `BNRAppDelegate`, а в нем хранятся

указатели на четыре других объекта: экземпляры `UITableView`, `UITextField`, `UIButton` и `NSMutableArray`.

Но прежде чем мы перейдем к настройке и связыванию этих объектов - немного теории о объектах и отношениях между ними.

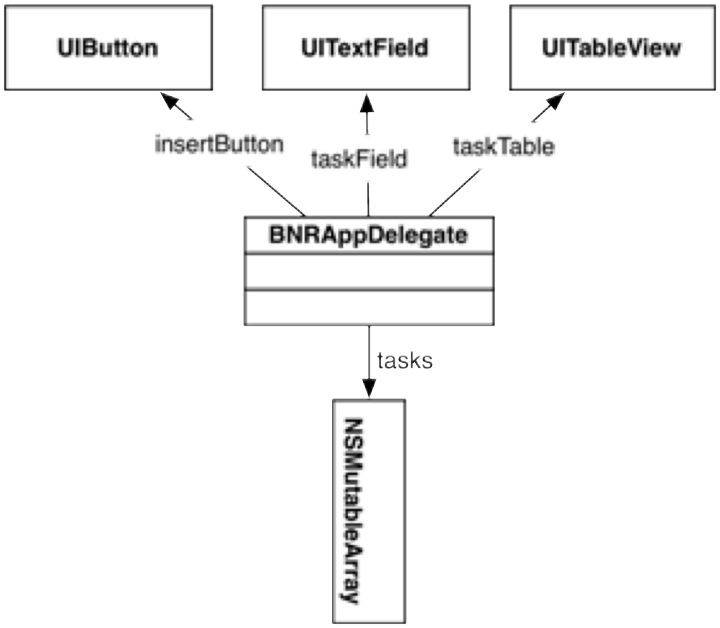


Рис.27.4, Диаграмма объектов iTahDoodle

Модель-Представление-Контроллер

В паттерне проектирования «Модель-Представление-Контроллер», или сокращенно MVC (Model-View-Controller), каждый созданный вами класс должен относиться к одной из трех функциональных категорий: модель, представление или контроллер. Краткая сводка «разделения обязанностей» между категориями:

- *Модели* отвечают за хранение данных и предоставление их другим объектам. Модели ничего не знают о пользовательском интерфейсе и не умеют отображать себя на экране; их единственное предназначение - хранение и управление данными. `NSString`, `NSDate` и `NSArray` - традиционные объекты моделей. В `iTahDoodle` пока используется только один объект модели: экземпляр `NSMutableArray`, в котором хранятся задачи. Однако каждая от- дельная задача будет описываться экземпляром `NSString`, и он тоже будет объектом модели.

- *Представления* являются визуальными элементами приложений. Они знают, как отобразить себя на экране и как реагировать на ввод данных пользователем. Представления не располагают информацией о данных, которые ими отображаются, об их структуре и способе хранения. Типичные примеры объектов представлений - `UIView` и его различные субклассы, включая `UIWindow`. В приложении `iTahDoodle`

объекты представлений являются экземплярами `UITableView`, `UITextView` и `UIButton`. Простое и практичное правило: если вы видите какой-то элемент приложения на экране, значит, это представление.

- *Контроллеры* выполняют логику, необходимую для связывания и управления различными частями вашего приложения. Они обрабатывают события и координируют работу других объектов вашего приложения. Контроллеры - настоящая «рабочая сила» любого приложения. Хотя в `iTahDoodle` используется только один контроллер `BNRAppDelegate`, в сложном приложении вполне может быть много разных контроллеров, координирующих работу объектов моделей и представлений, а также других контроллеров.

На рис.27.5 представлена схема передачи управления между объектами при поступлении пользовательского события (например, нажатия кнопки). Обратите внимание: модели и представления не взаимодействуют друг с другом напрямую; в середине любого взаимодействия находится контроллер, получающий сообщения от одних объектов и передающий инструкции другим.

Рис. 27.5. Обработка пользовательского

Очень важно четко понимать обязанности каждого из участников паттерна `MVC`. Большая часть программных интерфейсов Cocoa и Cocoa Touch ориентирована на `MVC` - и ваш код тоже должен ориентироваться на этот паттерн.

А теперь вернемся к контроллеру, который является экземпляром `BNRAppDelegate`.

Делегат приложения

При запуске приложения iOS производится инициализация, незаметная для пользователя. В этой фазе создается экземпляр `UIApplication`, который управляет состоянием приложения и обеспечивает связь с операционной системой. Также создается экземпляр `BNRAppDelegate`, назначаемый делегатом экземпляра `UIApplication` (отсюда и термин «делегат приложения»).

В процессе запуска приложение еще не готово к выполнению работы или вводу данных. Когда эта ситуация изменяется, экземпляр `UIApplication` отправляет своему делегату сообщение `application:didFinishLaunchingWithOptions:`. Этот метод очень важен. Именно в нем выполняются все действия, которые должны быть выполнены до того, как пользователь начнет взаимодействие с приложением.

В приложении `iTahDoodle` одной из операций, выполняемых в этом методе, должен быть поиск списка свойств и загрузка его в массив. В файле *BNRAppDelegate.m* уже имеется готовая заглушка для метода `application:didFinishLaunchingWithOptions:`. Найдите ее и замените код в фигурных скобках следующим:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
// попытка загрузки существующего списка задач
// из массива, хранящегося на диске.
    NSArray *plist = [NSArray arrayWithContentsOfFile:docPath()];
    if (plist) {
// если набор данных существует, он копируется в переменную экземпляра.
        tasks = [plist mutableCopy];
    } else {
// в противном случае просто создаем пустой исходный набор.
    tasks = [[NSMutableArray alloc] init];
}
}
```

Заметили `#pragma mark` в начале кода? Программисты Objective-C часто используют эту конструкцию для группировки методов в классах. Xcode тоже знает о ее существовании. На панели навигации в верхней части редактора найдите элемент справа от *BNRAppDelegate.m*. (Возможно, он содержит текст `@implementation AppDelegate`, но содержимое зависит от текущего местонахождения курсора в коде.) Щелкните на элементе; Xcode выведет список позиций в этом файле. Если щелкнуть на любой из предложенных позиций, вы перейдете прямо к соответствующему месту кода. Обратите внимание на присутствие в списке директивы `pragma mark`. Этот способ перемещения чрезвычайно удобен, когда класс содержит много методов.

Подготовка представлений

Чтобы приложение было готово к работе, мы должны подготовить объекты представлений: создать их, настроить и отобразить на экране. Логично, не правда ли? Пользователь не сможет нажать кнопку, которая не существует или не отображается на экране.

В `iTahDoodle` мы создадим представления на программном уровне в `application: didFinishLaunchingWithOptions:`. Также существуют визуальные средства создания представлений, которыми мы воспользуемся в следующей главе.

Я должен предупредить, что дальше код становится достаточно серьезным. Подробный синтаксис создания и отображения представлений на экране - тема для отдельной книги, посвященной программированию приложений iOS. Попробуйте уловить суть происходящего. Мы создаем каждый объект, а затем настраиваем его, задавая значения некоторых свойств. Затем настроенные объекты включаются как *субпредставления* в объект окна, и наконец, окно размещается на экране.

```
#pragma mark - Application delegate callbacks
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
// попытка загрузки существующего списка задач
// из массива, хранящегося на диске.
    NSArray *plist = [NSArray arrayWithContentsOfFile:docPath()];
    if (plist) {
// если набор данных существует, он копируется в переменную экземпляра.
        tasks = [plist mutableCopy];
    } else {
// в противном случае просто создаем пустой исходный набор.
        tasks = [[NSMutableArray alloc] init];
    }

// создание и настройка экземпляра UIWindow.
// структура CGRect представляет прямоугольник с базовой точкой
// (x,y) и размерами (width,height)
    CGRect windowFrame = [[UIScreen mainScreen] bounds];
    UIWindow *theWindow = [[UIWindow alloc] initWithFrame:windowFrame];
    [self setWindow:theWindow];

// определение граничных прямоугольников для трех элементов
// пользовательского интерфейса.
// CGRectMake() создает экземпляр CGRect по данным (x, y, width, height)
    CGRect tableFrame = CGRectMake(0, 80, 320, 380);
    CGRect fieldFrame = CGRectMake(20, 40, 200, 31);
    CGRect buttonFrame = CGRectMake(228, 40, 72, 31);

// создание и настройка табличного представления
    taskTable = [[UITableView alloc] initWithFrame:tableFrame

style:UITableViewStylePlain];
    [taskTable setSeparatorStyle:UITableViewCellSeparatorStyleNone];
```

```
// создание и настройка текстового поля для ввода новых задач
    taskField = [[UITextField alloc] initWithFrame:fieldFrame];
    [taskField setBorderStyle:UITextBorderStyleRoundedRect];
    [taskField setPlaceholder:@"Type a task, tap Insert"];

// создание и настройка кнопки Insert в виде прямоугольника
// с закругленными углами.
    insertButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [insertButton setFrame:buttonFrame];

// работа кнопок основана на механизме обратного вызова типа
// "приемник/действие". Действие кнопки Insert настраивается
// на вызов метода -addTask: текущего объекта
    [insertButton addTarget:self
                     action:@selector(addTask:)
           forControlEvents:UIControlEventTouchUpInside];

// Определение надписи на кнопке
    [insertButton setTitle:@"Insert"
                  forState:UIControlStateNormal];

// включение трех элементов пользовательского интерфейса в окно
    [[self window] addSubview:taskTable];
    [[self window] addSubview:taskField];
    [[self window] addSubview:insertButton];

// Завершение настройки окна и отображение его на экране
    [[self window] setBackgroundColor:[UIColor whiteColor]];
    [[self window] makeKeyAndVisible];
return YES;
}
```

Выполнение в iOS Simulator

Итак, представления созданы; теперь можно построить приложение и посмотреть, как они выглядят. В Xcode найдите раскрывающийся список `Scheme` рядом с кнопкой `Run`. Выберите в нем строку iPhone 5.X Simulator для запуска последней версии эмулятора iOS:

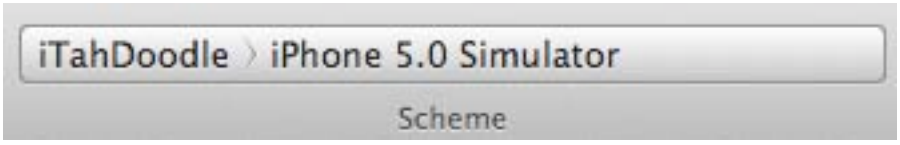


Рис. 27.6. Список Scheme

Постройте и запустите приложение. Компилятор выдает предупреждение о том, что метод `addTask`: еще не реализован. Пока не обращайте на него внимания; вскоре мы реализуем `addTask`:.

Эмулятор позволяет запускать приложения Cocoa Touch на настольном компьютере. Это простой и быстрый способ увидеть, как ваша программа будет выглядеть и работать на устройстве iOS.

Вы увидите все представления, созданные и настроенные в `application:didFinishLaunchingWithOptions:`, но сделать они ничего не могут. Более того, попытка нажать кнопку Insert приведет к сбою приложения, потому что метод действия кнопки `addTask:` еще не реализован. (Это одна из причин, по которой компилятор выдал предупреждение.):

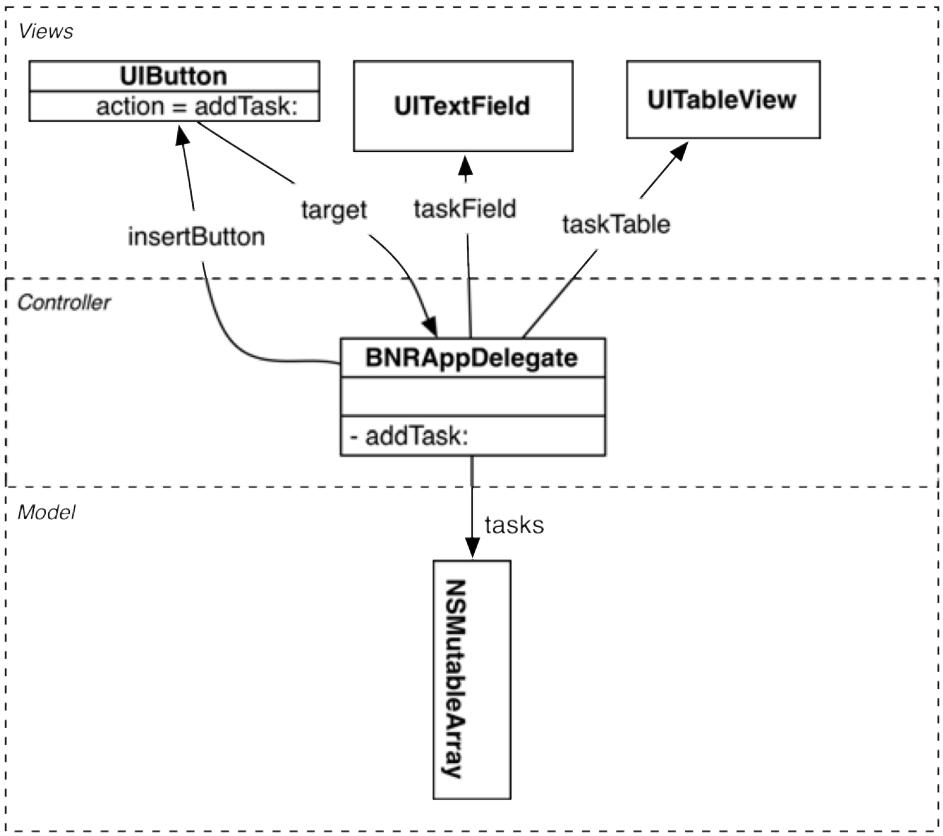


Рис. 27.7, Диаграмма объектов iTahDoodle

Связывание табличного представления

Табличное представление отображается на экране, но оно понятия не имеет, какие данные в нем должны выводиться. Табличное представление, будучи объектом представления, не располагает никакой информацией о своих данных. Ему нужен объект, который будет использоваться в качестве источника данных. В `iTahDoodle` источником данных табличного представления будет экземпляр `BNRAppDelegate`.

В файле *BNRAppDelegate.m* измените метод `application:didFinishLaunchingWithOptions:` так, чтобы он отправлял табличному представлению сообщение, назначающее экземпляр `BNRAppDelegate` его источником данных:

```
// Создание и настройка табличного представления
taskTable = [[UITableView alloc] initWithFrame:tableFrame
                                         style:UITableViewStylePlain];
[taskTable setSeparatorStyle:UITableViewCellSeparatorStyleNone];

// Назначение текущего объекта источником данных табличного представления
[taskTable setDataSource:self];

// Создание и настройка текстового поля для ввода новых задач
taskField = [[UITextField alloc] initWithFrame:fieldFrame];
```

Чтобы источник данных табличного представления выполнял свою работу, он должен реализовать методы протокола `UITableViewDataSource`. Сначала следует объявить о поддержке этого протокола `BNRAppDelegate` в файле *BNRAppDelegate.h*:

```
@interface BNRAppDelegate : UIResponder
<UIApplicationDelegate, UITableViewDataSource>
{
    UITableView *taskTable;
    UITextField *taskField;
    UIButton *insertButton;
    NSMutableArray *tasks;
}
- (void)addTask:(id)sender;
```

Протокол `UITableViewDataSource` содержит два обязательных метода, которые должны быть реализованы классом `BNRAppDelegate`. Как минимум источник данных табличного представления должен быть способен сообщить табличному представлению, сколько строк содержит заданная секция таблицы и что должно отображаться в ячейке заданной строки. Реализуйте методы соответствующим образом:

```
#pragma mark - Table View management
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
// так как данное табличное представление содержит только одну
// секцию, количество строк в ней равно количеству элементов
// массива tasks
    return [tasks count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
// для улучшения быстродействия мы заново используем
// ячейки, вышедшие за пределы экрана, и возвращаем их
// с новым содержимым вместо того, чтобы всегда создавать
// новые ячейки. Сначала мы проверяем, имеется ли ячейка,
// доступная для повторного использования.
    UITableViewCell *c = [taskTable dequeueReusableCellWithIdentifier:@"Cell"];
```

```
    if (!c) {
// .. и создаем новую ячейку только в том случае,
// если доступных ячеек нет.
        c = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:@"Cell"];
}

// Затем ячейка настраивается в соответствии с информацией
// объекта модели (в нашем случае это массив todoItems)
    NSString *item = [tasks objectAtIndex:[indexPath row]];
    [[c textLabel] setText:item];
    // and hand back to the table view the properly configured cell
return c; }
```

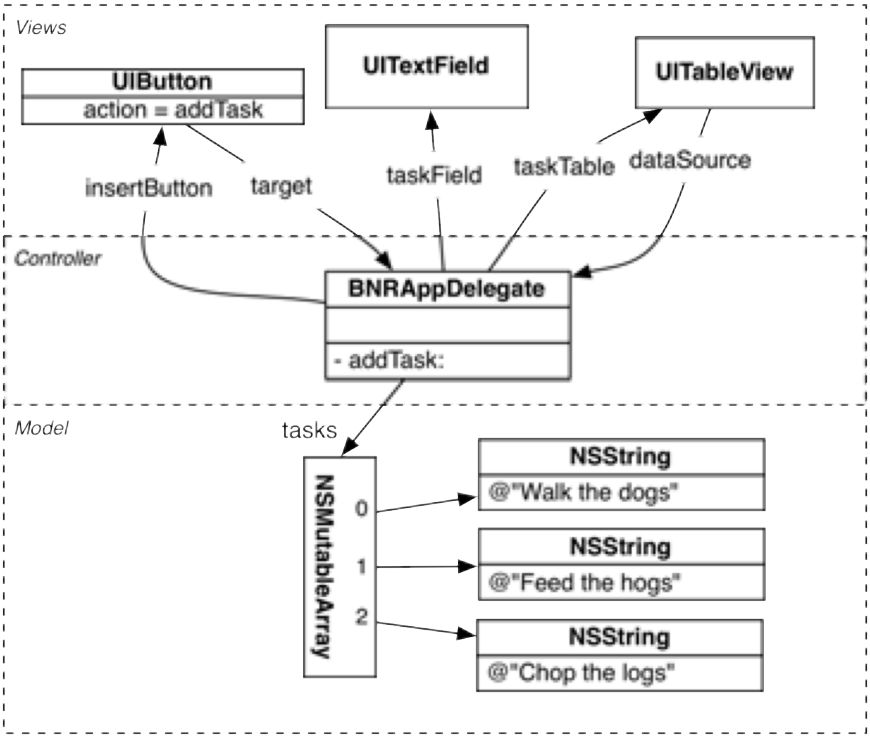


Рис. 27.8, Полная диаграмма объектов iTahDoodle

Чтобы протестировать приложение, добавьте данные прямо в массив в начале **application:didFinishLaunchingWithOptions:**.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
// Попытка загрузки существующего списка задач
// из массива , хранящегося на диске.
    NSArray *plist = [NSArray arrayWithContentsOfFile:docPath()];
    if (plist) {
// Если набор данных существует, он копируется
// в переменную экземпляра.
        tasks = [plist mutableCopy];
    } else {
// в противном случае просто создаем пустой исходный набор.
        tasks = [[NSMutableArray alloc] init];
```

```
    }
// массив tasks пуст?
    if ([tasks count] == 0) {
        // Put some strings in it
        [tasks addObject:@"Walk the dogs"];
        [tasks addObject:@"Feed the hogs"];
        [tasks addObject:@"Chop the logs"];
}
// создание и настройка экземпляра UIWindow
    CGRect windowFrame = [[UIScreen mainScreen] bounds];
    UIWindow *theWindow = [[UIWindow alloc] initWithFrame:windowFrame];
    [self setWindow:theWindow];
...
}
```

Постройте и запустите приложение. В табличном представлении должны отображаться тестовые данные, но новые задачи по-прежнему не добавляются. За дело!

Добавление новых задач

Создавая экземпляр `UIButton` в `application:didFinishLaunchingWithOptions`, мы задали для него пару «приемник/действие»:

```
[insertButton addTarget:self
                 action:@selector(addTask:)
      forControlEvents:UIControlEventTouchUpInside];
```

Приемником является self, а действием - `addTask:`. Таким образом, кнопка `Insert` отправляет `BNRAppDelegate` сообщение `addTask:`. Следовательно, мы должны реализовать метод `addTask:` в *BNRAppDelegate.m*.

```
- (void)addTask:(id)sender
{
    // получение задачи
    NSString *t = [taskField text];
    // выход, если поле taskField пусто
    if ([t isEqualToString:@""]) {
return; }
    // включение задачи в рабочий массив
    [tasks addObject:t];
    // обновление таблицы, чтобы в ней отоброжался новый элемент
    [taskTable reloadData];
    // очистка текстового поля
    [taskField setText:@""];
    // клавиатура убирается с экрана
    [taskField resignFirstResponder];
}
```

А что это за вызов `resignFirstResponder`? В двух словах дело обстоит так.

Некоторые объекты представлений также являются элементами управления - представлениями, с которыми пользователь может взаимодействовать. Кнопки, текстовые поля, ползунки - все это при меры элементов управления. Когда на экране находятся элементы, один из них может обладать особым статусом *первого обработчика*, который означает, что элемент реагирует на ввод текста с клавиатуры или другие действия (скажем, встряхивание устройства для отмены операции).

Когда пользователь нажимает элемент управления, который может получать статус первого обработчика, этому элементу отправляется сообщение `resignFirstResponder`. До того, как другой элемент станет первым обработчиком или текущему элементу будет отправлено сообщение `resignFirstResponder`, этот элемент будет сохранять указанный статус и получать ввод с клавиатуры и встряхивания устройства.

Когда элемент ввода текста (например, текстовое поле) становится первым обработчиком, на экране появляется клавиатура. Пока текущим первым ответчиком остается элемент, получающий текстовый ввод, клавиатура остается на экране. В конце `addTask`: мы приказываем текстовому полю отказаться от статуса первого обработчика, в результате чего клавиатура исчезает с экрана.

Постройте и запустите приложение. Теперь вы можете вводить новые задачи!

Сохранение задач

В `iTahDoodle` осталось реализовать последнюю функцию. Естественно, когда пользователь закрывает приложение, он рассчитывает на то, что список задач будет сохранен для использования в будущем.

Когда приложение Cocoa Touch завершается или переходит в фоновый режим, оно отправляет своему делегату сообщение из протокола `UIApplicationDelegate`, чтобы делегат мог заняться делом и корректно от- реагировать на эти события. В файле *BNRAppDelegate.m* заполните заглушки двух методов делегата, обеспечивающих сохранение списка:

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
// этот метод вызывается только в iOS 4.0+
// сохранение массива tasks на диске
    [tasks writeToFile:docPath() atomically:YES];
}
- (void)applicationWillTerminate:(UIApplication *)application
{
// Этот метод вызывается только в версия iOS до 4.0
// Сохранение массива tasks на диске
    [tasks writeToFile:docPath() atomically:YES];
}
```

Постройте и запустите готовое приложение. Это упражнение всего лишь дает первое представление о программировании для iOS. Вам предстоит узнать намного, намного больше.

Для самых любопытных: как насчет main()?

В самом начале своего знакомства с C и Objective-C вы узнали, что точкой входа в код программы является функция `main()`. Это утверждение абсолютно справедливо и для программирования Cocoa/Cocoa Touch, хотя в приложениях Cocoa и Cocoa Touch эта функция редактируется крайне редко. Чтобы понять, почему, откройте файл *main.m*:

```
return UIApplicationMain(argc, argv, nil, NSStringFromClass([BNRAppDelegate
class]));
```

Прямо скажем, не впечатляет. Всего одна строка реального кода?

Функция `UIApplicationMain()` создает объекты, необходимые для работы вашего приложения. Сначала она создает единственный экземпляр класса `UIApplication`. Далее создает экземпляр класса, определяемого четвертым (последним) аргументом, и назначает его делегатом приложения. Делегату будут отправляться различные сообщения: о нехватке памяти, о завершении или переходе в фоновый режим приложения, о завершении процесса запуска.

Так функция `main()` связывается с `application:didFinishLaunchingWithOptions:` и кодом вашего приложения.

28. Первое приложение Cocoa

В этой главе мы создадим `TahDoodle` - настольное приложение Cocoa. Как и `TahDoodle`, оно представляет собой простой список задач и хранит данные в списке свойств; однако между двумя приложениями существуют некоторые различия. В приложении iOS мы использовали экземпляры `UITableView`, `UITextField` и `UIButton`. В настольной версии список задач будет размещаться в элементе `NSTableView`, где и будет редактироваться напрямую. Также в приложении используется элемент `NSButton`, который вставляет в таблицу строку для ввода новой задачи.

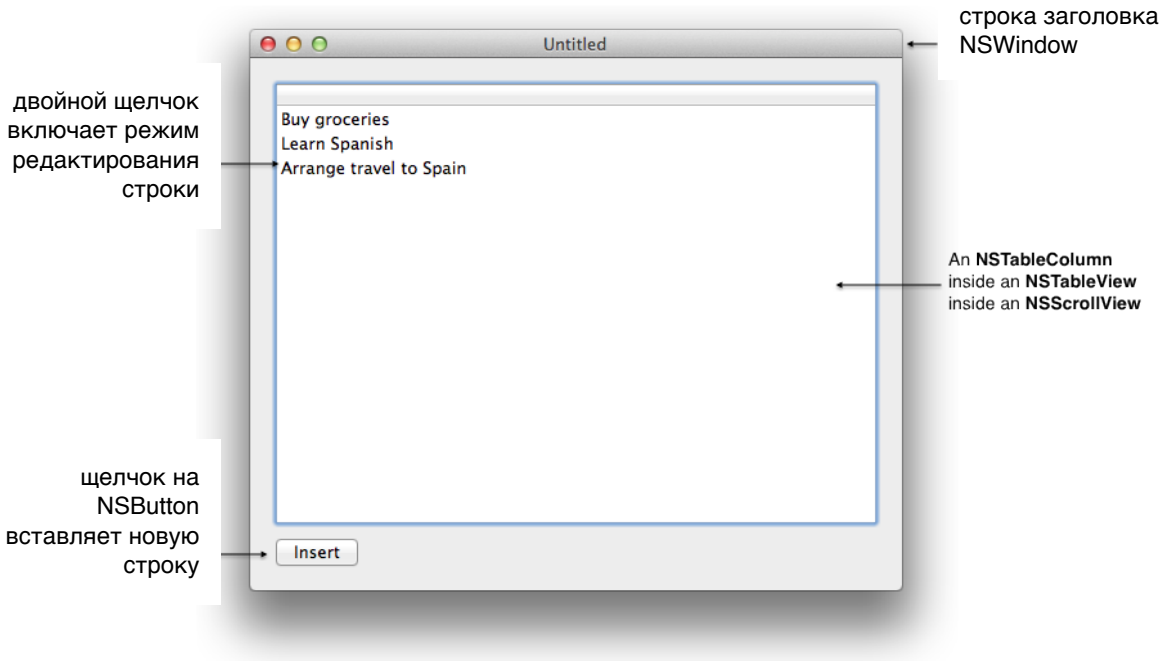


Рис. 28.1, Приложение TahDoodle

Кроме того, в последней главе пользовательский интерфейс приложения строился на программном уровне. В этой главе мы воспользуемся визуальным инструментом Interface Builder, включенным в Xcode, для создания, настройки и связывания элементов пользовательского интерфейса.

В Xcode выполните команду `File→New→NewProject...`. В секции Mac OS X щелкните на строке `Application`. Выберите в открывшемся списке шаблон Cocoa `Application` и присвойте проекту имя `TahDoodle`. Приложение `TahDoodle` является документным, то есть пользователь может держать открытыми сразу несколько списков задач. Поле Document Extension содержит расширение файла, используемое при сохранении документов (списков задач) на диске. Укажите в этом поле, что ваши файлы данных будут использовать расширение `tdl`. Приложение `TahDoodle` не использует Core Data и модульные тесты ему тоже не нужны.

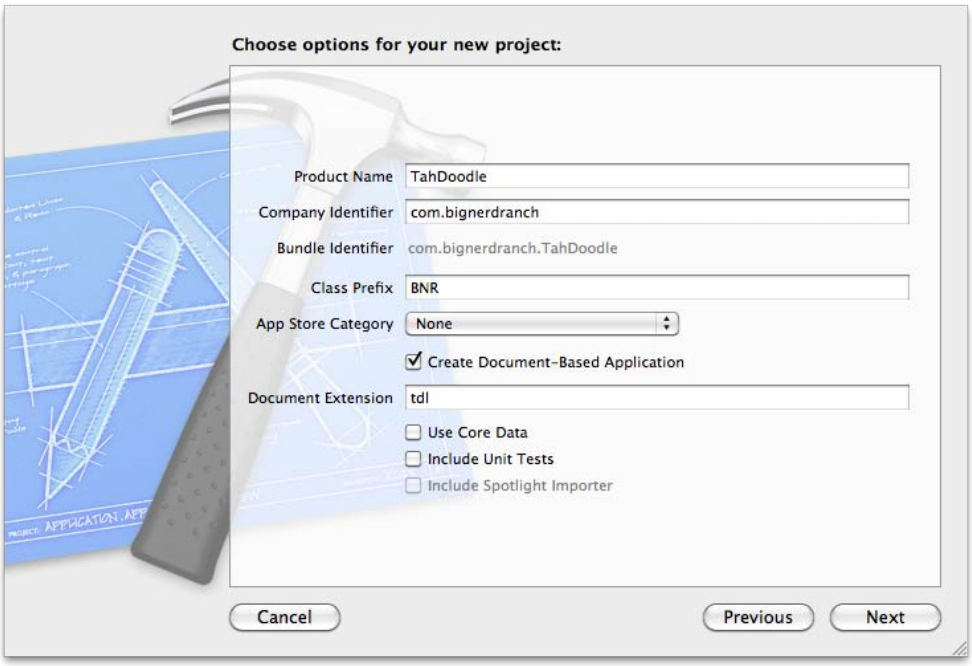


Рис. 28.2. Создание нового приложения Cocoa

Редактирование файла BNRDocument.h

Откройте файл *BNRDocument.h*, включите в него метод и две переменные экземпляра: `todoItems` - изменяемый массив строк, а `itemTableView` - указатель на объект `NSTableView`, в котором отображаются строки из `todoItems`. Также объявите, что `BNRDocument` поддерживает протокол `NSTableViewDataSource`.

```
#import <Cocoa/Cocoa.h>
@interface BNRDocument : NSDocument <NSTableViewDataSource> {
    NSMutableArray *todoItems;
    IBOutlet NSTableView *itemTableView;
}
- (IBAction)createNewItem:(id)sender;
@end
```

Обратите внимание: у кнопки `Insert` нет переменной экземпляра. (Вскоре вы поймете, почему.) Тем не менее этой кнопке назначено действие - метод `createNewItem:`.

В предыдущей главе приемником действия кнопки был экземпляр класса делегата приложения, `BNRAppDelegate`. Документно-базированное приложение не имеет объекта делегата приложения, а в основу его архитектуры заложен субкласс `NSDocument`. Для приложения `TahDoodle` это класс `BNRDocument`.

В Документно-базированном приложении могут одновременно существовать несколько экземпляров объектов документов. Следовательно, во время выполнения

TahDoodle могут существовать сразу несколько экземпляров BNRDocument (несколько списков задач). Каждый экземпляр имеет собственное табличное представление, кнопку, массив tasks и окно. Каждый экземпляр реагирует на сообщения независимо от других, и каждый экземпляр является приемником своей кнопки.

Во введенных вами объявлениях также появились два новых обозначения: IBOutlet и IBAction. Они сообщают Xcode:. Это указатель (IBOutlet) или метод действия (IBAction), для связывания которых разработчик будет использовать Interface Builder - вместо связывания на программном уровне».

Знакомство с Interface Builder

В навигаторе проекта найдите и выберите файл с именем *BNRDocument.xib*. Когда в навигаторе проекта выбирается файл с расширением *.xib* (XML-документ Interface Builder), в панели редактора открывается Interface Builder макетной сеткой.

В данный момент на сетке находится всего один объект представления - объект окна. Это экземпляр NSWindow, к которому мы вскоре добавим другие объекты представлений.

В правом верхнем углу окна Xcode щелкните на правой кнопке View, что-бы открыть панель Utlities. Найдите в верхней части панели Utilities кнопку и щелкните на ней, чтобы открыть панель *инспектора*.

Основной инспектор дополнительно делится на несколько инспекторов. В этой главе мы используем инспекторов атрибутов, размеров и связей.

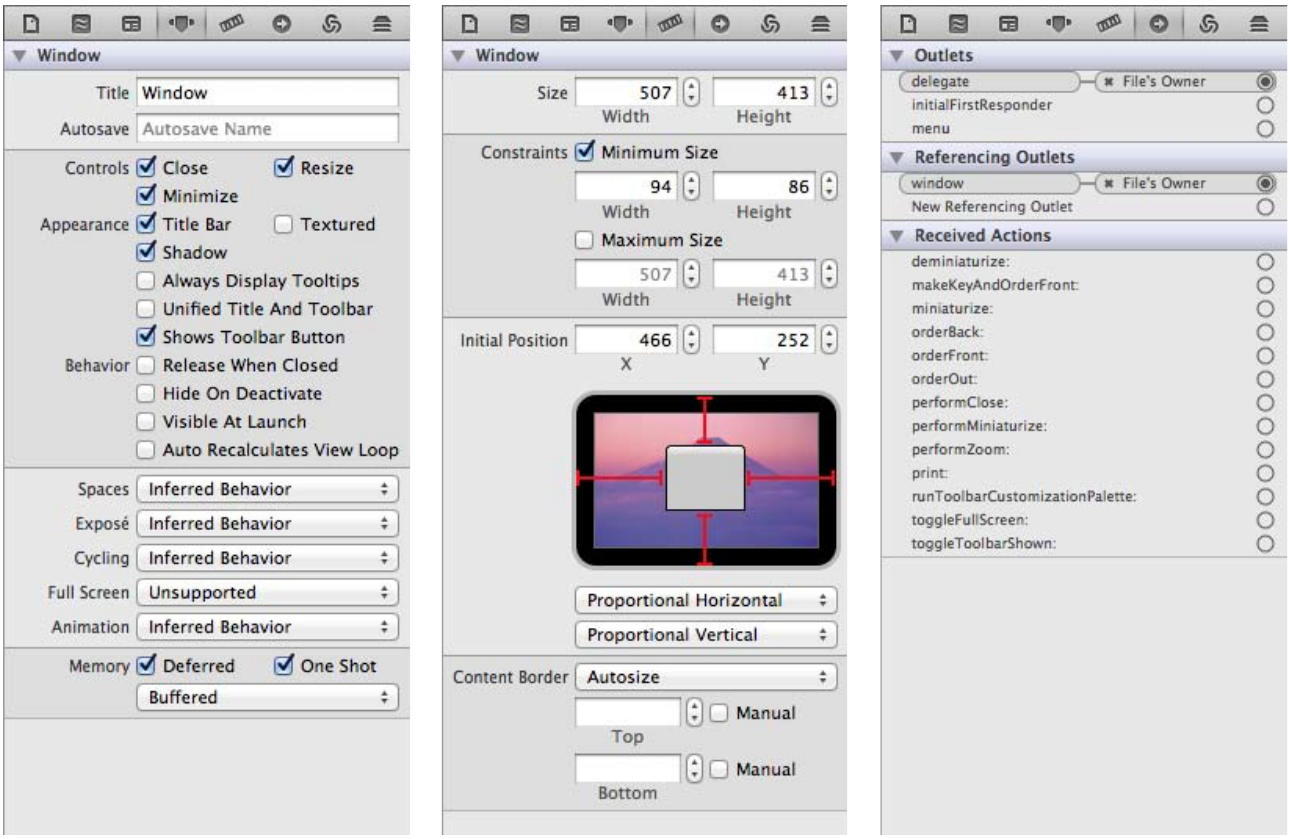


Рис. 28.3. Инспекторы атрибутов, размеров и связей

В нижней части панели `Utilities`, под инспектором, находится *библиотека*, также разделенная на вкладки. Выберите кнопку , чтобы открыть библиотеку объектов. В ней содержится список всех типов объектов, известных Interface Builder, и здесь же размещаются интерфейсные элементы, которые должны перетаскиваться разработчиком в объект окна.

В нижней части библиотеки расположено поле поиска. Введите в нем строку `table`, чтобы отфильтровать список объектов. Первый элемент, Table View, представляет экземпляр класса `NSTableView`. Вы можете щелкнуть на нем, чтобы просмотреть подробную информацию.

Редактирование файла BNRDocument.xib

Перетащите экземпляр `NSTableView` из библиотеки объектов в объект окна. Измените размеры таблицы так, чтобы она закрывала большую часть окна, но оставьте место внизу для кнопки.

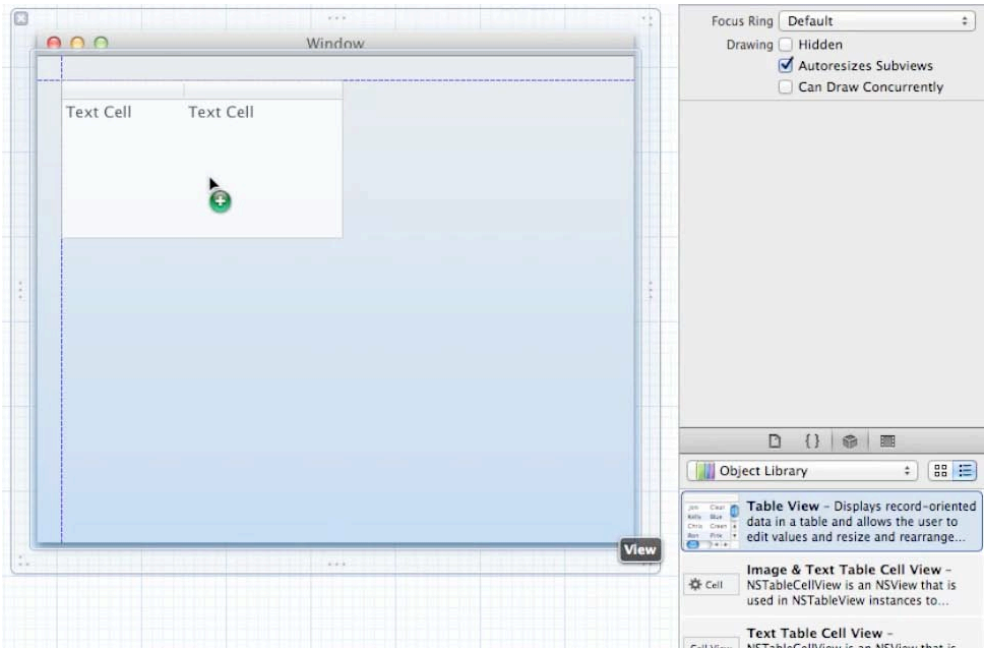


Рис. 28.4. Добавление NSTableView

Обратите внимание: края объекта автоматически выравниваются по направляющим при приближении к краям окна или других объектов. Эти направляющие обеспечивают выравнивание объектов представлений в соответствии с рекомендациями «Human Interface Guidelines» (HIG) фирмы Apple - правилами, которые должен соблюдать каждый разработчик при проектировании пользовательских интерфейсов для Mac. Также существуют документы HIG для iPhone и iPad. Все документы HIG приведены в документации разработчика.

Теперь мы зададим некоторые атрибуты табличного представления в инспекторе атрибутов. Обычно для этого разработчик щелкает на объекте на

макетной сетке, а контекст инспектора изменяется для отображения атрибутов выбранного объекта. добраться до атрибутов экземпляра `NSTableView` сложнее. Объект табличного представления, перетащенный в окно, в действительности представляет собой набор вложенных объектов: `NSScrollView`, `NSTableView` и один или несколько экземпляров `NSTahleColumn`. Чтобы добраться до нужного объекта этого набора, удерживайте нажатыми клавиши `Control` и `Shift` при щелчке на табличном представлении. Под курсором появляется список объектов, из которого можно выбрать нужный вам. Выберите объект `NSTableView`.

В инспекторе атрибутов настройте табличное представление так, чтобы оно состояло из одного столбца. Затем вернитесь в редактор, выберите заголовок табличного представления и растяните единственный столбец до полной ширины табличного представления.

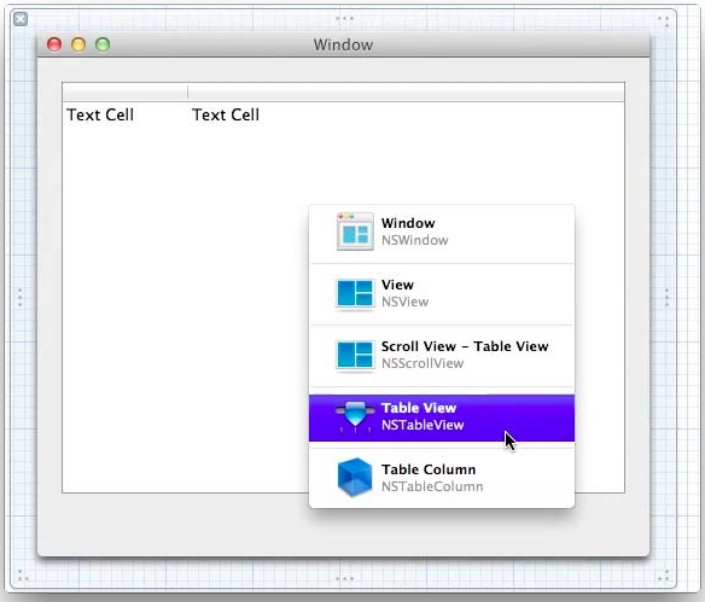


Рис. 28.5. Выбор представления из стопки

Мы аккуратно расставили представления по местам, но что произойдет, когда пользователь изменит размер окна приложения? Необходимо позаботиться о том, чтобы табличное представление изменяло размеры в окне. Вообще говоря, изменяться вместе с окном будет прокручиваемое представление, которое содержит табличное представление. Щелкните на табличном представлении с нажатыми клавишами `Control` и `Shift`, выберите `NSScrollView` из списка. В верхней части панели инспектора щелкните на кнопке  чтобы открыть инспектор размеров. Найдите в нем раздел `Autosizing`.

Рис 28.6 Изменение ширины столбца

Этот раздел содержит маску автоматического изменения размеров - элемент управления, позволяющий настроить реакцию выбранного объекта представления на измерение размеров суперпредставления. Представления, как и классы, образуют иерархию, поэтому каждое представление может иметь суперпредставления и субпредставления. Суперпредставлением `NSScrollView` является экземпляр `NSWindow`.

Маска автоматического изменения размеров состоит из четырех якорей в виде буквы I и двух *пружин* в виде двусторонних стрелок. Выбор одной из пружин позволяет выбранному объекту представления расширяться в заданном направлении при расширении суперпредставления. Выбор якоря привязывает выбранный объект представления к заданному краю его суперпредставления. К счастью, справа от маски автоматического изменения размеров отображается удобная анимация, по которой можно составить представление о том, как будет работать выбранный объект представления с заданной комбинацией якорей и пружин.

Выберите в маске автоматического изменения размеров все четыре якоря и обе пружины. Тем самым вы привязываете все четыре стороны прокручиваемого представления к соответствующим краям окна и позволяете прокручиваемому представлению (и содержащемуся в нем табличному представлению) расширяться по горизонтали и вертикали при изменении размеров окна.

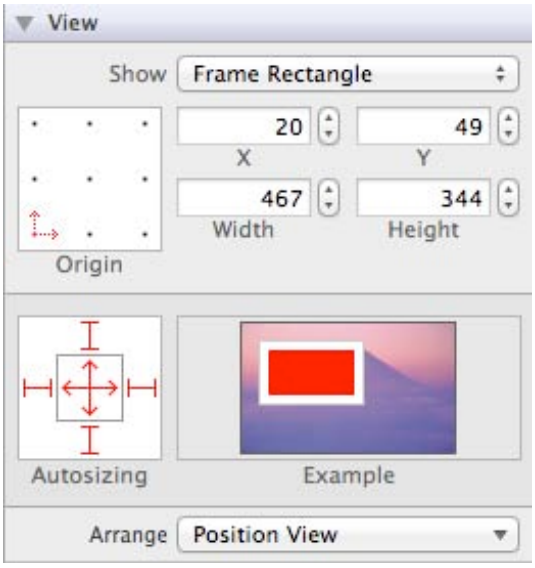


Рис. 28.7. Настройка маски автоматического изменения размеров

Теперь перейдем от табличного представления к кнопке. Вернитесь в библиотеку объектов. Найдите экземпляр `NSButton` и перетащите его в объект окна. Вы можете выбрать любой из стилей кнопок, представленных в библиотеке; классический вариант - прямоугольная кнопка с закругленными краями (Rounded Rect Button). После того как кнопка будет перетащена в окно, вы можете изменить надпись на кнопке; для этого сделайте двойной щелчок на тексте кнопки и введите нужный текст. В нашем приложении следует ввести текст `Insert`.

Наконец, в инспекторе размеров задайте маску автоматического изменения размеров так, чтобы кнопка была привязана к левому нижнему углу окна и сохраняла свои текущие размеры.

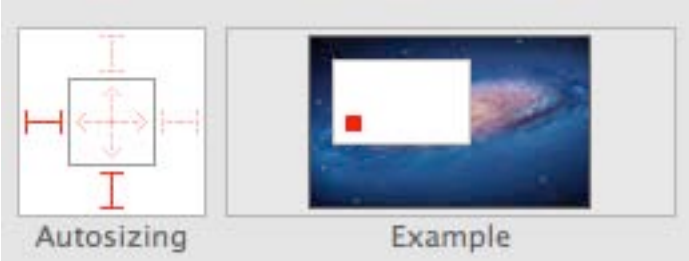


Рис. 28.8. Маска автоматического изменения размеров для кнопки Insert

Итак, мы создали два объекта представлений, необходимых для приложения `TahDoodle`: `NSTableView` и `NSButton`, а также настроили параметры этих объектов. Для табличного представления было задано количество столбцов, а для кнопки - текст надписи. Мы также позаботились об их правильном изменении размеров и позиционировании при изменении размеров окна.

Когда вы делаете двойной щелчок на `NSButton` и вводите текст надписи, результат эквивалентен тому, который достигался в предыдущей главе добавлением следующей строки кода:

```
[insertButton setTitle:@"Insert"
                  forState:UIControlStateNormal];
```

Итак, в каких случаях следует использовать Intecface Builder, а когда лучше создавать представления на программном уровне? В простых случаях подойдет любое решение. Мы могли построить интерфейс `iTahDoodle` с использованием XIB. Однако в общем случае чем сложнее интерфейс, тем больше причин для использования Interface Builder.

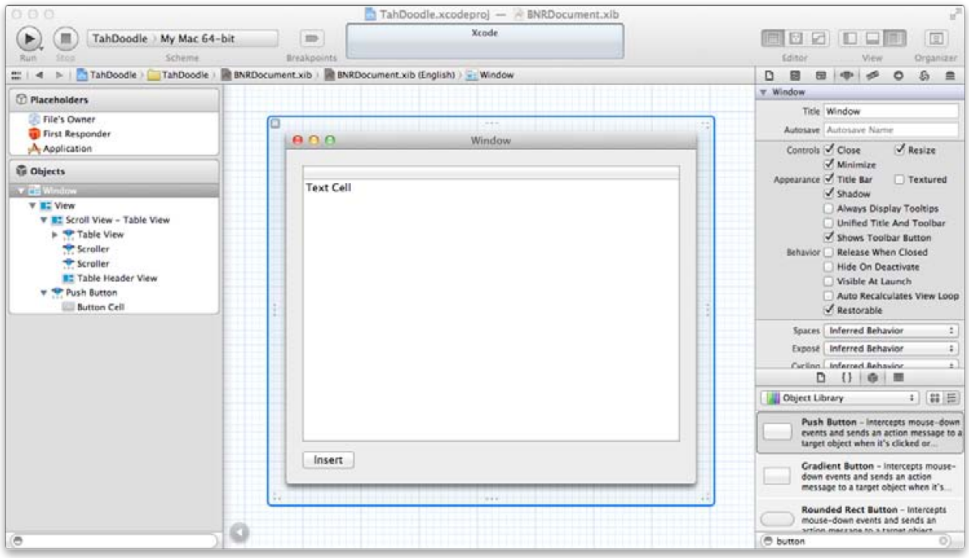


Рис. 28.9. BNRDocument.xib с настроенными представлениями

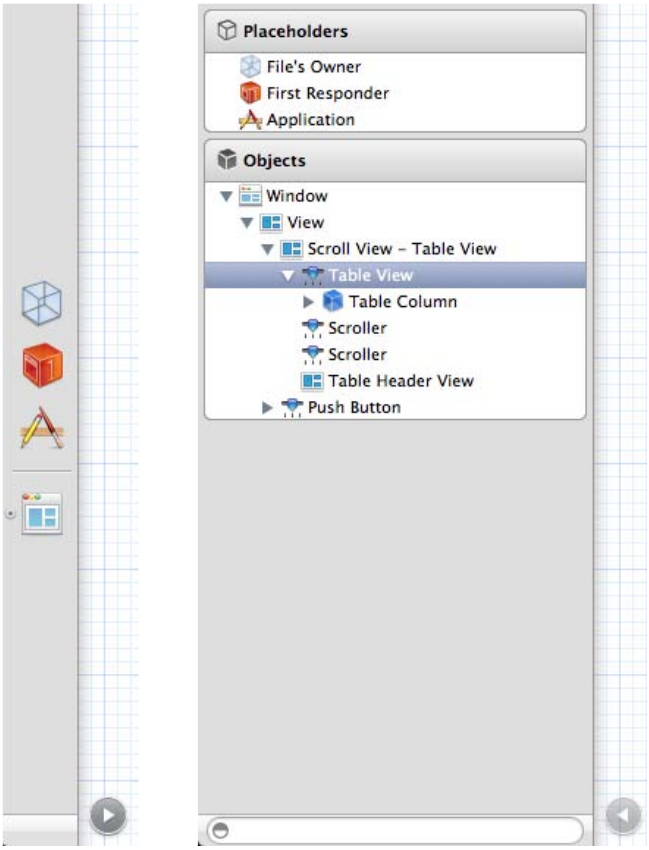
Связывание представлений

Возможности Interface Builder не ограничиваются созданием и настройкой представлений. Разработчик также может объединять объекты представлений с кодом приложения - в частности, определить пары «приемник/действие» и присвоить указатели.

Слева от макетной сетки список «заместителей» (`Placeholders`) и объектов. Он называется *сводкой документа* (`document outline`). (Если вы видите только набор значков без надписей, щелкните на кнопке , чтобы открыть расширенный список.)

Найдите в секции `Placeholders` сводки строку File's Owner. Заместитель обозначает некий объект. который не может быть задан до стадии выполнения. Так, File's Owner обозначает объект, который загрузит данный файл XIB для формирования своего пользовательского интерфейса. В нашем случае File's Owner представляет экземпляр `BNRDocument`.

Выберите кнопку Insert в редакторе и перетащите ее на строку File's Owner, удерживая клавишу Control.



Рис, 28.10. Сводка документа в свернутом и развернутом виде

Когда вы отпускаете кнопку мыши, на экране появляется маленькое окно с перечнем всех возможных связей. Выберите `createNewItem`: - тем самым вы выбираете действие, которое должно срабатывать при нажатии кнопки (рис. 28.12)

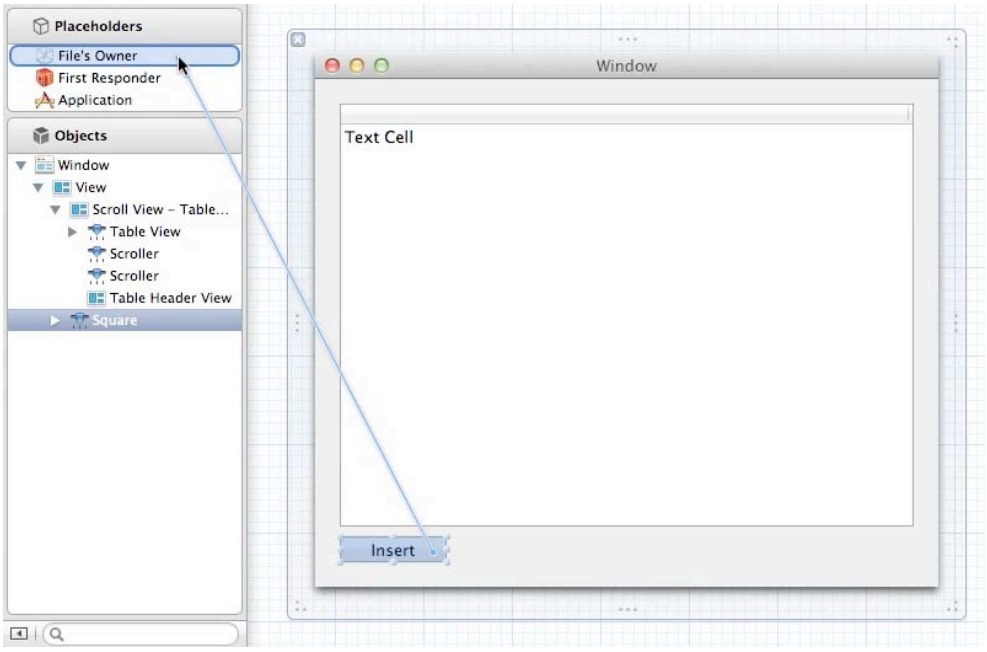


Рис. 28.11. Создание связи

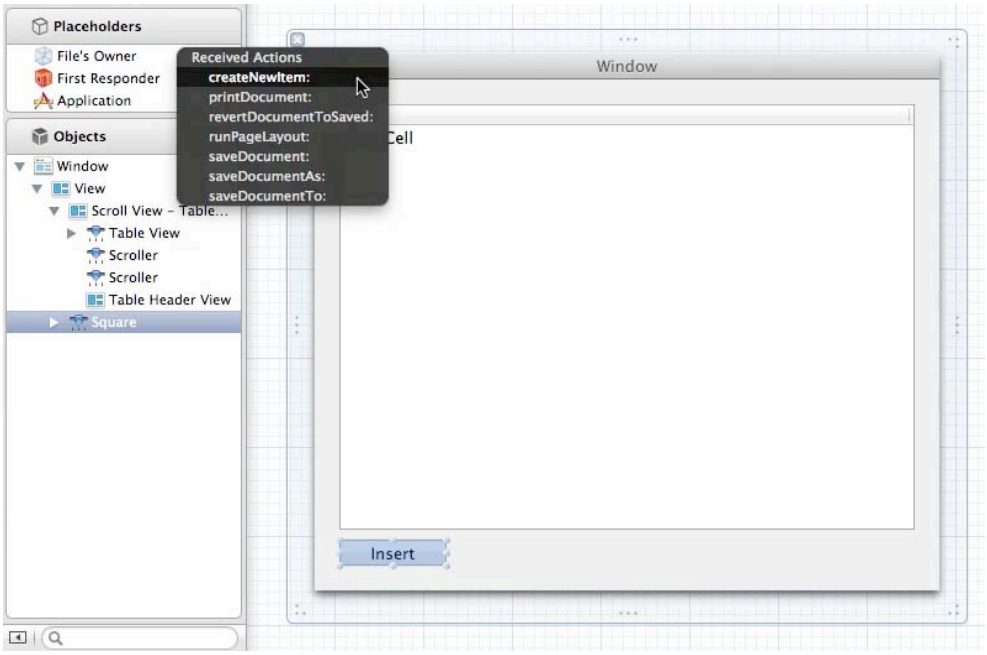


Рис. 28.12. Выбор действия

Мы настроили пару «приемник/действие». Результат эквивалентен следующему фрагменту кода в `iTahDoodle`:

```
[insertButton addTarget:self
          action:@selector(addTask:)
          forControlEvents:UIControlEventTouchUpInside];
```

Теперь свяжите ссылку `itemTableView` класса `BNRDocument`. Перетащите с нажатой клавишей Control строку File's Owner (заменяющую `BNRDocument`) на

NSTableView и отпустите кнопку мыши. Выберите единственный вариант, который будет предложен: itemTableView.

Наконец, щелкните на табличном представлении с нажатыми клавишами Control и Shift, и выберите в списке NSTableView. Перетащите с нажатой клавишей Control табличное представление на строку File's Owner и выберите в открывшемся списке связей вариант dataSource.

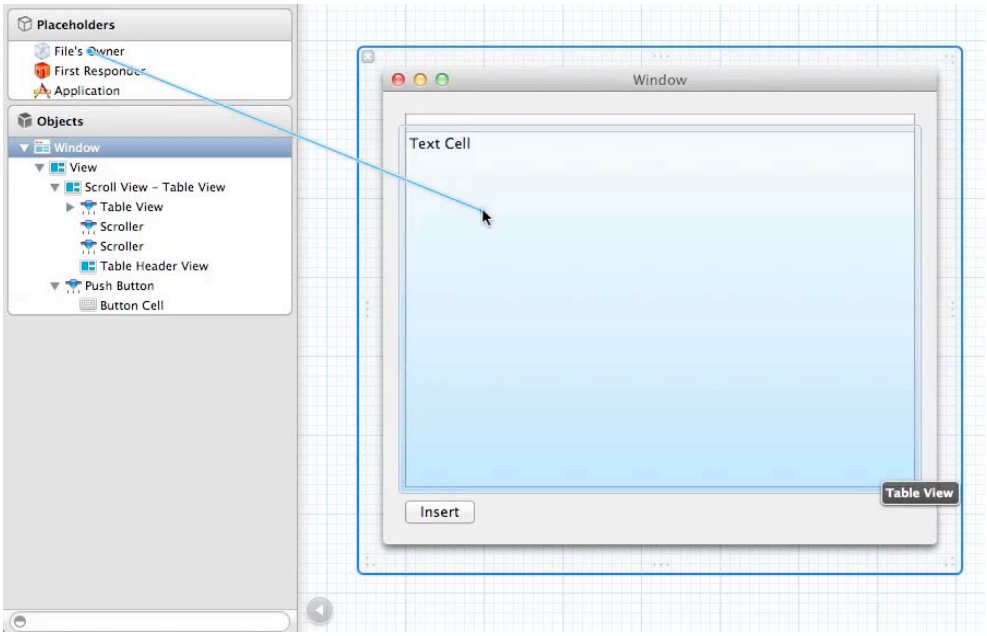


Рис. 28.13. Создание других связей

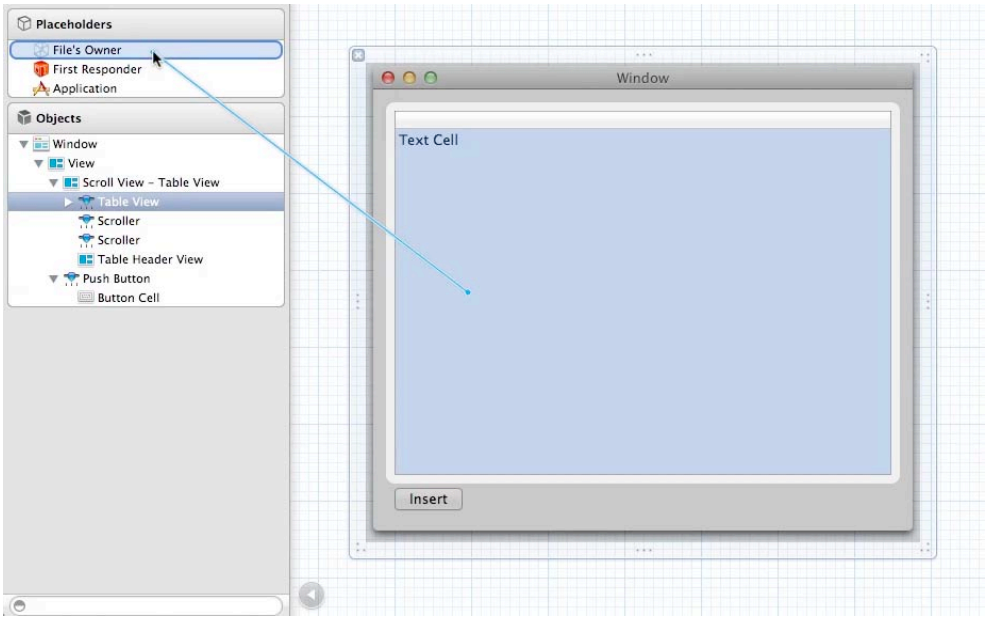


Рис. 28.14, Связывание источника данных табличного представления

Итак, что же мы только что сделали? Мы присвоили значения указателей. Класс `BNRDocument` объявляет указатель на `NSTableView`. Мы сообщили Interface Builder, что этот указатель должен указывать на конкретный экземпляр `NSTableView`, который был перетащен в это окно. Кроме того, табличное представление содержит ряд собственных указателей - таких, как `dataSource`. Мы сообщили Interface Builder, что указатель `dataSource` табличного представления должен указывать на наш экземпляр `BNRDocument`. Результат эквивалентен программному назначению источника данных табличного представления в `iTahDoodle`.

```
[taskTable setDataSource:self];
```

Вводя ключевые слова `IBOutlet` и `IBAction`, мы помечаем эти ссылки и действия для Interface Builder, говоря: «Когда я пытаюсь связать указатель в IB, этот объект должен быть включен в список возможных связей!» Во время написания кода Interface Builder следит за ключевыми словами `IBOutlet` и `IBAction` и знает, какие варианты следует предлагать при создании связей.

Фактические определения `IBOutlet` и `IBAction` тривиальны:

```
#define IBAction void
#define IBOutlet
```

Из того, что говорилось ранее о директиве `#define`, становится ясно, что ключевое слово `IBAction` заменяется на `void` еще до того, как его увидит компилятор, а `IBOutlet` полностью исчезает из программы. Таким образом, на стадии компилирования все метки `IBOutlet` удаляются полностью, а ключевые слова `IBAction` заменяются на `void`, потому что действия, инициируемые элементами управления пользовательского интерфейса, не должны иметь возвращаемого значения.

Учтите, что Элементы пользовательского интерфейса на базе XIB не создаются явно, как это делалось при программном создании интерфейса. Они создаются автоматически при загрузке XIB во время выполнения.

Наконец обратите внимание на отсутствие указателя на кнопку. Дело в том, что объекту нужны переменные экземпляров только для тех объектов, которым он собирается отправлять сообщения. Кнопка должна отправлять сообщения экземпляру `BNRDocument`, поэтому мы связали ее действие с приемником. Однако экземпляр `BNRDocument` не собирается отправлять сообщения кнопке, поэтому указатель на кнопку ему не нужен.

Снова о MVC

Итак, пользовательский интерфейс приложения готов. Давайте взглянем на диаграмму объектов данного проекта:

NSDocument - суперкласс, от которого наследует класс BNRDocument,- вообще интересная штука. На первый взгляд он является объектом модели. Но из справочного описания класса NSDocument становится видно, что это скорее контроллер, чем что-либо другое. NSDocument координирует различные дисковые операции и связывается напрямую с представлениями, обеспечивающими ввод данных пользователем. При создании BNRDocument как субкласса BNRDocument мы добавили указатели на реальные объекты моделей (массив NSMutableArray объектов NSString).

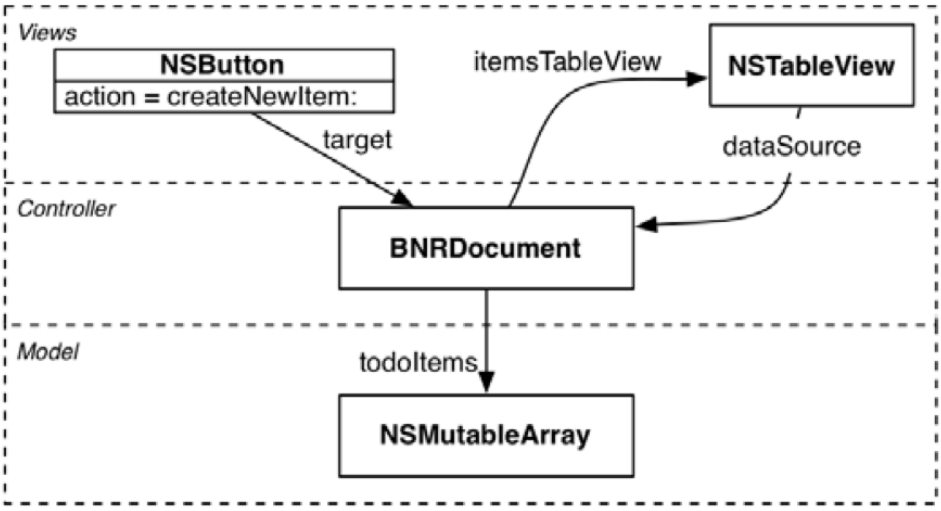


Рис. 28.15. Диаграмма объектов TahDoodle

Редактирование файла BNRDocument.m

Теперь, когда пользовательский интерфейс приложения создан и настроен и в нем созданы все связи, пришло время вернуться к написанию кода. Щелкните на файле *BNRDocument.m* в навигаторе проекта, чтобы снова открыть его в редакторе, и реализуйте createNewItem:.

```
#import "BNRDocument.h" @implementation BNRDocument #pragma mark - NSDocument
Overrides
- (NSString *)windowNibName
{
    return @"BNRDocument";
}

#pragma mark - Actions
- (IBAction)createNewItem:(id)sender
{
    // если массив еще не существует, создаем его для хранения новой задачи
    if (!todoItems) {
        todoItems = [NSMutableArray array];
    }
    [todoItems addObject:@"New Item"];
```

```
    // -reloadData приказывает табличному представлению обновится
    // и запросить у источника данных (которым в данном случае
    // является объект BNRDocument) новые данные для отображения
    [itemTableView reloadData];
    // -updateChangeCount: сообщает приложению, содержит ли документ
    // несохраненные изменения. NSChangeDone помечает документ как
несохраненный
    [self updateChangeCount:NSChangeDone];
}
```

Теперь реализуйте обязательные методы источника данных табличного представления (определяемые протоколом `NSTableViewDataSource`):

```
#pragma mark Data Source Methods
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tv
{
    // табличное представление предназначено для отображения todoItems,
    // поэтому количество строк в табличном представлении совподает
    // с колличеством объектов в массиве.
    return [todoItems count];
}
- (id)tableView:(NSTableView *)tableView
      objectValueForTableColumn:(NSTableColumn *)tableColumn
                            row:(NSInteger)row
{
    // возвращает элемент todoItems, соответствующей ячейки
    // которая должна отоброжаться в табличном представлении
    return [todoItems objectAtIndex:row];
}
- (void)tableView:(NSTableView *)tableView
        setObjectValue:(id)object
        forTableColumn:(NSTableColumn *)tableColumn
                   row:(NSInteger)row
{
    // когда пользователь изменяет задачу в табличном представлении,
    // массив todoItems необходимо обновить
    [todoItems replaceObjectAtIndex:row withObject:object];
    // А затем пометить документ как содержащий несохраненные изминения.
    [self updateChangeCount:NSChangeDone];
}
```

Постройте и запустите программу Окно `TahDoodle` появляется на экране, вы можете добавлять и изменять задачи. Однако в приложении отсутствует одна важная возможность: сохранение и повторная загрузка списка задач. Чтобы реализовать ее, необходимо переопределить следующие методы, унаследованные от `NDocument` - суперкласса `BNRDocument`:

Обратите внимание: паша реализация метода получает `NSError**`. В данном случае мы просто возвращаем объект `NSError`, сгенерированный `propertyListWithData: options:format:error:`, но также в зависимости от природы ошибки можно было бы создать и вернуть новый объект `NSError`.

Снова постройте и запустите приложение. Теперь оно умеет сохранять и загружатъ списки задач.

```
- (NSData *)dataOfType:(NSString *)typeName
                 error:(NSError **)outError
{
    // This method is called when our document is being saved
    // We are expected to hand the caller an NSData object wrapping our data
    // so that it can be written to disk

    // If there's no array, we'll write out an empty array for now
    if (!todoItems) {
        todoItems = [NSMutableArray array];
    }

    // Pack our todoItems array into an NSData object
    NSData *data = [NSPropertyListSerialization
                               dataWithPropertyList:todoItems
                                             format:NSPropertyListXMLFormat_v1_0
                                            options:0
                                              error:outError];

    // return our newly-packed NSData object
    return data;
}

- (BOOL)readFromData:(NSData *)data
              ofType:(NSString *)typeName
               error:(NSError **)outError
{
    // This method is called when a document is being loaded
    // We are handed an NSData object and expected to pull our data out of it

    // Extract our todoItems
    todoItems = [NSPropertyListSerialization
                            propertyListWithData:data
                                         options:NSPropertyListMutableContainers
                                          format:NULL
                                           error:outError];

    // return success or failure depending on success of the above call
    return (todoItems != nil);
}
```

Упражнения

Добавьте кнопку удаления выбранной задачи.

Просмотрите справочное описание класса `NSError` в документации разработчика и создайте экземпляры `NSError`, возвращаемые в случае неудачной попытки сохранения или загрузки (о чем упоминается в конце главы).

V

Расширенные возможности Objective-C

Сейчас вы уже знаете Objective-C в объеме, достаточном для несложного программирования iOS или Cocoa. Но не торопитесь! В следующих главах представлены многие приемы и концепции, которые сильно пригодятся вам в первый год самостоятельного программирования на Objective-C.

29. init

Класс NSObject содержит метод с именем init. После выделения памяти новому экземпляру отправляется сообщение init, чтобы экземпляр мог инициализировать сбой переменные экземпляров реальными значениями. Таким образом, alloc выделяет память для объекта, а init готовит объект к работе. Использование init выглядит примерно так:

```
NSMutableArray *things = [[NSMutableArray alloc] init];
```

Следует учитывать, что init является методом экземпляра, который возвращает адрес инициализированного объекта. Он обеспечивает инициализацию NSObject. В этой главе вы узнаете, как пишутся инициализаторы.

Написание методов init

Создайте новый проект: программу командной строки Foundation с именем Appliances. В этой программе мы создадим два класса: Appliance и OwnedAppliance (субкласс Appliance). Класс Appliance (устройство) содержит переменные productName (название продукта) и voltage (напряжение). Экземпляр OwnedAppliance также будет содержать множество с именами владельцев.

Создайте новый файл: субкласс NSObject с именем Appliance. В файле *Appliance.h* создайте переменные экземпляров и объявления свойств для productName и voltage:

```
#import <Foundation/Foundation.h>
@interface Appliance : NSObject {
NSString *productName;
int voltage;
}
@property (copy) NSString *productName;
@property int voltage;
@end
```

(Атрибут свойств copy будет рассмотрен в главе 30.)

В файле *Appliance.m* синтезируйте методы доступа к переменным экземпляров:

```
#import "Appliance.h"
@implementation Appliance
@synthesize productName, voltage;
```

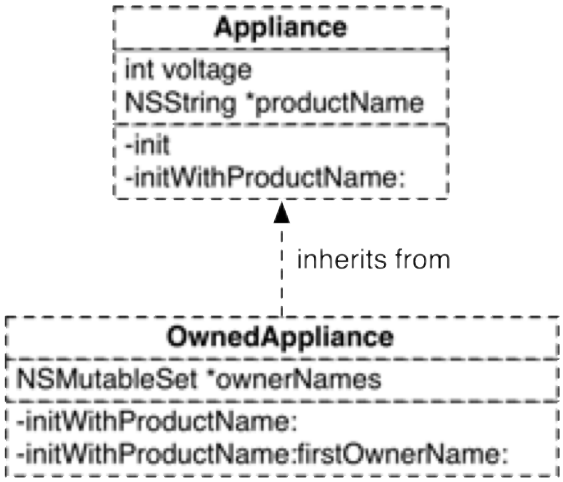


Рис. 29.1. Класс Appliance и его субкласс OwnedAppliance

Экземпляры `Appliance` создаются следующим образом:

```
Appliance *a = [[Appliance alloc] init];
```

Так как `Appliance` не реализует метод `init`, будет выполнен метод `init`, определенный в `NSObject`. Когда это произойдет, все переменные экземпляров, специфические для `Appliance`, будут обнулены. Соответственно переменная `productName` нового экземпляра Appliance будет равна `nil`, а переменная `voltage` будет равна нулю.

Простейший метод init

В некоторых случаях инициализация переменных экземпляров нулями может работать хорошо. В других случаях экземпляры класса должны начинать свое существование с переменными экземпляров, инициализированными не нулевыми значениями.

Допустим, в каждом экземпляре `Appliance` переменная `voltage` должна инициализироваться значением 120. В файле *Appliance.m* добавьте свою реализацию `init`, переопределяя тем самым метод `init` класса `NSObject`.

```
- (id)init {
    // вызов метода init класса NSObject
    self = [super init];
    // присваивание начального значение voltage?
    voltage = 120;
    // возвращение указателя на новый объект object
    return self;
}
```

Теперь при создании нового экземпляра `Appliance` переменная `voltage` по умолчанию инициализируется значением 120. (Учтите, что это никак не влияет на

работу методов доступа. После того как экземпляр будет инициализирован, переменная может быть изменена обычным образом, то есть с использованием setVoltage:.)

Обратите внимание на вызов метода init суперкласса; он инициализирует переменные экземпляра, объявленые в суперклассе, и возвращает указатель на инициализированный объект. В большинстве случаев такая схема работает безупречно. Тем не менее некоторые классы содержат аномальные методы init. Аномалии делятся на две разновидности:

- Метод init выполняет какую-то хитрую оптимизацию, уничтожает исходный объект, выделяет память для другого объекта и возвращает новый объект.

- Попытка выполнения init завершается неудачей, метод уничтожает объект и возвращает nil.

В первом случае фирма Apple требует, чтобы self был присвоен указатель на объект, возвращаемый методом init суперкласса. Мы делаем это в первой строке своего метода init.

Во втором случае Apple рекомендует проверить, что инициализатор суперкласса возвращает действительный объект, а не nil. И это понятно - бессмысленно выполнять пользовательскую настройку несуществующего объекта. Измените метод init в соответствии с рекомендациями Apple:

```
- (id)init {
    // вызов метода init класса NSObject
    self = [super init];

    // метод вернул значение, отличное от nil?

if (self) {
       // присваивание начального значения voltage

        voltage = 120;

}
    return self;
}
```

Откровенно говоря, такие проверки необходимы только в очень специфических ситуациях, и на практике многие программисты Objective-C пропускают их. Тем не менее в книге мы всегда будем использовать такие проверки, потому что фирма Apple рекомендует именно так подходить к реализации методов init.

Использование методов доступа

Наш метод `init` класса `Appliance` отлично работает, но я хочу показать другую разновидность, которая часто встречается в коде других людей. Обычно я выполняю простое присваивание в методе `init` но многие программисты используют метод доступа. Внесите некоторые изменения в метод `init`:

```
- (id)init {
        // вызов метода init класса NSObject
       self = [super init];

        // метод вернул значение, отличное от nil?
        if (self) {

        // присваивание начального значения voltage

            [self setVoltage:120];

}
        return self;
}
```

В большинстве случаев нет особых причин предпочитать одно решение другому, но зато есть хорошая тема для спора. Сторонник присваивания говорит: «Методы доступа не должны использоваться в `init`! Метод доступа подразумевает, что объект готов к использованию, а состояние-готовности появляется только после завершения `init`». На это сторонник методов доступа отвечает: «Да ладно». В реальном мире это почти никогда не создает проблем. Я использую свой метод доступа повсюду, где задаю значение этой переменной. На самом деле в подавляющем большинстве случаев можно использовать любой из двух способов.

init с аргументами

Иногда нормальная инициализация объекта требует передачи дополнительной информации от метода, который его вызывает. Представьте, что экземпляр `Appliance` не может нормально функционировать без названия (`nil` не в счет). В таком случае необходимо как-то передать инициализатору название, закрепляемое за устройством.

Сделать это в `init` нельзя, потому что `init` не получает аргументов. Значит, необходимо создать новый инициализатор. Тогда создание экземпляра `Appliance` в другом методе будет выглядеть так:

```
Appliance *a = [[Appliance alloc] initWithProductName:@"Toaster"];
```

Новый инициализатор класса `Appliance` - `initWithProductName`: - получает аргумент `NSString`. Объявите новый метод в *Appliance.h*:

```
#import <Foundation/Foundation.h>
@interface App1iance : NSObject {
NSString *productName;
int voltage;
}
@property (copy) NSString *productName;
@property int voltage;
- (id)initWithProductName:(NSString *)pn;
@end
```

Найдите в *Appliance.m* реализацию init. Переименуйте метод в `initWithProductName:` и `productName`, используя переданное значение.

```
- (id)initWithProductName:(NSString *)pn
{
// Вызов метода init класса NSObject
se1f = [super init];

// Метод вернул значение, отличное от nil?
if (self) {
// Инициализация названия продукта
[self setProductName:pn];

// Присваивание начального значения voltage
[self setVoltage:120];
}
return self;
}
```

Прежде чем продолжать, постройте проект и убедитесь в правильности синтаксиса.

Теперь вы можете создать экземпляр `Appliance` с заданным названием. Но если передать файлы *Appliance.h* и *Appliance.m* другому программисту, он может и не понять, что в коде следует вызвать `initwithProductName:`. Что, если он создаст экземпляр `Appliance` стандартным способом?

```
Appliance *a = [[Appliance alloc] init];
```

Такое действие не назовешь неразумным. Ожидается, что экземпляр `Appliance` - субкласса `NSObject` - способен делать все, на что способен экземпляр `NSObject`. А экземпляры `NSObject` реагируют на сообщения `init`.

Однако в данном случае это создает проблемы, потому что приведенная строка кода создает экземпляр Appliance с названием продукта `nil` и нулевым значением `voltage`. А ранее мы договорились, что каждый экземпляр `Application` нормально работает только при значении `voltage`, равном 120, и названии, отличном от `nil`. Как предотвратить подобную неприятность?

Проблема решается просто: в файле *Appliance.m* добавьте метод `init`, вызывающий `initWithProductName:` с названием по умолчанию.

```
- (id)init {
    return [self initWithProductName:@"Unknown"];
}
```

Новая переопределенная версия init всего лишь вызывает метод `initWithProductName:`, который выполняет всю тяжелую работу.

Для тестирования двух инициализаторов нам понадобится метод `description`. Включите его реализацию в *Appliance.m*:

```
- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@: %d volts>", productName, voltage];
}
```

Немного по экспериментируем с классом в *main.m*:

```
#import <Foundation/Foundation.h>
#import "Appliance.h"
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        Appliance *a = [[Appliance alloc] init];
        NSLog(@"a is %@", a);
        [a setProductName:@"Washing Machine"];
        [a setVoltage:240];
        NSLog(@"a is %@", a);
}
return 0;
}
```

Постройте и запустите программу

Продолжим наше изучение инициализаторов. Создайте новый файл - субкласс `Appliance` с именем `OwnedAppliance`.

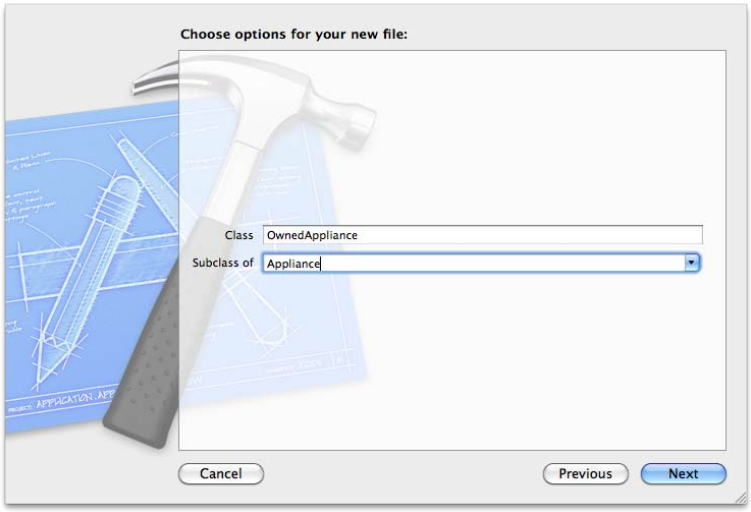


Рис. 29.2. Создание субкласса Appliance

Включите в файл *OwnedAppliance.h* изменяемый набор имен владельцев и три метода.

```
#import "Appliance.h"
@interface OwnedAppliance : Appliance { NSMutableSet *ownerNames;
}
- (id)initWithProductName:(NSString *)pn
           firstOwnerName:(NSString *)n;
- (void)addOwnerNamesObject:(NSString *)n;
- (void)removeOwnerNamesObject:(NSString *)n;
@end
```

Один из объявленых методов - инициализация, получающий два аргумента. Реализуйте методы в файле *OwnedAppliance.m*:

```
#import "OwnedAppliance.h"
@implementation OwnedAppliance
- (id)initWithProductName:(NSString *)pn
           firstOwnerName:(NSString *)n
{
    // вызов инциализатора суперкласса
    self = [super initWithProductName:pn];

if (self) {
        // создание множества для хранения имен владельцев
        ownerNames = [[NSMutableSet alloc] init];
        // имя первго владельца отличного от nil?
        if (n) {
            [ownerNames addObject:n];
        }
    }
    // возвращение указателя на новый объект
    return self;
}
- (void)addOwnerNamesObject:(NSString *)n
{
    [ownerNames addObject:n];
}
- (void)removeOwnerNamesObject:(NSString *)n
{
    [ownerNames removeObject:n];
}
@end
```

Обратите внимание: класс не инициализирует `vo1tage` или `productName`. Этим занимается метод `initWithProductName`: в классе `Appliance`. Когда вы создаете субкласс, обычно необходимо инициализировать только те переменные экземпляров, которые в него добавили вы; а о переменных экземпляров, которые добавил суперкласс, пусть позаботится он сам.

Однако теперь мы сталкиваемся с ситуацией, уже знакомой нам по Appliance и инициализатору его суперкласса init. В какой-то момент один из ваших коллег может создать ужасную ошибку, написав следующую строку кода:

```
OwnedAppliance *a = [[OwnedAppliance alloc] initWithProductName:@"Toaster"];
```

Этот код приведет к выполнению метода initWithProductName: в классе Appliance. Метод ничего не знает о множестве ownerNames, а следовательно, переменная ownerNames будет некорректно инициализирована для данного экземпляра OwnedAppliance.

Решение проблемы тоже остается прежним. Включите в файл *OwnedAppliance.m* реализацию инициализатора суперкласса initWithProductName:, которая вызывает initWithProductName:firstOwnerName: и передает значение по умолчанию для firstOwnerName.

```
- (id)initWithProductName:(NSString *)pn
{
    return [self initWithProductName:pn firstOwnerName:nil];
}
```

Вопрос на засыпку: нужно ли также реализовать init в OwnedAppliance? Нет. На этой стадии следующий код будет нормально работать:

```
OwnedAppliance *a = [[OwnedAppliance alloc] init];
```

Почему? Потому что при отсутствии реализации init в OwnedAppliance эта строка приведет к срабатыванию реализации init из Appliance, которая вызывает [self initWithProductName:@"Unknown"].self - экземпляр OwnedAppliance, поэтому он вызывает версию initWithProductName: класса OwnedAppliance, которая вызывает

```
[self initWithProductName: pn firstOwnerName:nil].
```

В результате образуется цепочка инициализаторов, вызывающих другие иициализаторы.

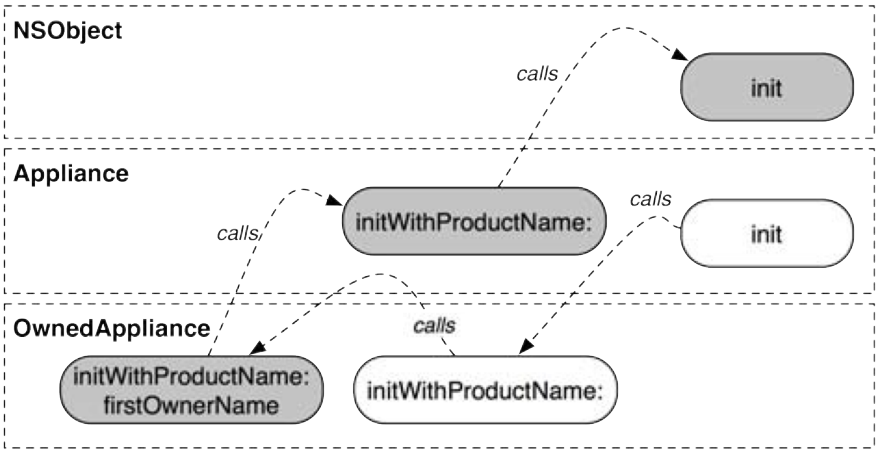


Рис 29.3. Цепочка инициализаторов

На рис. 29.3 один инициализатор для каждого класса выделен темным фоном. Он является *основным инициализатором* для данного класса. Так, init является основным инициализатором для класса `NSObject`, `initWithProductName:` - для класса `Appliance`, а `initWithProductName:firstOwnerName:` - для класса `OwnedAppliance`. Класс содержит только один основной инициализатор. Если класс содержит другие инициализаторы, то их реализация должна вызывать (прямо или косвенно) основной инициализатор.

При создании класса, у которого имя основного инициализатора отличается от имени основного инициализатора суперкласса (как у `Appliance` с `OwnedAppliance`), необходимо документировать это обстоятельство в заголовочном файле. Включите соответствующий комментарий в *Appliance.h*:

```
#import <Foundation/Foundation.h>
@interface Appliance : NSObject {
    NSString *productName;
    int voltage;
}
@property (copy) NSString *productName;
@property int voltage;
// основной инциализатор
- (id)initWithProductName:(NSString *)pn; @end
and in OwnedAppliance.h:
#import "Appliance.h"
@interface OwnedAppliance : Appliance {
    NSMutableSet *ownerNames;
}
// основной инциализатор
- (id)initWithProductName:(NSString *)pn
           firstOwnerName:(NSString *)n;
- (void)addOwnerNamesObject:(NSString *)n;
- (void)removeOwnerNamesObject:(NSString *)n;
@end
```

Итак, мы подходим к правилам, которые должны соблюдаться каждым приличным программистом Objective-C при написании инициализаторов:

- Если класс имеет несколько инициализаторов, только один из них должен выполнять реальную работу. Этот метод называется *основным инициализатором*. Все остальные инициализаторы Должны вызывать основной инициализатор (прямо или косвенно).

- Основной инициализатор вызывает основной инициализатор суперкласса перед инициализацией своих переменных экземпляров.

- Если имя основного инициализатора вашего класса отличается от имени основного инициализатора его супер класса, вы должны переопределить основной инициализатор суперкласса, чтобы он вызывал новый основной инициализатор.

- Если класс содержит несколько инициализаторов, четко укажите в заголовочном файле, какой из них является основным.

Фатальный вызов init

Время от времени встречаются ситуации, в которых безопасное переопределение основного инициализатора суперкласса оказывается невозможным. Допустим, мы создаем субкласс `NSObject` с именем `WallSafe`, а его основным инициализатором является `initWithSecretCode:`. Однако инициализация `secretCode` значением по умолчанию не обеспечивает необходимого уровня безопасности в вашем приложении. А это означает, что используемая схема - переопределение init для вызова основного инициализатора нового класса со значениями по умолчанию - оказывается неприемлемой.

Что же делать? Экземпляр `WallSafe` по-прежнему реагирует на сообщение init. И кто-нибудь может легко использовать конструкцию вида:

```
WallSafe *ws = [[WallSafe alloc] init];
```

Лучшее, что возможно в такой ситуации - такое переопределение основного инициализатора супер класса, которое сообщит разработчику о совершенной ошибке и подскажет ему, как эту ошибку исправить:

```
- (id)init {
    @throw [NSException exceptionWithName:@"WallSafeInitialization"
                                   reason:@"Use initWithSecretCode:, not init"
                                   userInfo:nil];
}
```

30. Свойства

В предыдущей главе мы создали класс `Appliance`, который содержал два свойства: `productName` и `voltage`. Давайте разберемся, как работают эти свойства.

В файле *Appliance.h* объявляются две переменные экземпляров для хранения данных:

```
{
    NSString *productName;
    int voltage;
}
```

Также для них были объявлены методы доступа. Объявления могли бы выглядеть так:

```
- (void)setProductName:(NSString *)s;
- (NSString *)productName;
- (void)setVoltage:(int)x;
- (int)voltage;
```

Но мы вместо этого воспользовались конструкцией `@property`:

```
@property (copy) NSString *productName;
@property int voltage;
```

В файле *Appliance.m* методы доступа *можно* было реализовать явно:

```
- (void)setProductName:(NSString *)s
{
    productName = [s copy];
}
- (NSString *)productName
{
    return productName;
}
- (void)setVoltage:(int)x
{
voltage = x; }
- (int)voltage
{
    return voltage;
}
```

Однако мы использовали для их реализации конструкцию `@synthesize`:

```
@synthesize productName, voltage;
```

Любопытный факт о компиляторе Objective-C: при компилировании приложения для iOS или 64-разрядной платформы Mac OS X объявлять переменные экземпляров не нужно. Вызовов `@property/@synthesize` достаточно для того, чтобы память для данных была зарезервирована.

Закомментируйте переменные экземпляров в файле *Appliance.h*: {

```
{
// NSString *productName;
// int voltage;
}
```

Заново постройте и запустите программу.

В этой книге мы всегда объявляем переменные экземпляров. Объявление может рассматриваться как дополнительная форма документированная и позволяет использовать код в 32-разрядных программах для Mac OS X. Раскомментируйте переменные экземпляров.

Атрибуты свойств

А теперь поближе познакомимся с различными атрибутами, управляющими созданием методов доступа для свойств.

Изменяемость

Свойство можно объявить доступным для чтения/записи (`readwrite`) или только для чтения (`readonly`). По умолчанию используется значение `readwrite`, при котором создается как set-, так и `get`-метод. Если вы не хотите, чтобы для свойства создавался `set`-метод, пометьте свойство атрибутом `readonly`:

```
@property (readonly) int voltage;
```

Срок жизни

Свойство также может быть объявлено с атрибутом `unsafe_unretained`, `strong`, `weak` или copy. Это значение определяет тем, как `set`-метод организует управление памятью,

Значение `unsafe_unretained` используется по умолчанию; это самый простой вариант - свойству просто присваивается переданное значение. Для примера возьмем следующее объявление и определение:

```
@property (unsafe_unretained) int averageScore;
```

```
// "@property int averageScore" would also work here
...
@synthesize averageScore;
```

Сгенерированный set-метод будет практически эквивалентен следующему:

```
- (void)setAverageScore:(int)d
{
    averageScore = d;
}
```

В Appliance свойство voltage не изменяет счетчик ссылок. Атрибут unsafe_unretained всегда используется для свойств, содержащих не-объектные значения.

Атрибут strong, как упоминалось в главе 20, обеспечивает сохранение сильной ссылки на переданный объект. Он также уменьшает счетчик ссылок старого объекта (который уничтожит себя при отсутствии других владельцев). Для свойств, содержащих объекты, обычно используется атрибут strong.

Атрибут weak не подразумевает владения объектом. Он синтезирует set-метод, который присваивает свойству переданный объект. В случае уничтожения объекта свойству задается значение nil. (Обратите внимание: если указатель имеет атрибут unsafe_unretained, то при уничтожении объекта, на который он ссылается, в программе появляется «висячий указатель». Отправка сообщения такому указателю обычно приводит к сбою программы.)

Атрибут copy создает сильную ссылку на копию переданного объекта. Но здесь имеется одна тонкость, которую многие разработчики понимают неправильно...

copy

Атрибут copy создает копию объекта и пере водит на нее указатель. Допустим, объявление и определение свойства выглядят следующим образом:

```
@property (copy) NSString *lastName;
@synthesize lastName;
```

Сгенерированный set-метод выглядит примерно так:

```
- (void)setLastName:(NSString *)d
{
    lastName = [d copy];
}
```

Атрибут copy чаще всего используется с объектными типами, имеющими изменяемые субклассы. Например, NSString имеет субкласс с именем

NSMutableString. Как нетрудно представить, методу setLastName: может передаваться изменяемая строка:

```
// создание изменяемой строки
NSMutableString *x = [[NSMutableString alloc] initWithString:@"Ono"];

// передача ее setLastName:
[myObj setLastName:x];

// 'copy' запрещает this изменять lastName
[x appendString:@" Lennon"];
```

А если передаваемый объект не является изменяемым? Создавать копию неизменяемого объекта было бы неэффективно. Метод copy просто вызывает copyWithZone: и передает аргумент nil. Например, в NSString метод copyWithZone: переопределятся следующим образом:

```
- (id)copyWithZone:(NSZone *)z
{
    return self;
}
```

Таким образом, копия вообще не создается. (Учтите, что NSZone и зонирование памяти вообще - устаревшие, рудиментарные возможности программирования Cocoa, поэтому здесь они подробно не рассматриваются. Впрочем, метод copyWithZone: еще находит практическое применение и не считается полностью вытесненным.)

Для объектов, существующих в изменяемой и неизменяемой версиях, метод copy возвращает неизменяемую копию. Например, NSMutableString содержит метод copy, который возвращает экземпляр NSString. Если вы хотите, чтобы копия была изменяемым объектом, используйте метод mutableCopy.

Атрибута срока жизни свойств с именем mutableCopy не существует. Если бы хотите, чтобы set-метод назначал свойству изменяемую копию объекта, вы должны реализовать его самостоятельно, с вызовом метода mutableCopy для входного объекта. Например, в классе OwnedAppliance можно создать метод setOwnerNames: :

```
- (void)setOwnerNames:(NSSet *)newNames
{
    ownerNames = [newNames mutableCopy];
}
```

Подробнее о копировании

Большинство классов Objective-C вообще не содержит метода copyWithZone:. Программисты Objective-C создают меньше копий, чем кажется на первый взгляд. Интересно, что методы copy и mutableCopy определяются в NSObject следующим образом:

```
- (id)copy {
    return [self copyWithZone:NULL];
}
- (id)mutableCopy
{
    return [self mutableCopyWithZone:NULL];
}
```

Следовательно, при попытке выполнения кода

```
Appliance *b = [[Appliance alloc] init];
Appliance *c = [b copy];
```

вы получите ошибку следующего вида:

```
-[Appliance copyWithZone:]: unrecognized selector sent to instance 0x100110130
```

Атомарность

Эта книга начального уровня, а атрибут `atomic/nonatomic` относится к относительно нетривиальной теме многопоточности. Сейчас вам необходимо знать следующее: в режиме `nonatomic` ваш `set`-метод будет выполняться чуть быстрее. Заглянув в заголовки UlKit, вы увидите, что все свойства помечены атрибутом `nonatomic`. Вам тоже "Стоит объявлять свои свойства с атрибутом `nonatomic`.

(Я даю этот совет всем без исключения. Однако в каждой группе находится умник, кто знает ровно столько, чтобы возразить. Он говорит: «Но ведь в многопоточном приложении мне потребуется защита, которую мне обеспечат атомарные `set`-методы». И мне стоило бы сказать: «Не думаю, что вы скоро будете писать многопоточный код. А когда начнете, атомарность `set`-методов вам вряд ли поможет». Но я говорю другое: «Ладно, тогда оставьте свои `set`-методы атомарными». Потому что не стоит говорить то, что люди еще не готовы услышать.)

В *Appliance.h* объявите свои методы доступа с атрибутом `nonatomic`:

```
@property (copy, nonatomic) NSString *productName;
@property (nonatomic) int voltage;
```

К сожалению, на момент написания книги значение `atomic` используется по умолчанию, поэтому вам придется внести это изменение.

Запись «ключ-значение»

Запись «ключ-значение» позволяет читать и задавать свойства по имени. Соответствующие методы определены в `NSObject`, поэтому каждый объект поддерживает данную возможность.

Откройте файл *main.m* и найдите строку:

```
[a setProductName:@"Washing Machine"];
```

Перепишите ее с использованием записи «ключ-значение»:

```
[a setValue:@"Washing Machine" forKey:@"productName"];
```

В этом случае метод `setValue:forKey:`, определенный в `NSObject`, будет искать `set`-метод С именем `setProductName:`. Если у объекта нет метода `setProductName:`, он обращается к переменной экземпляра напрямую.

Запись «ключ-значение» также позволяет прочитать значение переменной. Включите в *main.m* строку для вывода имени устройства:

```
int main (int argc, const char * argv[])
{
    @autorelease {
      Appliance *a = [[Appliance alloc] init];
      NSLog(@"a is %@", a);
      [a setValue:@"Washing Machine" forKey:@"productName"];
      [a setVoltage:240];
      NSLog(@"a is %@", a);
      NSLog(@"the product name is %@", [a valueForKey:@"productName"]);
        }
        return 0;
}
```

На этот раз метод `valueForKey:`, определенный в `NSObject`, ищет метод доступа с именем `productName`. Если метод `productName` не найден, он обращается к переменной экземпляра напрямую.

Если имя свойства указано неверно, вы не получите предупреждения от компилятора, но во время выполнения произойдет ошибка. Сделайте ошибку в *main.m*:

```
NSLog(@"the product name is %@", [a valueForKey:@"productNammmme"]);
```

Постройте и запустите программу - вы получите следующее сообщение об ошибке:

```
*** Terminating app due to uncaught exception 'NSUnknownKeyException',
reason: '[<Appliance 0x100108dd0> valueForUndefinedKey:]:
this class is not key value coding-compliant for the key productNammmme.'
```

Исправьте ошибку, прежде чем продолжать.

Почему запись .ключ-значение~ представляет интерес? Каждый раз, когда стандартная библиотека записывает данные в ваши объекты, она использует `setValue:forKey:`. Каждый раз, когда стандартная библиотека читает данные из ваших объектов, она использует `valueForKey:`. Например, библиотека `CoreData` упрощает сохранение объектов в базе данных `SQLite` и их последующую загрузку. Для работы с пользовательским объектами, содержащими данные, используется запись «ключ-значение».

Чтобы убедиться в том, что запись «ключ-значение» выполнит операции с переменными даже при отсутствии методов доступа, закомментируйте объявление `@property` для свойства `productName` в файле *Appliance.h*:

```
#import <Foundation/Foundation.h>
@interface Appliance : NSObject {
    NSString *productName;
    int voltage;
}
// @property (copy) NSString *productName; @property (nonatomic) int voltage;
// основной инициализатор
- (id)initWithProductName:(NSString *)pn;
@end
```

Также удалите все случаи использования методов `setProductName:` и `productName` из *Appliance.m*:

```
@implementation Appliance
@synthesize voltage;
- (id)initWithProductName:(NSString *)pn
{
    self = [super init];
    if (self) {
        productName = [pn copy];
        [self setVoltage:120];
    }
    return self;
}
- (id)init {
    return [self initWithProductName:@"Unknown"];
}
- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@: %d volts>", productName, voltage];
}
@end
```

Постройте и запустите программу. Несмотря на отсутствие методов доступа для `productName`, значение переменной все равно может читать и задавать из других методов. Это является очевидным нарушением идеи инкапсуляции объекта - методы объекта открыты, но переменные экземпляров должны оставаться приватными, то

есть недоступными извне. Если бы запись «ключ-значение» не была невероятно полезной, никто бы с этим не смирился.

Не-объектные типы

Методы записи «ключ-значение» предназначены для работы с объектами, но в некоторых свойствах хранятся не-объектные типы (например, `int` или `float` - скажем, `voltage` относится к типу `int`). Как задать значение `voltage` в записи «ключ-значение»? Используйте `NSNumber`.

В файле main.m измените строку, задающую значение `voltage`:

```
[a setVoltage:240];
```

следующей строкой:

```
[a setValue:[NSNumber numberWithInt:240] forKey:@"voltage"];
```

Добавьте в *Appliance.m* метод доступа для вывода информации о вызове:

```
- (void)setVoltage:(int)x
{
    NSLog(@"setting voltage to %d", x);
voltage = x;
}
```

Постройте и запустите программу

Аналогичным образом по запросу `valueForKey:@"voltage"` вы получите объект `NSNumber`, содержащий значение `voltage`.

31. Категории

Категории позволяют программисту добавить новые методы в любой существующий класс. Например, фирма Apple предоставила нам класс `NSString`. Исходный код этого класса недоступен, но категории дают возможность добавить новые методы в `NSString`.

Создайте новую программу командной строки `Foundation` с именем `VowelCounter`. Затем создайте новый файл, являющийся категорией Objective-C (Objective-C category). Присвойте категории имя `VowelCounting` и назначьте ее категорией для `NSString`.

Откройте *NSString+VowelCounting.h* и объявите метод, который вы хотите добавить в класс `NSString`:

```
#import <Foundation/Foundation.h> @interface NSString (VowelCounting)
- (int)vowelCount;
@end
```

Реализуйте метод в *NSString+ VowelCount.m*:

```
#import "NSString+VowelCounting.h" @implementation NSString (VowelCounting)
- (int)vowelCount
{
    NSCharacterSet *charSet =
            [NSCharacterSet
characterSetWithCharactersInString:@"aeiouyAEIOUY"];
    NSUInteger count = [self length];
    int sum = 0;
    for (int i = 0; i < count; i++) {
        unichar c = [self characterAtIndex:i];
        if ([charSet characterIsMember:c]) {
sum++; }
}
return sum; }
@end
```

Теперь используйте новый метод в *main.m*:

```
#import <Foundation/Foundation.h>
#import "NSString+VowelCounting.h"
int main (int argc, const char * argv[])
{
    @autorelease {
        NSString *string = @"Hello, World!";
        NSLog(@"%@ has %d vowels", string, [string vowelCount]);
}
return 0;
}
```

Постройте и запустите программу. Удобно, не правда ли? Категории чрезвычайно полезны.

Важно отметить, что категория присутствует только в этой программе. Если вы хотите, чтобы метод был доступен в другой программе. добавьте файл в проект и откомпилируйте категорию при построении программы.

32. Блоки

В главе 24 были рассмотрены механизмы обратного вызова: делегирование и оповещения. Обратный вызов позволяет другим объектам вызывать методы ваших объектов в ответ на возникновение каких-либо событий. Хотя эти механизмы абсолютно работоспособны, они приводят к фрагментации кода. Части программы. которые вам хотелось бы держать рядом друг с другом для большей ясности, обычно оказываются удаленными друг от друга.

Например, в программе `Callbacks` из главы 24 мы добавили код регистрации объекта на оповещения об изменении часового пояса пользователя и указали, что при получении такого оповещения должен вызываться метод `zoneChange:`. Но когда я читаю ваш код, мне интересно, что делает метод `zoneChange:` при вызове, поэтому я обращаюсь к реализации этого метода. В примере `Callbacks` код регистрации объекта для оповещений и реализация метода находятся по соседству, но нетрудно представить, что в большем, более сложном приложении эти два блока будут удалены друг от друга на сотни строк.

В Mac OS X 10.6 и iOS 4 появилась новая возможность - так называемые *блоки* (`blocks`). Блок Objective-C представляет собой фрагмент кода (как и функция C), но он может передаваться в программе по аналогии сданными. Вскоре мы увидим, как это помогает хранить взаимосвязанный код по соседству.

Блоки и их синтаксис определенно относятся к числу нетривиальных аспектов Objective-C, и на первый взгляд они кажутся довольно запутанными. Однако в API фирмы Apple блоки встречаются все чаще. В этой главе мы рассмотрим пару простых примеров, чтобы вы были готовы к встрече с ними.

Если У вас имеется опыт программирования на других языках, возможно, вы уже встречались с блоками под названиями *анонимных функций*, *замыканий* или *лямбда-функций*. Если вы знакомы с указателями на функции, блоки могут показаться чем-то похожим, но вскоре вы убедитесь, что при правильном использовании блоков код получается более элегантным, чем при использовании указателей на функции.

Определение блоков

Блок выглядит так:

```
^{
    NSLog(@"I'm a log statement within a block!");
}
```

Похоже на функции, но имя функции заменено символом «^». Этот символ идентифицирует следующий фрагмент кода как блок. Кроме того, блоки, как и функции, могут получать аргументы:

```
^(double dividend, double divisor) {
    double quotient = dividend / divisor;
    return quotient;
}
```

Этот блок получает аргументы - два значения типа double. Блок также может иметь возвращаемое значение, но об этом чуть позднее.

Есть ли у блоков имена? Пока нет. Блок является значением (как, скажем, является значением число 5). Чтобы иметь возможность обращаться к блоку по имени, необходимо присвоить его блочной переменной.

Использование блоков

Понять, как работают блоки, проще всего на конкретном примере. В этом упражнении мы используем блок для исключения всех гласных букв из каждой строки массива.

Создайте новую программу командной строки `Foundation` с именем `VowelMovement`. В этой программе мы используем блок для перебора массива строк с последовательным преобразованием каждой строки. Сначала мы создадим три массива: для исходных строк, для строк с удаленными гласными и для хранения символов, удаляемых из строк. В файле *main.m* замените код в фигурных скобках `@autoreleasepool`:

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // создание массива строк, из которых удаляются гласные,
        // и контейнера для новых строк
        NSArray *oldStrings = [NSArray arrayWithObjects:
                 @"Sauerkraut", @"Raygun", @"Big Nerd Ranch", @"Mississippi",
nil];
        NSLog(@"old strings: %@", oldStrings);
        NSMutableArray *newStrings = [NSMutableArray array];
        // создание списка символов, удаляемых из строки
        NSArray *vowels = [NSArray arrayWithObjects:
            @"a", @"e", @"i", @"o", @"u", nil];
}
return 0;
}
```

Здесь ничего нового; мы просто создаем массивы. Постройте и запустите программу. На предупреждения о не используемых переменных пока не обращайте внимания.

Объявление блочной переменной

Теперь займемся кодом блока. Хотя блоки внешне похожи на функции, они могут храниться в переменных. Как и другие переменные. блочные переменные объявляются, после чего им присваиваются значения. Включите следующий код в *main.m*, чтобы объявить блочную переменную.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
       // создание массива строк, из которых удаляются гласные,
       // и контейнера для новых строк
        NSArray *oldStrings = [NSArray arrayWithObjects:
                 @"Sauerkraut", @"Raygun", @"Big Nerd Ranch", @"Mississippi",
nil];
        NSLog(@"old strings: %@", oldStrings);
        NSMutableArray *newStrings = [NSMutableArray array];
        // создание списка символов, удаляемых из строки
        NSArray *vowels = [NSArray arrayWithObjects:
                           @"a", @"e", @"i", @"o", @"u", nil];
        // объявление блочной переменной
        void (^devowelizer)(id, NSUInteger, BOOL *);
}
return 0;
}
```

Давайте посмотрим, из каких частей состоит это объявление.

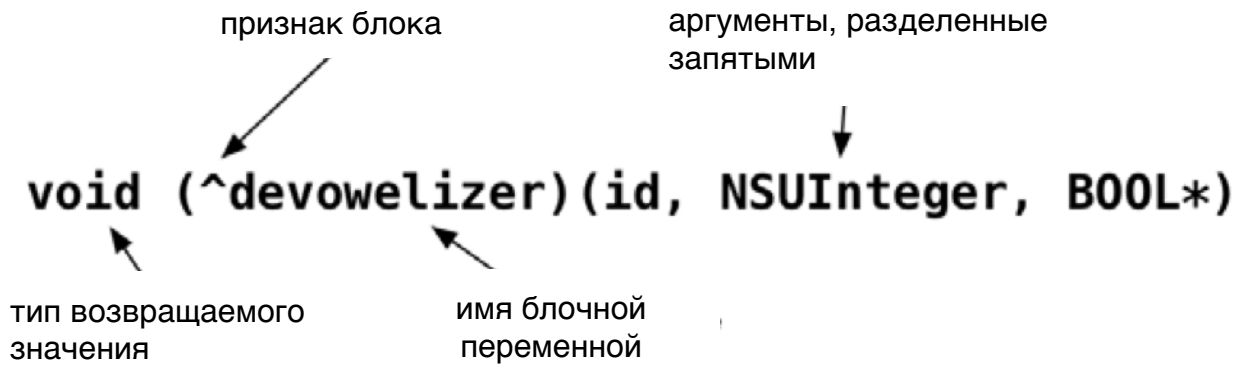


Рис. 32.1. Объявление блочной переменной

Объявляя примитивную переменную, вы указываете ее тип и имя - например, `int i`. Для блочной переменной имя указывается в середине объявления, сразу же за символом «^». Тип блочной переменной зависит от способа конструирования блока. В данном случае `devowelizer` относится к типу «блок, который получает объект, целое число и указатель на `BOOL` и не возвращает ничего».

Присваивание блока

Теперь присвоим значение нашей новой переменной. Значение блочной переменной всегда представляет собой набор инструкций в фигурных скобках. Включите в файл *main.m* следующий фрагмент:

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // создание массива строк, из которых удаляются гласные,
        // и контейнера для новых строк
        NSArray *oldStrings = [NSArray arrayWithObjects:
                 @"Sauerkraut", @"Raygun", @"Big Nerd Ranch",
                 @"Mississippi", nil];
        NSLog(@"old strings: %@", oldStrings);
        NSMutableArray *newStrings = [NSMutableArray array];

        // создание списка символов, удаляемых из строки
        NSArray *vowels = [NSArray arrayWithObjects:
                           @"a", @"e", @"i", @"o", @"u", nil];

        // объявление блочной перменной
        void (^devowelizer)(id, NSUInteger, BOOL *);

        // присваивание блока переменной
        devowelizer = ^(id string, NSUInteger i, BOOL *stop) {
                  NSMutableString *newString = [NSMutableString
                                                stringWithString:string];
        // перебор массива гласных и замена всех вхождений
        // элементов пустой строки
        for (NSString *s in vowels) {
            NSRange fullRange = NSMakeRange(0, [newString length]);
            [newString replaceOccurrencesOfString:s
                               withString:@""
                               options:NSCaseInsensitiveSearch
                               range:fullRange];
}
    [newStrings addObject:newString];
}; // конец присваивания блока
}
return 0;
}
```

Снова постройте программу чтобы проверить правильность ввода. Предупреждение о не используемых переменных исчезает.

Итак, мы создали блок (набор инструкций) и присвоили его блочной переменной devowelizer. Обратите внимание: присваивание блока завершается символом «`;`», как и присваивание любой другой переменной.

Объявление `devowelizer` можно совместить с присваиванием, как и в случае с любой другой переменной:

```
void (^devowelizer)(id, NSUInteger, BOOL *) = ^(id string, NSUInteger i, BOOL
*stop) {
    NSMutableString *newString = [NSMutableString stringWithString:string];
    // перебор массива гласных и замена всех вхождений
    // элементов пустой строки
    for (NSString *s in vowels) {
        NSRange fullRange = NSMakeRange(0, [newString length]);
        [newString replaceOccurrencesOfString:s
                                withString:@""
                                options:NSCaseInsensitiveSearch
                                range:fullRange];
}
    [newStrings addObject:newString];
};
```

Как и в предыдущем варианте, здесь объявляется блочная переменная с именем `devowelizer`, которая получает три аргумента и не возвращает ничего. Затем мы создаем блок и сохраняем его в `devowelizer`.

Передача блока

Поскольку `devowelizer` является переменной, ее можно передать в аргументе. Класс `NSArray` содержит метод с именем `enumerateObjectsUsingBlock:`. Единственным аргументом этого метода является блок, который выполняется один раз для каждого объекта в массиве.

Включите в файл *main.m* следующий код, который вызывает `enumerateObjectsUsingBlock:` с блоком `devowelizer`, а затем выводит строки с удаленными гласными.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
       // создание массива строк, из которых удаляются гласные,
       // и контейнера для новых строк
       NSArray *oldStrings = [NSArray arrayWithObjects:
                 @"Sauerkraut", @"Raygun", @"Big Nerd Ranch",
                 @"Mississippi", nil];
        NSLog(@"old strings: %@", oldStrings);
        NSMutableArray *newStrings = [NSMutableArray array];

       // создание списка символов, удаляемых из строки
       NSArray *vowels = [NSArray arrayWithObjects:
                           @"a", @"e", @"i", @"o", @"u", nil];

       // объявление блочной перменной
       void (^devowelizer)(id, NSUInteger, BOOL *);
       // присваивание блока переменной
       devowelizer = ^(id string, NSUInteger i, BOOL *stop) {

      NSMutableString *newString = [NSMutableString stringWithString:string];
```

```
        // перебор массива гласных и замена всех вхождений
        // элементов пустой строки
        for (NSString *s in vowels) {
                NSRange fullRange = NSMakeRange(0, [newString length]);
                [newString replaceOccurrencesOfString:s
                                withString:@""
                                options:NSCaseInsensitiveSearch
                                range:fullRange];
}
    [newStrings addObject:newString];
}; // конец присваивания блока

        // перебор элементов массива с блоком
        [oldStrings enumerateObjectsUsingBlock:devowelizer];
        NSLog(@"new strings: %@", newStrings);
}
return 0;
}
```

Постройте и запустите программу На консоль выводятся два массива. Второй массив получается в результате удаления из первого всех гласных.

```
2011-09-03 10:27:02.617 VowelMovement[787:707] old strings: (
    Sauerkraut,
    Raygun,
    "Big Nerd Ranch",
    Mississippi
)
2011-09-03 10:27:02.618 VowelMovement[787:707] new strings: (
    Srkrt,
    Rygn,
    "Bg Nrd Rnch",
    Mssspp
)
```

Важно помнить, что в аргументе `enumerateObjectsUsingBlock:` может передаваться не любой блок. Метод требует, чтобы ему передавался «блок, который получает объект, целое число и указатель на `BOOL`, и не возвращает ничего». Вот почему блок; присвоенный `devowelizer`, был построен именно так. Три его аргумента предназначены специально для перебора содержимого массива.

Первый аргумент содержит указатель на текущий объект. Обратите внимание: указатель имеет тип `id`, поэтому он будет работать независимо от того, какие объекты хранятся в массива. Во втором аргументе передается значение `NSUInteger` - индекс текущего объекта. Третий объект содержит указатель на значение `BOOL`, по умолчанию равное `NO`. Если вы хотите, чтобы перебор элементов массива завершился после текущей итерации, измените его на `YES` .

Включите дополнительную проверку в начало присваивания блока:

```
devowelizer = ^(id string, NSUInteger i, BOOL *stop){
```

```
    NSRange yRange = [string rangeOfString:@"y"
                      options:NSCaseInsensitiveSearch];
    // нашли «y»?
    if (yRange.location != NSNotFound) {
        *stop = YES; // отмена дальнейшей итерации
        return;      // прерывание текущей итерации
    }

    NSMutableString *newString = [NSMutableString stringWithString:string];
    // перебор массива гласных и замена всех вхождений
    // элементов пустой строкой
    for (NSString *s in vowels) {
        NSRange fullRange = NSMakeRange(0, [newString length]);
        [newString replaceOccurrencesOfString:s
}

    [newStrings addObject:newString];
}; // конец присваивания блока
```

Новый фрагмент проверяет, содержит ли строка текущей итерации букву «y» в верхнем или нижнем регистре. Если вхождение найдено, то указателю задается значение YES (отменяющее последующие итерации), после чего текущий перебор прерывается.

Постройте и запустите программу. И снова на консоль выводятся два массива, но на этот раз перебор отменяется на второй итерации, когда блок обнаруживает слово с буквой «y». от всего массива остается только Srkrt.

Итак, мы немного потренировались в использовании блоков. Давайте вернемся и посмотрим, как блоки помогают в решении проблемы логической удаленности фрагментов кода в программах. При использовании обратного вызова - как в главе 24, где использовалась следующая строка кода:

```
[[NSNotificationCenter defaultCenter]
                      addObserver:logger
                      selector:@selector(zoneChange:)
                      name:NSSystemTimeZoneDidChangeNotification object:nil];
```

вы задаете метод (обычно с использованием @selector( )), после чего реализуете этот метод где-то в другом месте файла:

```
- (void)zoneChange:(NSNotification *)note
{
    NSLog(@"The system time zone has changed!");
}
```

Также возможно использование метода NSNotificationCenter addObserverForName:object:queue:usingBlock: и передачей блока. При использовании этого метода NSNotificationCenter происходит непосредственная передача инструкций, так что вам не придется размещать код обратного вызова где-то в другом месте. Любой читатель вашего кода увидит инструкции и сообщение,

отправляемое NSNotificationCenter, в одном фрагменте кода. (Именно такое изменение в программе Callbacks вам будет предложено реализовать в упражнении в конце этой главы.)

typedef

Синтаксис блоков может быть довольно сложным, но его можно упростить при помощи ключевого слова typedef, о котором вы узнали в главе 10. Напомню, что определения typedef располагаются в начале файла или заголовка, вне реализации каких-либо методов. Включите в *main.m* следующую строку кода:

```
#import <Foundation/Foundation.h>
typedef void (^ArrayEnumerationBlock)(id, NSUInteger, BOOL *);
int main (int argc, const char * argv[])
{
```

Определение typedef идентично объявлению переменной блока. Однако в данном случае мы объявляем тип, а не переменную, поэтому рядом с символом «^» указывается соответствующее имя типа. Использование typedef упрощает объявления блоков. Запутанное объявление devowelizer:

```
void (^devowelizer)(id, NSUInteger, BOOL *);
```

заменяется намного более понятным:

```
ArrayEnumerationBlock devowelizer;
```

Такое объявление блочной переменной выглядит более знакомым. Следует помнить, что тип блока определяет только типы его аргументов и возвращаемого значения; он не имеет отношения к набору инструкций в блоке этого типа.

Возвращаемые значения

Наконец, если блок возвращает значение, то блочную переменную можно вызывать как функцию.

```
double (^divBlock)(double,double) = ^(double k, double j) {
    return k/j;
}
```

В этом коде объявляется блочная переменная divBlock, которая возвращает значение double и получает два значения double в аргументах. Далее переменной

присваивается значение - инструкция, возвращающая результат деления двух аргументов.

Возможное использование блока может выглядеть так:

```
double quotient = divBlock(42.0, 12.5);
```

Управление памятью

Блоки, как и примитивные переменные, создаются и хранятся в стеке. Следовательно, блок уничтожается вместе с кадром стека при возврате управления функцией или методом, где этот блок создается. Однако в некоторых ситуациях бывает нужно, чтобы блок продолжал существование и после выхода - на- пример, он может быть переменной экземпляра объекта. В таком случае блок необходимо скопировать из стека в кучу.

Чтобы скопировать блок из стека в кучу, отправьте ему сообщение `copy`:

```
ArrayEnumerationBlock iVarDevowelizer = [devowelizer copy];
```

Теперь копия блока существует в куче, а новая блочная переменная содержит указатель на этот блок.

Методы, получающие блоки в аргументах (такие, как `enumerateObjectsUsingBlock:` класса `NSArray` или `addObserverForName:object:queue:usingBlock:` класса `NSNotificationCenter`), должны скопировать переданные блоки. Тем самым они создают указатели - и сильные ссылки - на эти блоки.

Итак, блоки можно объявлять, присваивать значения и передавать, как переменные. Мы также видели, что блоки отчасти похожи на функции. А теперь мы отправляем блоку сообщение так, словно он является объектом.

Нахождение блока в куче и его «объектное» поведение создает некоторые проблемы с управлением памятью.

Что происходит с переменными, используемыми в блоке?

В своем коде блок обычно использует другие переменные (и примитивные, и указатели на объекты), созданные за его пределами. Чтобы внешние переменные оставались доступными на все время их использования в блоке, переменные захватываются блоком при создании копии.

Для примитивных переменных это означает копирование и сохранение их в локальных переменных блока. Для указателей блок хранит сильную ссылку на все объекты, к которым он обращается. Это означает, что любые объекты, ссылка на которые хранится в блоке, заведомо будут существовать по крайней мере до тех пор, пока существует сам блок. (Вас интересовало, чем блоки отличаются от указателей на функции? Пусть-ка указатель на функцию попробует сделать нечто подобное!)

Вернемся к программе `VowelMovement`. В блоке `devowelizer` упоминаются два объекта, созданных за пределами этого блока: `newStrings` (массив для хранения измененных версий строк) и `string` (текущая строка, копируемая для изменения).

devowelizer сохраняет сильные ссылки на оба объекта, вследствие чего они заведомо продолжают существовать до тех пор, пока существует сам блок.

Могут ли сильные ссылки привести к появлению циклических ссылок?

Конечно, могут. Решение проблемы нам уже известно: одна из ссылок должна стать слабой. Для этого за пределами блока объявляется слабый указатель (_weak), который потом используется в блоке.

Могу ли я изменить переменные, скопированные блоком?

По умолчанию переменные, скопированные блоком, в блоке изменяться не могут, то есть фактически превращаются в константы. Например, переменные указателей на объекты сохраняют постоянное значение в пределах блока. (Хотя вы можете отправить объекту сообщение, изменяющее его содержимое, сам указатель изменить нельзя.) Однако в некоторых случаях требуется изменить внешнюю переменную внутри блока. Для этого внешняя переменная должна быть объявлена с ключевым словом _block. Например, в следующем фрагменте увеличивается внешняя переменная counter:

```
__block int counter = 0;
void (^counterBlock)() = ^{ counter++; };
...
counterBlock(); // значение counter увеличевается до 1
counterBlock(); // значение counter увеличевается до 2
```

Без ключевого слова _block вы получите ошибку компиляции в определении блока, с сообщением о том, что значение counter не может изменяться.

Будущее блоков

Понять, как работают блоки и как их правильно использовать, может быть непросто. Однако блоки чрезвычайно полезны в приложениях, интенсивно использующих обработку событий, так часто встречающихся в программировании Mac и iOS. В программных интерфейсах Apple блоки используются все чаще. Например, библиотеки ALAssetLibrary и GameKit содержат многочисленные методы с использованием блоков. Старайтесь по возможности использовать блочные методы Apple, чтобы привыкнуть к работе с блоками.

Упражнения

Анонимный блок

В примере этой главы операции объявления, присваивания и использования блока находятся в трех разных строках кода. Это сделано для удобства чтения кода.

Если вам потребуется передать целое число методу (например, numberWithInt: класса NSNumber),вы можете передать тип int анонимно:

```
// Вариант 1: Последовательное выполнение всех операций
int i;
i = 5;
NSNumber *num = [NSNumber numberWithInt:i];

// Вариант 2: Значение передается без объявления переменной
NSNumber *num = [NSNumber numberWithInt:5];
```

Так как блоки являются переменными, аналогичным образом можно поступать и с блоками. Более того, это самый распространенный способ их использования. Разработчики почти никогда не объявляют блочные переменные для передачи блоков методам; они почти всегда выбирают вариант с анонимным использованием.

Измените упражнение этой главы так, чтобы блок передавался анонимно в аргументе enumerateObjectsUsingBlock:. Иначе говоря, вы оставляете блок, но избавляетесь от блочной переменной.

NSNotificationCenter

В главе 24 метод addObserver:selector:name:object: класса NSNotificationCenter использовался для регистрации обратных вызовов через метод zoneChange:. Измените это упражнение так, чтобы вместо него использовался метод addObserverForName:object:queue:usingBlock:. За подробностями обращайтесь к документации разработчика.

Метод получает блок в аргументе, а затем выполняет его при получении заданного оповещения вместо того, чтобы обращаться к объекту с обратным вызовом. Это означает, что метод zoneChange: вызываться не будет, а код, находящийся в этом методе, следует переместить в блок.

Передаваемый блок должен получать один аргумент (NSNotification *) и не возвращать ничего, как это делает zoneChange:.

В аргументе queue: можно передать nil; этот аргумент используется для синхронизации - тема, которая в книге не рассматривается.

VI

Нетривиальные возможности C

Квалифицированный программист Objective-C также должен хорошо владеть языком C. Существует еще немало возможностей программирования на C, о которых вам необходимо знать. Возможно, концепции, представленные в этой части, не будут постоянно использоваться в вашей повседневной работе. Тем не менее время от времени вы неизбежно будете сталкиваться C ними, поэтому я хочу представить их вам.

33. Поразрядные операции

В первой части книги я сравнил память компьютера с огромным массивом переключателей, которые могут находиться во включенном или выключенном состоянии. Каждый переключатель представляет один бит; значение 1 обычно считается «включенным», а 0 - «выключенным».

Однако в своих программах вы не адресуете отдельные биты, а имеете дело с группами битов - байтами. Если представить байт как 8-разрядное целое число без знака, его биты соответствуют последовательным степеням 2 (рис. 33.1).

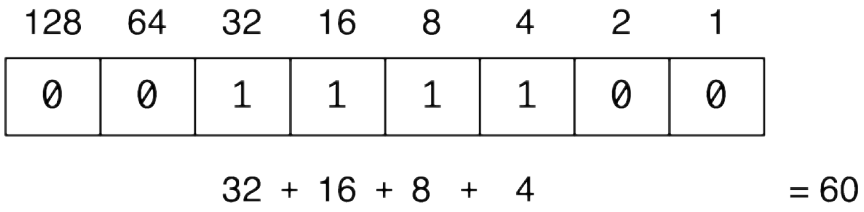


Рис. 33.1. Байт, представляющий десятичное число 60

Люди предпочитают работать в десятичной системе счисления - побочный эффект того, что у нас на руках по 10 пальцев. Однако компьютерам удобнее иметь дело со степенями 2. Программисты часто используют *шестнадцатеричную* систему ($16 = 2^4$) - особенно при работе с отдельными разрядами целых чисел.

В качестве дополнительных цифр в шестнадцатеричной системе используются буквы a, b, c, d, e и f. Таким образом, счет в шестнадцатеричной системе выглядит так: 0,1,2,3,4,5,6,7,8,9, a, b, c, d, e, f, 10, 11, ...

Числа, записанные в шестнадцатеричной системе, обозначаются префиксом 0x. На рис. 33.2 показано, как выглядит число и байт на рис.33.1 в шестнадцатеричной записи.

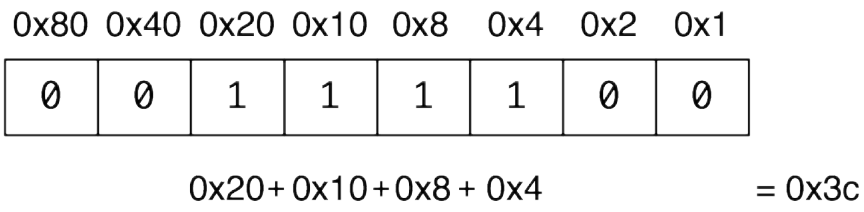


Рис. 33.2. Один байт, представляющий шестнадцатеричное число 0x3c

Один байт всегда может быть представлен шестнадцатеричным числом из двух цифр (например, 3c). По этой причине шестнадцатеричные числа удобны для работы с

двоичными данными. Наверняка вы слышали, как опытные программисты говорят что-нибудь вроде: «Я расшифровал формат файла, просматривая документ в шестнадцатеричном редакторе». Хотите просмотреть содержимое файла в виде последовательности байтов в шестнадцатеричной записи? Выполните в Terminal команду `hexdump` с именем файла:

```
$ hexdump myfile.txt
0000000 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31
0000010 2e 30 22 3f 3e 0a 3c 62 6f 6f 6b 20 78 6d 6c 6e
0000020 73 3d 22 68 74 74 70 3a 2f 2f 64 6f 63 62 6f 6f
0000030 6b 2e 6f 72 67 2f 6e 73 2f 64 6f 63 62 6f 6f 6b
0000040 22
0000041
```

В первом столбце указано смещение (шестнадцатеричное) для байта, указанного во втором столбце. Каждое шестнадцатеричное число из двух цифр представляет один байт.

Поразрядная операция ИЛИ

Два байта можно объединить поразрядной операцией `ИЛИ`; результат представляет собой третий байт. Бит третьего байта содержит 1 в том случае, если хотя бы один из двух соответствующих битов первых двух байтов равен 1.

Поразрядная операция `ИЛИ` выполняется оператором 1. Для экспериментов с отдельными битам создайте новый проект: программу командной строки C (не `Foundation`!) с именем `bitwize`.

Отредактируйте файл *main.c*:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    unsigned char a = 0x3c;
    unsigned char b = 0xa9;
    unsigned char c = a | b;
    printf("Hex: %x | %x = %x\n", a, b, c);
    printf("Decimal: %d | %d = %d\n", a, b, c);
return 0; }
```

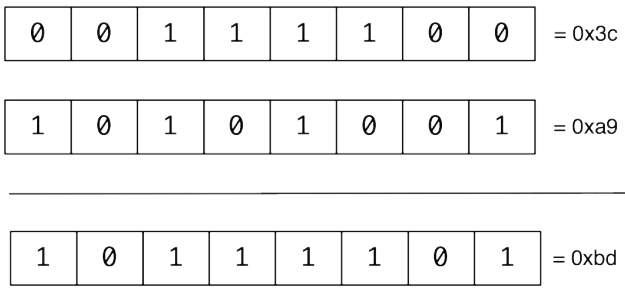


Рис. 33.3. Объединение двух байтов поразрядной операцией ИЛИ

Программа выдает результат объединения двух байтов поразрядной операцией ИЛИ:

```
Hex: 3c | a9 = bd
Decimal: 60 | 169 = 189
```

Зачем это нужно? В Objective-C целые числа часто используются для определения настроек. Целое число всегда представляет собой последовательность битов, и каждый бит представляет один аспект общей настройки, который может включаться/выключаться независимо от других. Это целое число (также называемое битовой маской) строится из констант, входящих в заранее определенный набор. Константы также являются целыми числами, и каждая константа задает один аспект настройки, то есть устанавливает только один бит в общем значении. Итоговое значение образуется посредством объединения констант, представляющих нужные аспекты, поразрядной операцией ИЛИ.

Рассмотрим конкретный пример. В iOS входит класс с именем NSDataDetector. Экземпляры NSDataDetector ищут в тексте вхождения стандартных обозначений - таких, как даты или URL-адреса. Искомые обозначения определяются объединением целочисленных констант из фиксированного набора поразрядной операцией ИЛИ.

В файле *NSDataDetector.h* определяются следующие константы: NSTextCheckingTypeDate, NSTextCheckingTypeAddress, NSTextCheckingTypeLink, NSTextCheckingTypePhoneNumber, и NSTextCheckingTypeTransitInformation. При создании экземпляра NSDataDetector вы указываете, что именно он должен искать. Например, чтобы экземпляр искал в тексте телефонные номера и даты, вы определяете его следующим образом:

```
NSError *e;
NSDataDetector *d = [NSDataDetector dataDetectorWithTypes:
                        NSTextCheckingTypePhoneNumber|NSTextCheckingTypeDate
                        error:&e];
```

Обратите внимание на оператор поразрядной операции или. Этот прием очень часто встречается в программировании Cocoa и iOS; теперь вы знаете, что при этом происходит «за кулисами».

Поразрядная операция И

Также два байта можно объединить поразрядной операцией И для создания третьего байта. В этом случае бит третьего байта равен 1 только в том случае, если оба соответствующих бита первых двух байтов равны 1.

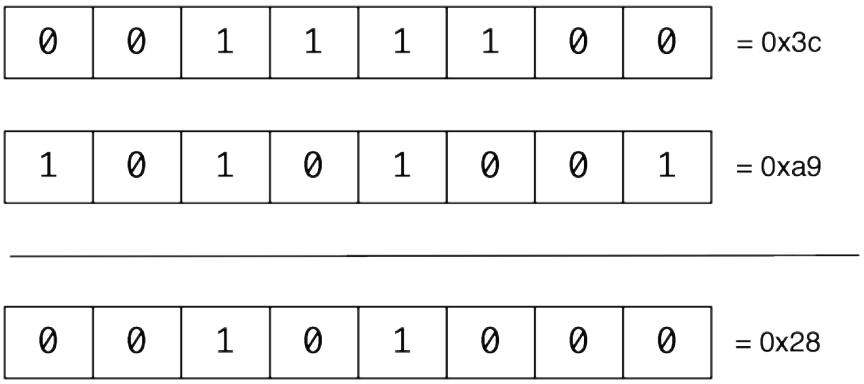


Рис. 33.4. Объединение двух байтов поразрядной операцией И

Поразрядная операция И выполняется оператором &. Включите в *main.c* следующие строки:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    unsigned char a = 0x3c;
    unsigned char b = 0xa9;
    unsigned char c = a | b;

    printf("Hex: %x | %x = %x\n", a, b, c);
    printf("Decimal: %d | %d = %d\n", a, b, c);
    unsigned char d = a & b;
    printf("Hex: %x & %x = %x\n", a, b, d);
    printf("Decimal: %d & %d = %d\n", a, b, d);
return 0;
}
```

Программа выдает результат объединения двух байтов поразрядной операцией И:

```
Hex: 3c & a9 = 28
Decimal: 60 & 169 = 40
```

В Objective-C поразрядная операция И используется для проверки состояния конкретного бита (или флага). Например, при получении экземпляра NSDataDetector вы можете проверить, включен ли для него поиск телефонных номеров:

```
if ([currentDetector checkingTypes] & NSTextCheckingTypePhoneNumber) {
    NSLog(@"This one is looking for phone numbers");
}
```

Метод checkingTypes возвращает целое число, которое является результатом объединения поразрядной операцией ИЛИ всех флагов, установленных в данном экземпляре NSDataDetector. Число объединяется поразрядной операцией И с конкретной константой NSTextCheckingType, после чего проверяется результат. Если

бит, установленный в NSTextCheckingTypePhoneNumber, не установлен в настройках NSDataDetector, результат поразрядной операции И будет состоять из одних нулей. В противном случае результат будет отличен от нуля, а вы знаете, что NSDataDetector ищет в тексте телефонные номера.

Другие поразрядные операторы

Для полноты картины также следует упомянуть о других поразрядных операторах. Они реже используются в Objective-C, но знать о них полезно.

Исключающая операция ИЛИ

Исключающая операция ИЛИ объединяет два байта и создает третий байт. Бит результата равен 1 в том случае, если ровно один из двух соответствующих битов входных байтов равен 1.

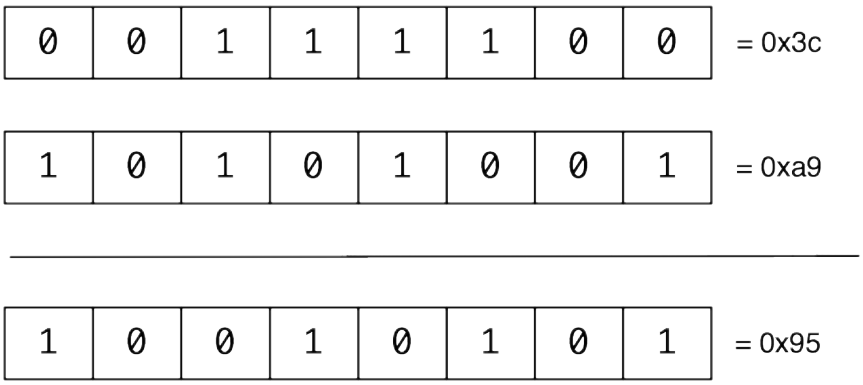


Рис. 33.5. Объединение двух байтов поразрядной исключающей операцией ИЛИ

Операция выполняется оператором ^. Включите в *main.c* следующий фрагмент:

```
unsigned char e = a ^ b;
    printf("Hex: %x ^ %x = %x\n", a, b, e);
    printf("Decimal: %d ^ %d = %d\n", a, b, e);
    return 0; }
      Программа выводит следующий результат:

Hex: 3c ^ a9 = 95
Decimal: 60 ^ 169 = 149
```

У неопытных программистов иногда возникают трудности с использованием этого оператора. В большинстве электронных таблиц оператор л выполняет операцию возведения в степень: 2^3 означает 2^3. В C возведение в степень выполняется функцией pow():

```
double r = pow(2.0, 3.0); // Вычисляет 2 в третьей степени
```

Дополнение

Дополнением к существующему байту называется байт, биты которого находятся в противоположном состоянии: все нули заменяются единицами, а все единицы заменяются нулями.

Операция выполняется оператором ~. Добавьте несколько строк в файл *main.c*:

```
    unsigned char f = ~b;
    printf("Hex: The complement of %x is %x\n", b, f);
    printf("Decimal: The complement of %d is %d\n", b, f);
    return 0;
}
```

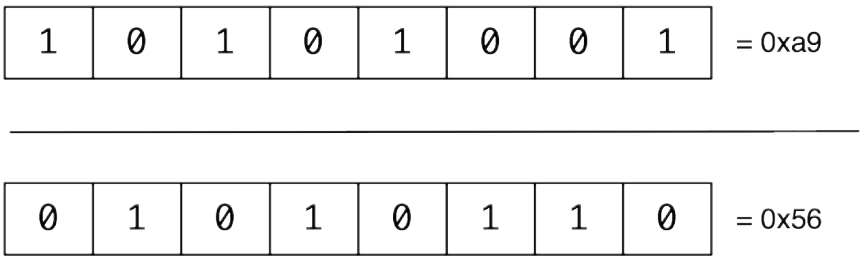


Рис. 33.6. Дополнение

Результат должен выглядеть так:

```
Hex: The complement of a9 is 56
Decimal: The complement of 169 is 86
```

Сдвиг влево

При выполнении операции сдвига влево каждый бит смещается в направлении старшего бита. Биты, выходящие за пределы числа, теряются, а возникающие справа «пустоты» заполняются нулями.

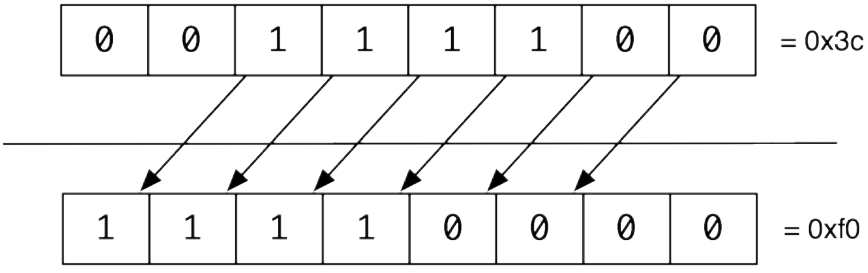


Рис. 33.7. Сдвиг влево на два разряда

Операция сдвига влево выполняется оператором <<. Включите в файл *main.c* сдвиг на две позиции влево:

```
unsigned char g = a << 2;
    printf("Hex: %x shifted left two places is %x\n", a, g);
    printf("Decimal: %d shifted left two places is %d\n", a, g);
return 0; }
```

Программа выводит следующий результат:

```
Hex: 3c shifted left two places is f0
Decimal: 60 shifted left two places is 240
```

При каждом сдвиге влево на один разряд значение числа удваивается.

Сдвиг вправо

Операция сдвига вправо работает вполне предсказуемо. Включите в файл *main.c* следующий фрагмент:

```
unsigned char h = a >> 1;
    printf("Hex: %x shifted right one place is %x\n", a, h);
    printf("Decimal: %d shifted right one place is %d\n", a, h);
return 0; }
```

Результат выполнения:

```
Hex: 3c shifted right one places is 1e
Decimal: 60 shifted right two places is 30
```

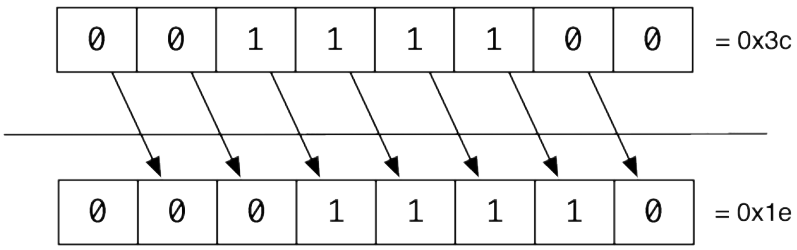


Рис. 33.8. Сдвиг вправо на один разряд

Использование enum для определения битовых масок

Часто в программе бывает нужно определить список констант; каждая из которых представляет целое число с одним установленным битом. Далее эти числа

объединяются поразрядной операцией ИЛИ и проверяются с использованием поразрядной операции И (см. выше).

Элегантное решение этой задачи основано на создании перечисления, использующего оператор сдвига влево для определения значений. Например, константы UIDataDetector определяются следующим образом:

```
enum {
    UIDataDetectorTypePhoneNumber         =1<<0,
    UIDataDetectorTypeLink                =1<<1,
    UIDataDetectorTypeAddress             =1<<2,
    UIDataDetectorTypeCalendarEvent       =1<<3,
    UIDataDetectorTypeNone                = 0,
    UIDataDetectorTypeAll                 = NSUIntegerMax
};
```

Больше байтов

В этой главе мы работали с типом unsigned char, состоящим из одного 8-разрядного байта. Любой беззнаковый целый тип работает аналогичным образом. Например, NSTextCheckingTypePhoneNumber в действительности объявляется как unit64_t - 64-разрядное число без знака.

Упражнение

Напишите программу, которая создает 64-разрядное целое число без знака, в котором установлен каждый второй бит. (Возможны два результата: четный и нечетный; создайте нечетный.) Выведите полученное число.

34. Строки C

Если у программиста Objective-C есть выбор, он всегда предпочтет работать с объектами `NSString` вместо строк C. Впрочем, иногда выбора нет. Самая распространенная причина для использования строк C? Работа с библиотеками C из кода Objective-C. Например, существует библиотека функций C, позволяющая вашей программе взаимодействовать с сервером баз данных `PostgreSQL`. Функции этой библиотеки используют строки C, а не экземпляры `NSString`.

char

В предыдущем разделе рассматривалась возможность интерпретации байтов как чисел. Байты также могут интерпретироваться как символы. Как упоминалось ранее, существует много разных кодировок символов. Самая старая, и пожалуй, самая известная - кодировка ASCII (American Standard Code for Information Interchange). В кодировке ASCII разные байты определяют разные символы. Например, код 0x4b соответствует символу «K». Создайте новую программу командной строки C с именем `yostring`. Программа будет выводить некоторые символы из стандарта ASCII. Отредактируйте *main.c*:.

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    char x = 0x21; // символ '!'
    while (x <= 0x7e) { // символ '~'
        printf("%x is %c\n", x, x);
        x++;
}
return 0; }
```

Возникает законный вопрос: «Байт может содержать одно из 256 чисел. Мы только что вывели 94 символа. А что происходит с остальными? Важно понимать, что кодировка ASCII разрабатывалась для управления старыми терминалами, работавшими по принципу телетайпа, которые печатали данные на бумаге (вместо вывода на экран). Например, при выводе числа 7 в кодировке ASCII устройство издает звуковой сигнал. Символы 0-31 в ASCII соответствуют непечатаемым управляющим символам: 32 - символ пробела, 127 - удаление предыдущего символа и т. д. Как насчет символов 128-255? В ASCII используются только 7 битов, для кода 128 ASCII - символа не существует.

Коды ASCII могут использоваться в коде как литералы, достаточно заключить их в апострофы. Внесите изменения в свой код:

```
int main (int argc, const char * argv[])
{
char x = '!'; // The character '!'
while (x <= '~') { // The character '~' printf("%x is %c\n", x, x);
x++;
}
return 0; }
```

Постройте и запустите программу

Непечатаемые символы могут быть представлены в виде управляющих *последовательностей*, начинающихся с символа \. Запись \n уже использовалась для символа новой строки. Приведу несколько распространенных последовательностей:

Таблица 34.1. Часто используемые управляющие последовательности

\n	новая строка
\t	табуляция
\'	апостроф
\"	кавычка
\0	нуль-байт (0x00)
\\	обратная косая черта

Char *

Строка C представляет собой простой набор символов, расположенных в смежной области памяти. Конец строки обозначается символом 0x00.

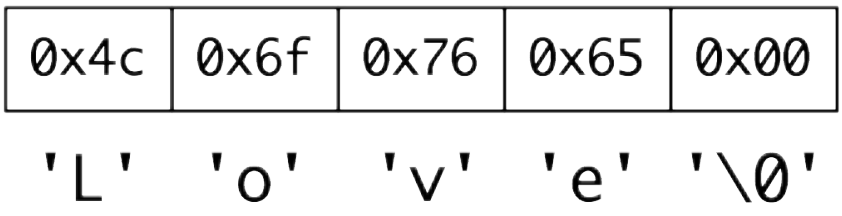


Рис. 34.1. Строка C со словом «Love»

Функции, которым передаются строки C, рассчитывают получить адрес первого символа строки. Например, функция `strlen()` считает количество символов в строке. Попробуйте создать строку и использовать `strlen()` для подсчета символов:

```
#include <stdio.h>  // For printf
#include <stdlib.h> // For malloc/free
#include <string.h> // For strlen
```

```
int main (int argc, const char * argv[])
{
    char x = '!'; // The character '!'
    while (x <= '~') { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
}
    // получение указателя на 5 байтпамяти в куче
    char *start = malloc(5);

   // в первый байт записывается 'L'
    *start = 'L';

    // во второй байт записывается 'o'
    *(start + 1) = 'o';

    // в третий байт аписывается 'v'
    *(start + 2) = 'v';

    // в четвертый байт записывается 'e'
    *(start + 3) = 'e';

    // в пятый байт записывается нуль
    *(start + 4) = '\0';

    // вывод строки и ее длины
    printf("%s has %zu characters\n", start, strlen(start));

    // вывод третьей буквы
    printf("The third letter is %c\n", *(start + 2));

    // освобождение памяти для повторного использования
    free(start);
    start = NULL;
return 0;
}
```

Постройте и запустите про грамму.

Обратите внимание на места с суммированием указателя и числа. Переменная `start` объявляется с типом `char *`. Тип `char` занимает один байт. Следовательно, значение `start+1` представляет собой указатель, смещенный на 1 байт в памяти по отношению к `start`; `start+2` - на 2 байта и т. д.

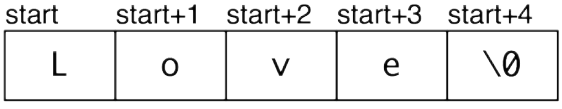


Рис. 34.2. Адреса символов строки

Прибавление к указателю и разыменование результата - операция настолько распространенная, что для нее было создано специальное сокращение: запись

`start[2]` эквивалентна `*(start+ 2)`. Внесите изменения в свой код:

```
char *start = malloc(5);
    start[0] = 'L';
    start[1] = 'o';
    start[2] = 'v';
    start[3] = 'e';
    start[4] = '\0';
printf("%s has %zu characters\n", start, strlen(start)); printf("The third
letter is %c\n", start[2]);
    free(start);
    start = NULL;
return 0; }
```

Постройте и запустите программу.

Следует отметить, что этот синтаксис работает с любыми типами данных. Например, я могу создать список трех вещественных чисел и вывести их:

```
int main (int argc, const char * argv[])
{
    // выделение блока памяти, размеров которого достаточен
    // для хранения трех значений типа float
    float *favorites = malloc(3 * sizeof(float));

    //занесение значений в ячейки буфера
    favorites[0] = 3.14158;
    favorites[1] = 2.71828;
    favorites[2] = 1.41421;

    // вывод всех чисел в списке
    for (int i = 0; i < 3; i++) {
        printf("%.4f is favorite %d\n", favorites[i], i);
    }

    // освобождение памяти для повторного использования
    free(favorites);
    favorites = NULL;
return 0; }
```

Единственное интересное различие заключается в том, что переменная `favorites` объявлена с типом `float *`. Тип float занимает 4 байта. Следовательно, результат `favorites+1` смещен на 4 байта в памяти по отношению к `favorites`.

Строковые литералы

Если вам приходится много работать со строками C, выделять память вызовом `malloc` и последовательно заносить символы было бы крайне неудобно. Вместо этого вы создаете указатель на строку символов (завершенную нуль-символом), заключая

строку в кавычки. Измените свой код так, чтобы в нем использовался строковый литерал:

```
int main (int argc, const char * argv[])
{
    char x = '!'; // The character '!'
    while (x <= '~') { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
}
    char *start = "Love";
    printf("%s has %zu characters\n", start, strlen(start));
    printf("The third letter is %c\n", start[2]);
    return 0;
}
```

Постройте и запустите программу

Для строковых литералов не нужно явно выделять память функцией `malloc` и освобождать ее. Это константы, которые размещаются в памяти только один раз, и управление памятью обеспечивается компилятором. У «константности» есть побочный эффект: попытка изменения символов строки приведет к неприятностям. Добавление следующей строки вызовет сбой программы:

```
char *start = "Love";
start[2] = 'z';
printf("%s has %zu characters\n", start, strlen(start));
```

При попытке построить и запустить программу выдается сообщение об ошибке EXC_BAD_ACCESS. Вы попытались выполнить запись в память, в которую запись запрещена.

Чтобы компилятор предупреждал о попытках записи в неизменяемые области памяти, воспользуйтесь модификатором `const` - он означает, что данные, на которые ссылается указатель, не должны изменяться. Попробуйте:

```
const char *start = "Love";
start[2] = 'z';
printf("%s has %zu characters\n", start, strlen(start));
```

Теперь при построении программы компилятор выдает сообщение об ошибке.

Удалите проблемную строку (`start[2] = 'z';`), прежде чем продолжать.

Вы также можете пропустить строки упомянутые выше. Попробуйте:

```
const char *start = "A backslash after two newlines and a tab:\n\n\t\\";
printf("%s has %zu characters\n", start, strlen(start)); printf("The third
letter is \'%c\'\n", start[2]);
return 0; }
```

Постройте и запустите программу.

Преобразования к NSString и обратно

Если вы работаете со строками C в программе Objective-C, вы должны уметь создавать `NSString` из строк C. Для этого в классе `NSString` имеется специальный метод:

```
char *greeting = "Hello!";
NSString *x = [NSString stringWithCString:greeting encoding:NSUTF8StringEncoding];
```

Также возможна и обратная операция - создание строки C на базе `NSString`. Сделать это немного сложнее, потому что `NSString` может работать с некоторыми символами, не поддерживаемыми другими кодировками. Желательно сначала убедиться в том, что преобразование возможно:

```
NSString *greeting = "Hello!";
const char *x = NULL;
if ([greeting canBeConvertedToEncoding:NSUTF8StringEncoding]) {
    x = [greeting cStringUsingEncoding:NSUTF8StringEncoding];
}
```

Вы не являетесь владельцем созданной строки C; система в конечном итоге освободит ее за вас. Гарантировано, что строка будет существовать по крайней мере до тех пор, пока существует текущий пул autorelease; если строка C должна существовать более долгое время, скопируйте ее в буфер, созданный вызовом `malloc()`.

Упражнение

Напишите функцию с именем `spaceCount()` для подсчета пробелов (ASCII 0x20) в строке C. Проверьте работу функции следующим образом:

```
#include <stdio.h>

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

    const char *sentence = "He was not in the cab at the time.";
    printf("\"%s\" has %d spaces\n", sentence, spaceCount(sentence));

    return 0;
}
```

И помните: обнаружение символа `'\0'` означает, что вы добрались до конца строки!

35. Массивы C

В предыдущей главе мы работали со строками C. Как вы узнали, строка C представляет собой последовательность символов, расположенных рядом друг с другом в памяти. Массивы C также представляют собой последовательности других типов данных, расположенных рядом друг с другом в памяти. Как и в случае со строками, операции со списком базируются на адресе первого элемента.

Допустим, вам потребовалось написать программу для вычисления среднего арифметического трех оценок. Создайте про грамму командной строки C с именем `gradeInTheShade`.

Отредактируйте файл *main.c*:

```
#include <stdio.h>
#include <stdlib.h> // malloc(), free()
float averageFloats(float *data, int dataCount)
{
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}
int main (int argc, const char * argv[])
{
    //создание массива значений floats
    float *gradeBook = malloc(3 * sizeof(float));
    gradeBook[0] = 60.2;
    gradeBook[1] = 94.5;
    gradeBook[2] = 81.1;

    // вычесление среднего
    float average = averageFloats(gradeBook, 3);

    // освобождение массива
    free(gradeBook);
    gradeBook = NULL;
    printf("Average = %.2f\n", average);
return 0; }
```

Постройте и запустите программу

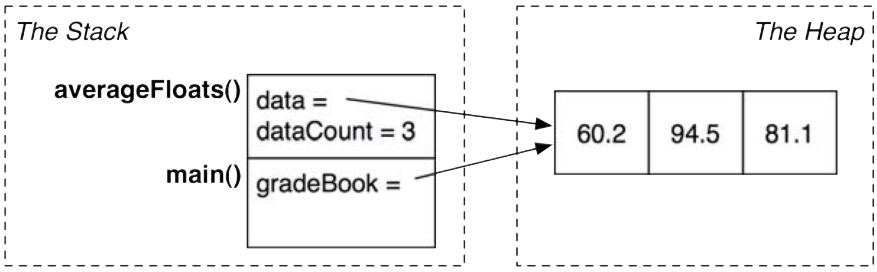


Рис. 35.1. Указатели на буфер значений fioat

Функция `malloc()` выделяет буфер в куче, поэтому вы должны проследить за тем, чтобы он был освобожден после завершения работы. А разве не удобнее было бы объявить буфер как часть кадра (в стеке), чтобы он автоматически освобождался после завершения выполнения функции? Да, это возможно.

Внесите изменения в *main.c*:

```
import <stdio.h>
float averageFloats(float *data, int dataCount)
{
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}
int main (int argc, const char * argv[])
{
    // объявление массива как части кадра
    float gradeBook[3];
    gradeBook[0] = 60.2;
    gradeBook[1] = 94.5;
    gradeBook[2] = 81.1;

    // вычисление среднего
    float average = averageFloats(gradeBook, 3);

    // освобождать массив не нужно
    // уничтожение производится автоматически при возвращении управления
функцией
    printf("Average = %.2f\n", average);
return 0; }
```

Постройте и запустите программу.

Строковый литерал упрощает заполнение массива символами. Также существуют специальные литералы массивов. Пример использования такого литерала для инициализации `gradeBook`:

```
int main (int argc, const char *argv[])
{
    float gradeBook[] = {60.2, 94.5, 81.1};
    float average = averageFloats(gradeBook, 3);
```

```
    printf("Average = %.2f", average);
return 0; }
```

Постройте и запустите программу.

Обратите внимание: указывать, что длина `gradeBook` равна 3, не обязательно; компилятор сам вычисляет ее по литералу массива. Этот тип может использоваться во многих местах, в которых обычно используется *. Например, измените объявление `averageFloats()` следующим образом:

```
float averageFloats(float data[], int dataCount) {
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}
```

Постройте и запустите программу.

36. Аргументы командной строки

При вызове `main()` тоже передаются аргументы, которые я так старательно обходил вниманием:

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

Но теперь вы готовы к знакомству с ними. `argv` - массив строк C; `argc` сообщает количество строк в массиве. Что представляют элементы массива? Аргументы командной строки.

Программы командной строки, которые мы создавали, запускались из Terminal-приложения, которое всего лишь предоставляет удобный интерфейс к так называемому *командному процессору* (shell). Существует несколько разных командных процессоров с похожими именами: *csh*, *sh*, *zsh* и *ksh*, но почти все пользователи Mac используют *bash*. При запуске программы из *bash* после имени программы можно задать любое количество аргументов, разделенных пробелами. Эти аргументы упаковываются в массив *argv* перед вызовом main ().

Честно говоря, программисты Cocoa и iOS редко используют `argv` и `argc`. Тем не менее для написания любой вспомогательной программы командной строки вам почти наверняка придется пользоваться ими.

Создайте в Xcode новый проект программы командной строки C с именем `Affirmation`. Программа `Affirmation` получает два аргумента: имя человека и число n, и n раз выводит сообщение с заданным именем.

```
$ Affirmation Mikey 3 Mikey is cool.
Mikey is cool.
Mikey is cool.
```

Прежде чем решать основную задачу, измените функцию `main()` так, чтобы она выводила каждый из аргументов в `argv`:

```
#include <stdio.h>
int main (int argc, const char * argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d = %s\n", i, argv[i]);
}
return 0; }
```

Если программа запускается из `bash`, аргументы можно просто ввести в командной строке:

```
$ Affirmation Aaron 4
```

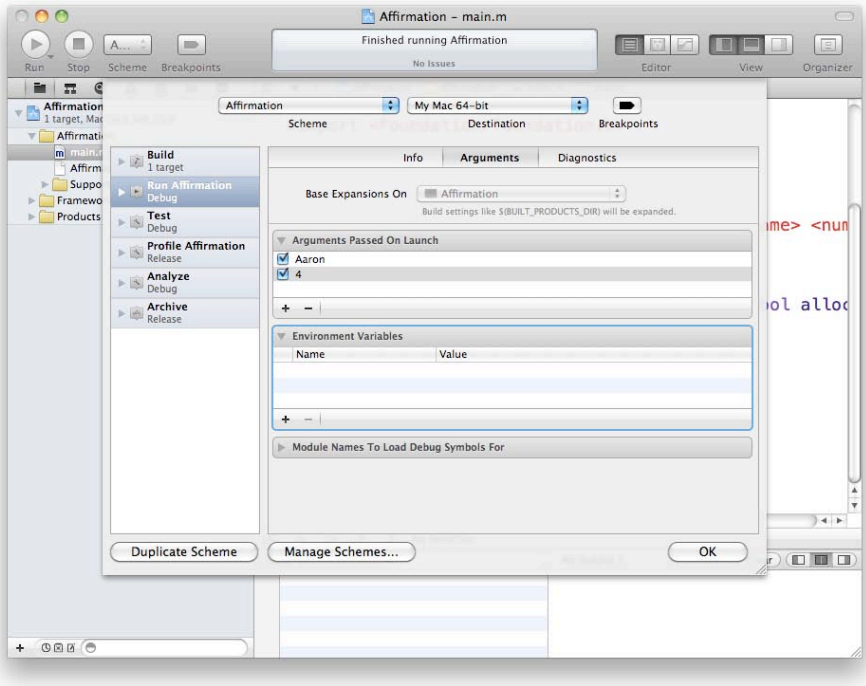


Рис. 36.1, Добавление аргументов

Но для выполнения программы с аргументами в Xcode необходимо предварительно отредактировать схему. Выберите в меню Product команду EditScheme...

В открывшемся окне выберите слева команду Run Affirmation и перейдите на вкладку Arguments в верхней части окна. Найдите список Arguments Passed On Launch и при помощи кнопки + добавьте два аргумента: имя и номер.

Щелкните на кнопке OK,чтобы закрыть окно.

При запуске программы в массиве `argv` передается список строк. Начинающих разработчиков обычно больше всего удивляет `argv[0]`:

```
arg 0 = /Users/aaron/Library/Developer/Xcode/DerivedData/
    Affirmation-enkfqsgavfsproeggoxwbrmcowvn/Build/Products/Debug/Affirmation
arg 1 = Aaron
arg 2 = 4
```

Элемент `argv[0]` содержит путь к исполняемому файлу программы.

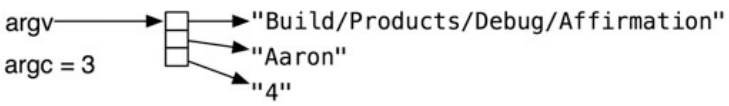


Рис. 36.2. argv и argc в программе Affirmation

Если программа получает аргументы, прежде всего необходимо убедиться в том, что ей передано правильное количество аргументов. Внесите изменения в *main.m*:

```
#include <stdio.h>
#include <stdlib.h> // atoi()


int main (int argc, const char * argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: Affirmation <name> <number>\n");
        return 1;
    }

    int count = atoi(argv[2]);
    for (int j = 0; j < count; j++) {
        printf("%s is cool.\n", argv[1]);
    }

    return 0;
}
```

`atoi` () - стандартная функция C, которая читает строку C и пытается преобразоватъ ее в число типа `int`.

Постройте и запустите программу.

37. Команда switch

В программах довольно часто встречается ситуация, когда переменную требуется проверить по набору значений. С использованием команд if-else проверка будет выглядеть так:

```
int yeastType = ...;
if (yeastType == 1) {
   makeBread();
} else if (yeastType == 2) {
   makeBeer();
} else if (yeastType == 3) {
   makeWine();
} else {
   makeFuel();
}
```

Для упрощения подобных проверок в C имеется команда `switch`. Приведенный код можно заменить следующим:

```
int yeastType = ...;
switch (yeastType) {
    case 1:
        makeBread();
        break;
    case 2:
        makeBeer();
        break;
    case 3:
        makeWine();
        break;
    default:
        makeFuel();
        break; }
```

Обратите внимание на команды `break`. Без них после выполнения подходящей секции `case` система продолжит выполнение всех последующих секций `case`. Например, если конструкция `switch` записана следующим образом:

```
int yeastType = 2;
switch (yeastType) {
    case 1:
        makeBread();
    case 2:
        makeBeer();
    case 3:
        makeWine();
```

```
    default:
        makeFuel();
}
```

программа выполнит `makeBeer()`, `makeWine()` и `makeFuel()`. В основном это сделано для того, чтобы один код мог выполняться для нескольких возможных значений:

```
int yeastType = ...;
switch (yeastType) {
    case 1:
    case 4:
        makeBread();
        break;
    case 2:
    case 5:
        makeBeer();
        break;
    case 3:
        makeWine();
        break;
    default:
        makeFuel();
break; }
```

Как нетрудно представить, отсутствие `break` в конце секции `case` - распространенная ошибка программирования, которая обнаруживается только при странном поведении программы.

В C для команд `switch` установлено одно жесткое ограничение: значение в секциях `case` может быть только целочисленной константой. По этой причине команды `switch` относительно редко встречаются в программах Objective-C, а я описал эту конструкцию почти в самом конце книги.