

Программирование на Objective-c 2.0

ББК 32.97

УДК 681.3

К79

Кочан Стивен

K79 Программирование на Objective-C 2.0 / Кочан С.; пер. с англ. — М.: ЭКОМ Паблишерз, 2010.— 608 с.: ил.

Objective-C – стандартный язык программирования приложений на платформах Mac OS X и iPhone. Он также распространен в операционных системах Linux, Unix и Windows. Это мощный и вместе с тем простой язык объектно-ориентированного программирования, базирующийся на языке C. Цель этой книги – обучение программированию на Objective-C. Работа с Objective-C показана на множестве подробных примеров, предназначенных для решения повседневных задач. Основы языка излагаются в части I. Многие возможности программирования на Objective-C основываются на использовании фреймворков. Фреймворк – это набор классов и процедур, логически сгруппированных для упрощения разработки программ. В части II описывается работа с фреймворком Foundation. В части III дается обзор фреймворка Cocoa Application Kit и приводится пример разработки простого приложения iPhone с использованием фреймворка UIKit, разработки и отладки кода с помощью Xcode и Interface Builder.

Широкому кругу программистов, от новичков до профессионалов.

ISBN 978-5-9790-0131-9 (рус.)

© ЭКОМ Паблишерз, 2010

ISBN 978-0-321-56615-7 (англ.)

Authorized translation from the English language edition, entitled PROGRAMMING IN OBJECTIVE-C 2.0, 2nd Edition. ISBN 0321566157, by KOCHAN, STEPHEN G.; published by Pearson Education, Inc, publishing as Peachpit Press, Copyright © 2010 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Оглавление

Глава 1. Введение	1
Что вы узнаете из этой книги	2
Как организована эта книга	3
Часть I	
Язык Objective-C 2.0	7
Глава 2. Программирование на Objective-C	9
Компиляция и выполнение программ	9
Xcode	10
Приложение Terminal	16
Объяснение вашей первой программы	18
Вывод значений переменных	22
Упражнения	25
Глава 3. Классы, объекты и методы	27
Понятие объекта	27
Экземпляры и методы	28
Класс Objective-C для работы с дробями	30
Секция @interface	33
Выбор имен	33
Переменные экземпляров	35
Методы класса и методы экземпляра	35
Секция @implementation	37
Секция program	38
Доступ к переменным экземпляра и инкапсуляция данных	44
Упражнения	47
Глава 4. Типы данных и выражения	49
Типы данных и константы	49
Тип int	49
Тип float	50
Тип double	51
Тип char	51
Квалификаторы: long, long long, short, unsigned и signed	53
Тип id	55
Арифметические выражения	56
Старшинство операторов	56
Целочисленная арифметика и унарный оператор «минус»	58
Оператор остатка от деления	60
Преобразования между целыми значениями и значениями с плавающей точкой	62
Оператор приведения типа	63
Операторы присваивания	64
Класс Calculator	65

Битовые операторы	67
Побитовый оператор И	68
Оператор побитового включающего ИЛИ	69
Оператор побитового исключающего ИЛИ	70
Оператор дополнения до единицы	70
Оператор левого сдвига	72
Оператор правого сдвига	72
Типы: _Bool, _Complex и _Imaginary	73
Упражнения	73
Глава 5. Циклы в программах	77
Оператор for	78
Ввод с клавиатуры	84
Вложенные циклы for	86
Варианты цикла for	88
Оператор while	89
Оператор do	94
Оператор break	95
Оператор continue	96
Упражнения	96
Глава 6. Принятие решений	99
Оператор if	99
Конструкция if-else	104
Составные операции сравнения	106
Вложенные операторы if	109
Конструкция else if	111
Оператор switch	120
Булевые переменные	124
Условный оператор	128
Упражнения	130
Глава 7. Более подробно о классах	133
Разделение файлов объявлений и определений (секции interface и implementation)	133
Синтезируемые методы доступа	139
Доступ к свойствам с помощью оператора «точка»	140
Передача методам нескольких аргументов	141
Методы без имен аргументов	143
Операции над дробями	144
Локальные переменные	146
Аргументы для метода	147
Ключевое слово static	147
Ключевое слово self	150
Выделение памяти и возврат объектов из методов	150
Расширение определений класса и файл секции interface	155
Упражнения	156
Глава 8. Наследование	157
Все начинается с корня	157
Поиск подходящего метода	161

Расширение посредством наследования: добавление новых методов	162
Класс для точки и выделение памяти	166
Директива @class	167
Классы, владеющие своими объектами	171
Замещающие методы	175
Какой из методов выбирается?	177
Замещение метода dealloc и ключевое слово super	179
Расширение через наследование:	
добавление новых переменных экземпляра	181
Абстрактные классы	184
Упражнения	184
Глава 9. Полиморфизм, динамический контроль типов и динамическое связывание	187
Полиморфизм: одно имя, различные классы	187
Динамическое связывание и тип id	191
Проверка на этапе компиляции и проверка на этапе выполнения	193
Тип данных id и статический контроль типов	194
Типы аргументов и возвращаемых значений	
при динамическом контроле типов	195
Как задавать вопросы о классах	196
Обработка исключительных ситуаций с помощью @try	200
Упражнения	203
Глава 10. Более подробно о переменных и типах данных	205
Инициализация классов	205
Снова об области действия	207
Директивы для управления областью действия переменных экземпляра	208
Внешние переменные	209
Статические переменные	211
Описатели хранения для класса	214
auto	214
const	215
volatile	215
Перечислимые типы данных	215
Оператор typedef	219
Преобразования типов данных	220
Правила преобразования	221
Расширение для знака	222
Упражнения	223
Глава 11. Категории и протоколы	225
Категории	225
Некоторые замечания по категориям	230
Протоколы	231
Неформальные протоколы	234
Составные объекты	235
Упражнения	236
Глава 12. Препроцессор	239
Оператор #define	239

Более сложные типы определений	241
Оператор #	246
Оператор ##	247
Оператор #import	247
Условная компиляция	250
Операторы #ifdef, #endif, #else и #ifndef	250
Операторы препроцессора #if и #elif	252
Оператор #undef	253
Упражнения	253
Глава 13. Базовые средства из языка C	255
Массивы	256
Инициализация элементов массива	258
Массивы символов	259
Многомерные массивы	260
Функции	262
Аргументы и локальные переменные	263
Возвращение результатов функций	265
Структуры	271
Инициализация структур	277
Массивы структур	278
Структуры внутри структур	278
Дополнительно о структурах	280
Не забывайте об объектно-ориентированном программировании!	282
Указатели	283
Указатели и структуры	287
Указатели, методы и функции	288
Указатели и массивы	290
Операции с указателями	300
Указатели и адреса памяти	301
Объединения	302
Это не объекты!	305
Различные средства языка	305
Составные литералы	305
Оператор goto	306
Пустой оператор	306
Оператор «запятая»	306
Оператор sizeof	307
Аргументы командной строки	308
Как это действует	310
Факт 1: переменные экземпляра сохраняются в структурах	310
Факт 2: переменная-объект на самом деле является указателем	311
Факт 3: методы и функции, а также выражения с сообщениями – это вызовы функций	311
Факт 4: тип id – это обобщенный тип указателя	311
Упражнения	312

Глава 19. Архивация	435
Архивация со списками свойств XML	435
Архивация с помощью NSKeyedArchiver	437
Написание методов кодирования и декодирования	440
Использование NSData для создания нестандартных архивов	447
Использование архиватора для копирования объектов	450
Упражнения	452
Часть III	
Cocoa и SDK iPhone	453
Глава 20. Введение в Cocoa	455
Уровни фреймворков	455
Cocoa Touch	456
Глава 21. Написание приложений iPhone	459
Комплект разработки программ (SDK) для iPhone	459
Ваше первое приложение iPhone	459
Создание нового проекта приложения iPhone	461
Ввод кода	463
Проектирование интерфейса	467
Калькулятор дробей для iPhone	476
Запуск нового проекта Fraction_Calculator	478
Определение контроллера представлений	480
Класс Fraction	486
Класс Calculator, который работает с дробями	489
Разработка пользовательского интерфейса (UI)	491
Сводка шагов	492
Упражнения	493
Часть IV	
Приложения	495
Приложение А. Словарь	497
Приложение В. Сводка языка Objective-C	505
Диграфы и идентификаторы	505
Символы-диграфы	505
Идентификаторы	506
Комментарии	509
Константы	510
Константы целого типа	510
Константы с плавающей точкой	510
Символьные константы	511
Константы символьных строк	512
Константы перечислимого типа	513
Типы и объявления данных	513
Объявления	513
Базовые типы данных	514
Производные типы данных	516
Перечислимые типы данных	522

typedef	523
Модификаторы типа: const, volatile и restrict	523
Выражения	524
Сводка операторов Objective-C	524
Константные выражения	528
Арифметические операторы	529
Логические операторы	529
Операторы отношения	530
Побитовые операторы	530
Операторы наращивания и уменьшения на 1 (операторы инкремента и декремента)	531
Операторы присваивания	531
Условный оператор	532
Оператор приведения типа	532
Оператор sizeof	533
Оператор «запятая»	533
Базовые операции с массивами	534
Базовые операции со структурами	534
Базовые операции с указателями	535
Составные литералы	537
Преобразование базовых типов данных	538
Классы памяти и область действия	539
Функции	539
Переменные	540
Переменные экземпляра	541
Функции	543
Определение функции	543
Вызов функции	544
Указатели на функции	545
Классы	546
Определение класса	546
Определение категории	551
Определение протокола	552
Объявление объекта	554
Выражения с сообщениями	555
Программные операторы	556
Составные операторы	557
Оператор break	557
Оператор continue	557
Оператор do	557
Оператор for	557
Оператор goto	558
Оператор if	559
Оператор null	559
Оператор return	560
Оператор switch	560
Оператор while	561
Обработка исключений	561
Препроцессор	561

Последовательности из триграмм	562
Директивы препроцессора	562
Заранее определенные идентификаторы	567
Приложение С. Исходный код адресной книги	569
Файл секции interface для AddressCard	569
Файл секции interface для AddressBook	570
Файл секции implementation для AddressCard	570
Файл секции implementation для AddressBook	572
Приложение D. Ресурсы	575
Ответы на вопросы упражнений, опечатки и ир.	575
Язык Objective-C	575
Книги	575
Веб-сайты	576
Язык программирования C	576
Книги	576
Cocoa	576
Книги	577
Веб-сайты	577
Разработка приложений для iPhone и iPod touch	577
Книги	578
Веб-сайты	578
Предметный указатель	579

Глава 1

Введение

Деннис Ритчи (Dennis Ritchie) был первым, кто в начале 1970-х гг. начал программировать на языке С в компании AT&T Bell Laboratories. Однако до конца 1970-х гг. этот язык не получал широкого распространения и поддержки, поскольку не существовало готовых компиляторов с языка С для коммерческого использования вне Bell Laboratories. Росту популярности С способствовал рост популярности операционной системы UNIX, которая почти целиком была написана на С.

В начале 1980-х гг. Бред Кокс (Brad J. Cox) разработал язык Objective-C. Этот язык основывался на языке SmallTalk-80. Objective-C был создан *поверх* языка С, то есть к языку С были добавлены расширения для создания нового языка программирования, который позволял создавать *объекты* и работать с ними.

Компания NeXT Software в 1988 г. лицензировала язык Objective-C, а также разработала его библиотеки и среду разработки под названием NEXTSTEP. В 1992 г. поддержка Objective-C была добавлена в среду разработки GNU организации Free Software Foundation. Авторские права на все продукты Free Software Foundation (FSF) принадлежат FSF. Они выпускаются как лицензия GNU General Public License.

В 1994 г. NeXT Computer и Sun Microsystems выпустили стандартизованную спецификацию системы NEXTSTEP под названием OPENSTEP. Реализация OPENSTEP, выпущенная Free Software Foundation, называется GNUStep. Версия для Linux, включающая также ядро Linux и среду разработки GNUStep, называется поэтому LinuxSTEP.

20 декабря 1996 г. компания Apple Computer объявила, что приобретает NeXT Software, и среда NEXTSTEP/OPENSTEP стала основой для следующей основной версии операционной системы Apple – OS X. Версия компании Apple этой среды разработки была названа Cocoa. Встроенная поддержка языка Objective-C в сочетании со средствами разработки, такими как Project Builder (или его преемника Xcode) и Interface Builder, позволила Apple создать мощную среду разработки для разработки приложений в Mac OS X.

В 2007 г. Apple выпустила обновление языка Objective-C и назвала его Objective-C 2.0. Эта версия языка описывается в настоящем (втором) издании этой книги.

После выпуска iPhone в 2007 г. разработчики программ потребовали, чтобы им предоставили возможности разрабатывать приложения для этого передового устройства. Поначалу компания Apple сомневалась, стоит ли предоставлять разработку приложений сторонним компаниям. Для «умиротворения» требовательных разработчиков компания предоставляла им возможность разработки веб-приложений. Такое веб-приложение выполняется в iPhone под управлением встроенного веб-браузера Safari и требует, чтобы пользователь

подсоединялся к веб-сайту, где содержится приложение, чтобы запускать его. Разработчики были недовольны из-за многих ограничений, присущих веб-приложениям, и вскоре компания Apple объявила, что разработчики смогут разрабатывать так называемые *собственные* (*native*) приложения для iPhone.

Собственное приложение хранится на самом iPhone и выполняется под управлением операционной системы iPhone таким же образом, как встроенные приложения iPhone (такие как Contacts, iPod и Weather). OS iPhone является фактически версией Mac OS X, а это означает, что приложения для iPhone можно разрабатывать и отлаживать, например, на MacBook Pro. И действительно, компания Apple вскоре выпустила мощный комплект разработки программ (Software Development Kit, SDK), который позволил быстро разрабатывать и отлаживать приложения iPhone. Наличие имитатора (*simulator*) iPhone позволило разработчикам отлаживать свои приложения на их компьютерах, без необходимости загрузки и тестирования программ непосредственно на iPhone или iPod Touch.

ЧТО ВЫ УЗНАЕТЕ ИЗ ЭТОЙ КНИГИ

Обдумывая, как писать учебник по Objective-C, я должен был принять важное решение. Как и в других книгах по Objective-C, я мог бы предположить, что читатель уже знает, как писать программы на языке C. Я мог бы обучать этому языку, исходя из возможностей обширной библиотеки процедур, например, в фреймворках Foundation и Application Kit. В некоторых книгах принято обучать использованию средств разработки, таких как Xcode и Interface Builder на Маках.

Однако такой подход имеет несколько проблем. Во-первых, изучение всего языка C перед изучением Objective-C неприемлемо. Процедурный язык C содержит много средств, которые не являются необходимыми для программирования на Objective-C, особенно на уровне новичков. На самом деле обращение к некоторым из этих средств противоречит методологии надежного объектно-ориентированного программирования. Также нет смысла в изучении деталей процедурного языка перед изучением объектно-ориентированного. Это дезориентирует программиста и мешает осваивать объектно-ориентированный подход к программированию. То, что Objective-C является расширением языка C, вовсе не означает, что нужно сначала выучить C.

Поэтому я решил, что не буду начинать с обучения языку C и не буду предполагать, что читатель знает C. Вместо этого я принял необычный подход: обучение языку Objective-C и базовому языку C как одному объединенному языку с точки зрения объектно-ориентированного программирования. Цель этой книги определена ее названием – обучение тому, как программировать на Objective-C. Она не содержит подробных сведений о том, как использовать средства разработки, доступные для ввода и отладки программы, и не содержит полных инструкций по разработке интерактивных графических приложений с помощью Cocoa. Вы можете ознакомиться с этим материалом где-либо еще после того, как научитесь писать программы на Objective-C. На самом деле все это будет намного проще после того, как вы разберетесь с программированием на Objective-C. В этой книге не предполагается, что читатель имеет серьезный опыт

программирования (или вообще имеет такой опыт). И если вы новичок в программировании, то сможете изучать Objective-C как свой первый программный язык.

Эта книга учит языку Objective-C с помощью примеров. Представляя каждое новое средство языка, я обычно привожу пример небольшой законченной программы, иллюстрирующей это средство. Правильно выбранный пример программы действует подобно изображению, которое может заменить тысячу слов. Я настоятельно рекомендую выполнить каждую программу (все они доступны в Интернете) и сравнить результаты, полученные на вашем компьютере, с результатами, показанными в книге. Это позволит вам не только изучать язык и его синтаксис, но и знакомиться с компиляцией и выполнением программ на Objective-C.

Как организована эта книга

Эта книга разделена на три логические части. В части I, «Язык Objective-C 2.0», излагаются основы самого языка. В части II, «Framework Foundation», описывается работа с обширным набором готовых классов, которые образуют фреймворк Foundation. В части III, «Cocoa и SDK iPhone», дается обзор фреймворка Cocoa Application Kit и приводится процесс разработки простого приложения iPhone с использованием фреймворка UIKit, а также разработка и отладка кода с помощью Xcode и Interface Builder.

Фреймворк – это набор классов и процедур, логически сгруппированных для упрощения разработки программ. Многие возможности программирования на Objective-C основываются на использовании существующих разнообразных фреймворков.

В главе 2 вы напишете свою первую программу на Objective-C.

Поскольку это не книга по программированию для Сocoa, здесь не приводится подробное описание графических пользовательских интерфейсов (GUI), и они почти не затрагиваются до части III. Поэтому мне потребовался подход, позволяющий вводить данные в программу и выводить результаты. В большинстве примеров этой книги ввод в программу выполняется с клавиатуры, а вывод выполняется в окне: это терминальное окно (Terminal), если вы используете gcc из командной строки, или консольное окно (Console), если используется Xcode.

В главе 3, «Классы, объекты и методы», излагаются основы объектно-ориентированного программирования. В этой главе вводится некоторая терминология, но она сведена к минимуму. Там же вводится механизм определения класса и средства передачи сообщений экземплярам или объектам. Преподаватели и опытные программисты отметят, что для объявления объектов я использую *статический* контроль типов. Я считаю, что это наиболее подходящий способ для начала обучения, поскольку компилятор может обнаруживать больше ошибок, делая программу более понятной и позволяя начинающему программисту явно объявлять типы данных, когда они известны. Поэтому понятие типа *id* и описание его возможностей не затрагиваются в полной мере до главы 9, «Полиморфизм, динамический контроль типов и динамическое связывание».

В главе 4, «Типы данных и выражения», описываются базовые типы данных Objective-C и их применение в ваших программах.

В главе 5, «Циклы в программах», вводятся три оператора цикла: `for`, `while` и `do`.

Принятие решений является основой любого языка программирования. В главе 6, «Принятие решений» подробно описываются операторы `if` и `switch` языка Objective-C.

В главе 7, «Более подробно о классах», приводится более глубокое изложение работы с классами и объектами. Здесь подробно описывается, как работать с методами, передавать несколько аргументов методам и использовать локальные переменные.

В главе 8, «Наследование», вводится ключевое понятие наследования. Это упрощает разработку программ, поскольку вы можете использовать то, что передается из вышележащих уровней. Наследование и подклассы упрощают изменение и расширение существующих определений классов.

В главе 9 описываются три основополагающие характеристики языка Objective-C. Здесь излагаются три ключевые концепции: полиморфизм, динамический контроль типов и динамическое связывание.

В главах 10–13 завершается описание языка Objective-C. Здесь рассматриваются такие вопросы, как инициализация объектов, протоколы, категории, препроцессор, и некоторые основы языка С: функции, массивы, структуры и указатели. Эти возможности не обязательно использовать (обычно их нужно избегать) в начальный период разработки объектно-ориентированных приложений. Рекомендуется пропустить главу 13 при первом чтении этой книги и возвращаться к ней по мере необходимости для изучения какого-либо конкретного средства языка С.

Часть II начинается с главы 14, «Введение в Foundation Framework», где дается введение во фреймворк Foundation и описывается доступ к его документации.

В главах 15–19 описываются важные возможности Foundation framework. Это числовые и строковые объекты, коллекции, файловая система, управление памятью и процесс копирования и архивации объектов.

Закончив работу с частью II, вы сможете разрабатывать довольно сложные программы на Objective-C, которые используют фреймворк Foundation.

Часть III начинается с главы 20, «Введение в Cocoa». Здесь дается краткий обзор фреймворка Application Kit, содержащего классы, необходимые для разработки сложных графических приложений на Маках.

В главе 21, «Написание приложений iPhone», дается введение в SDK iPhone и фреймворк UIKit. Здесь показан пошаговый подход к написанию простого приложения iPhone (или iTouch) и рассматривается приложение-калькулятор, позволяющее выполнять простые арифметические вычисления с дробями с помощью iPhone.

Поскольку объектно-ориентированный подход требует использования довольно обширной терминологии, в приложении А, «Словарь», приводятся определения некоторых распространенных терминов.

В приложении В, «Сводка языка Objective-C», приводится сводка языка Objective-C для быстрого поиска нужной информации.

В приложении С, «Исходный код адресной книги» приводится листинг исходного кода для двух классов, которые были разработаны и широко использовались в части II. В этих классах определяются классы для адресных карточек и адресной книги. Методы этих классов позволяют выполнять такие простые операции, как добавление и удаление адресных карточек из адресной книги, поиск нужного адресата, вывод содержимого адресной книги и т.д.

Изучив написание программ на Objective-C, можно продолжить работу в нескольких направлениях. Можно изучить более глубоко базовый язык программирования C, начать разрабатывать программы Сосоа для выполнения в Mac OS X или более сложные приложения iPhone. В любом случае, приложение D, «Ресурсы» поможет вам продвинуться в нужном направлении.

Глава 2

Программирование на Objective-C

В этой главе мы сразу приступим к делу и напишем первую программу на Objective-C. Пока мы не будем работать с объектами – это тема следующей главы. Сначала необходимо понять, что такое ввод программы, ее компиляция и выполнение. Это важно при программировании и в Windows, и на компьютерах Macintosh.

Для начала рассмотрим простую программу, которая выводит на экране фразу «Programming is fun!» (Программировать весело!). Эту задачу выполняет программа «Программа 2.1», приведенная ниже.

Программа 2.1

```
// First program example (Первый пример программы)

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Programming is fun!");
    [pool drain];
    return 0;
}
```

Компиляция и выполнение программ

Прежде чем приступить к подробному описанию программы, рассмотрим ее компиляцию и выполнение. Для компиляции и выполнения этой программы можно использовать Xcode или компилятор GNU Objective-C в окне Terminal. Мы рассмотрим оба метода, чтобы вы могли решить, каким из них будете работать с программами в остальной части книги.

Примечание. Соответствующие средства уже предустановлены на всех Маках, которые поставляются с системами OS X. Если вы устанавливаете OS X отдельно, установите также Developer Tools.

Xcode

Xcode — это мощное приложение, позволяющее вводить, компилировать, отлаживать и выполнять программы. Если вы намерены разрабатывать приложения на Маке, вам необходимо освоить это мощное инструментальное средство. Здесь вы только ознакомитесь с ним, в дальнейшем мы разработаем с его помощью одно графическое приложение.

Xcode находится в папке Developer внутри подпапки Applications. На рис. 2.1 показан значок Xcode.

Запустите Xcode. В меню File (Файл) выберите New Project (Новый проект), см. рис. 2.2.



Рис. 2.1. Значок Xcode

Появится окно, показанное на рис. 2.3.

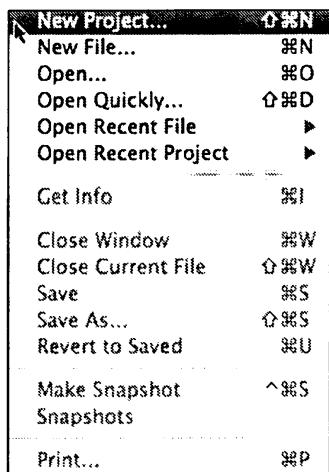


Рис. 2.2. Запуск нового проекта

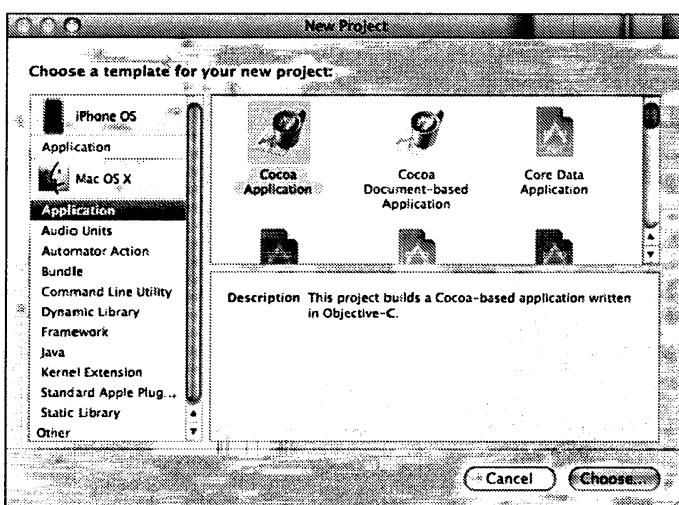


Рис. 2.3. Запуск нового проекта: выбор типа приложения

Выполните прокрутку в левой панели, пока не появится Command Line Utility (Утилита командной строки). В правой панели выделите Foundation Tool. Теперь появится окно, показанное на рис. 2.4.

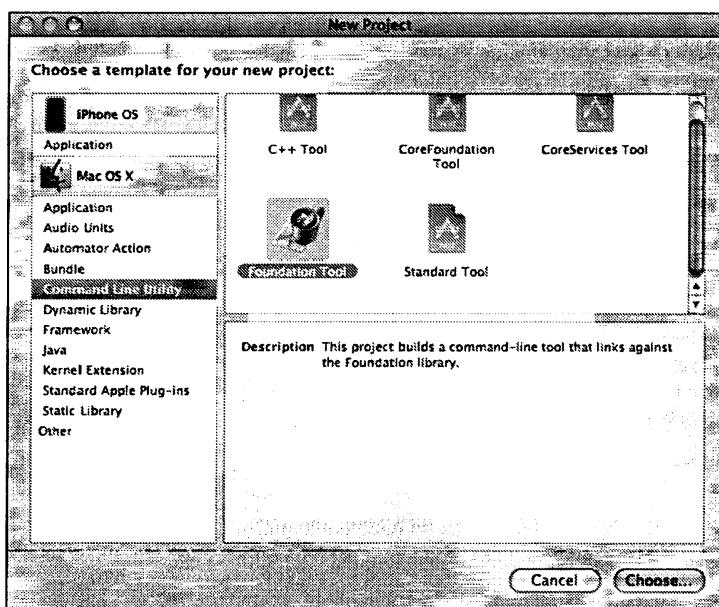


Рис. 2.4. Запуск нового проекта: создание инструмента Foundation

Шелкните на Choose (Выбрать). Появится новое окно (рис. 2.5).

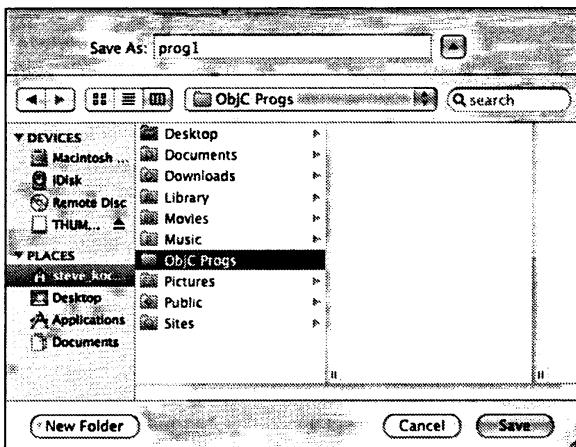


Рис. 2.5. Окно списка файлов Xcode

Мы назовем эту программу *prog1*. Введите это имя в поле **Save As** (**Сохранить как**). Возможно, для проектов этой книги следует создать отдельную папку. Я держу проекты для этой книги в папке *ObjC Progs*.

Щелкните на кнопке **Save** (**Сохранить**), чтобы создать новый проект. Появится окно проекта (рис. 2.6).

Ваше окно может выглядеть по-другому, если вы уже работали с Xcode или изменили какие-то параметры Xcode.

Теперь пора ввести первую программу. Выберите файл *prog1.m* в правой верхней панели. Окно Xcode показано на рис. 2.7.

В таблице 2.1 приводится список распространенных расширений имен файлов. Расширение имени для исходных файлов Objective-C – *.m*.

Табл. 2.1. Распространенные расширения имен файлов

Расширение	Описание
<i>.c</i>	Исходный файл на языке C
<i>.cc</i> , <i>.cpp</i>	Исходный файл на языке C++
<i>.h</i>	Файл заголовка (Header)
<i>.m</i>	Исходный файл Objective-C
<i>.mm</i>	Исходный файл Objective-C++
<i>.pl</i>	Исходный файл Perl
<i>.o</i>	Объектный (компилированный) файл

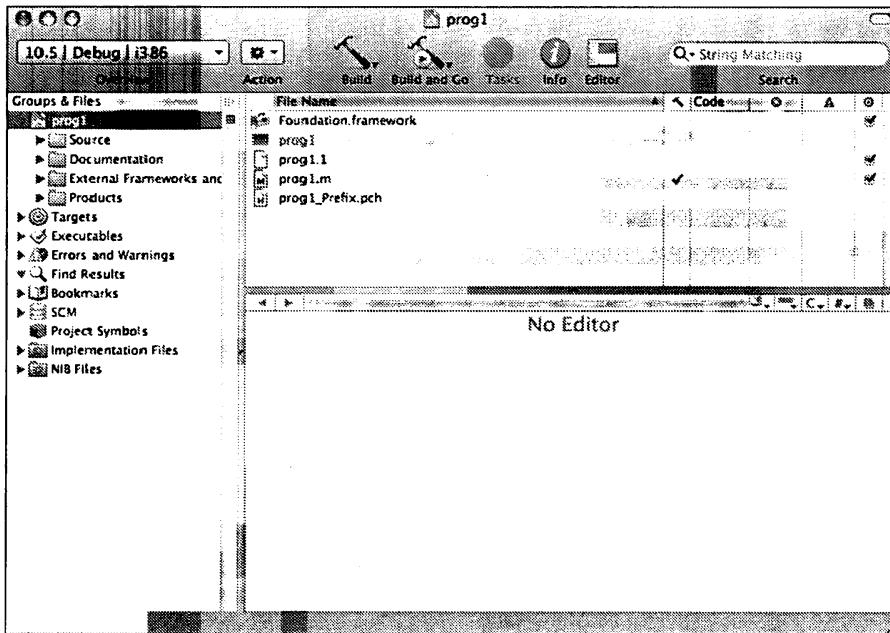


Рис. 2.6. Окно проекта Xcode prog1

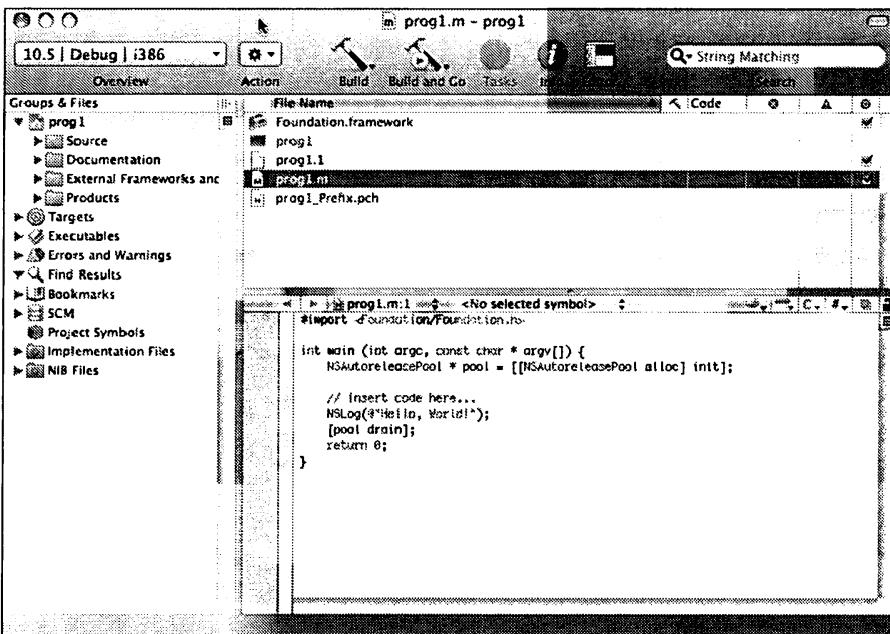


Рис. 2.7. Файл prog1.m и окно редактирования (edit)

В окне проекта Xcode (внизу справа) показан файл с именем prog1.m, содержащий следующие строки:

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

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

    // insert code here... (введите здесь код)
    NSLog (@"Hello World!");
    [pool drain];
    return 0;
}
```

Примечание. Если у вас не отображается содержимое этого файла, попробуйте щелкнуть и вытянуть нижнюю правую панель, чтобы снова появилось окно редактирования. Это может произойти, если вы уже работали с Xcode.

В этом окне Xcode предоставляет файл шаблона, который вы можете редактировать. Внесите изменения, соответствующие тексту программы 2.1. Стока в начале файла prog1.m, которая начинается с двух слэшей (//), называется комментарием; ниже мы поговорим о них более подробно.

Ваша программа в окне редактирования должна выглядеть следующим образом.

```
// First program example

#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Programming is fun!");

    [pool drain];
    return 0;
}
```

Обратите внимание на цвета текста. Xcode выделяет разными цветами значения, зарезервированные слова и т.д.

Теперь можно скомпилировать и выполнить программу. В терминологии Xcode эти этапы называются *build and run* (Сборка и выполнение). Сначала следует сохранить программу, выбрав пункт *Save* в меню *File*. Если попытаться скомпилировать и выполнить программу без сохранения файла, Xcode спросит, хотите ли вы сохранить его.

В меню Build (Сборка) можно выбрать Build или Build and Run. Выберите второй вариант, поскольку программа, скомпилированная без ошибок, будет выполнена автоматически.

Можно также щелкнуть на значке Build and Go, который показан в панели инструментов.

Примечание. Build and Go означает «Выполнить сборку и затем запустить последнюю операцию, которую я просил выполнить». Это может быть Run, Debug, Run with Shark or Instruments и т.д. При первом использовании для проекта Build and Go означает «собрать и выполнить программу» (Build and Run). Однако в дальнейшем «Build and Go» и «Build and Run» не всегда означают одно и то же.

Если в программе имеются ошибки, то на этом этапе вы увидите сообщение об ошибках. Вернитесь назад, исправьте ошибки и повторите процесс. После устранения всех ошибок появится окно prog1 – Debugger Console (Консоль отладчика). Это окно содержит выходные результаты программы (рис. 2.8). Если это окно не появляется автоматически, перейдите в линейку главного меню и выберите Console в меню Run. Содержимое окна Console будет описано ниже.

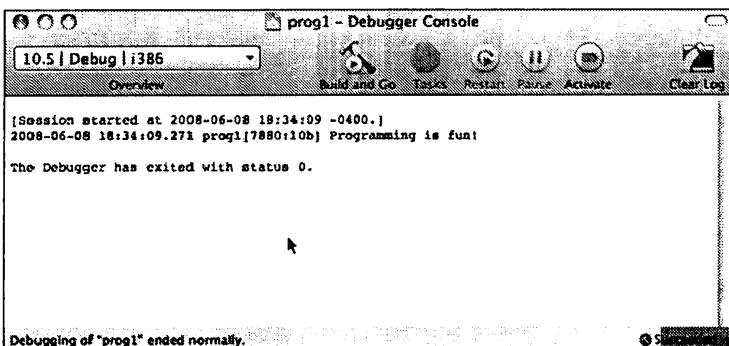


Рис. 2.8. Окно Xcode Debugger Console

Вы закончили процедурную часть компиляции и выполнения программы с помощью Xcode. Ниже приводится последовательность шагов для создания новой программы с помощью Xcode.

1. Запустите приложение Xcode.
2. Если это новый проект, выберите File, New Project.
3. Для типа приложения выберите Command Line Utility, Foundation Tool и щелкните на кнопке Choose.

4. Выберите имя проекта и папку для сохранения в ней файлов проекта. Щелкните на кнопке Save.
5. В верхней правой панели вы увидите файл prog1.m (или то имя, которое вы назначили своему проекту, с расширением имени .m). Выделите этот файл. Введите текст программы в окне редактирования (edit), которое появится непосредственно под этой панелью.
6. Сохраните изменения, выбрав File, Save.
7. Запустите сборку и выполнение приложения, выбрав Build, Build and Run или щелкнув на кнопке Build and Go.
8. В случае ошибок при компиляции внесите в программу изменения и повторите шаги 6 и 7.

Приложение Terminal

Если вы привыкли использовать оболочку UNIX и средства командной строки, то можете редактировать, компилировать и выполнять программы, используя приложение Terminal. Ниже описывается, как это делать.

Первый шаг – это запуск приложения Terminal на вашем Mac OS X. Приложение Terminal находится в папке Applications внутри Utilities. На рис. 2.9 показан его значок.

Запустите приложение Terminal. Появится окно, аналогичное рис. 2.10.



Terminal

Рис. 2.9. Значок программы Terminal

Команды вводятся в каждой строке после символа \$ (или % – в зависимости от настройки вашего приложения Terminal). Это вполне понятно, если вы знакомы с использованием UNIX.

Сначала нужно ввести в файл строки из программы 2.1. Можно начать с создания папки, в которой будут сохраняться примеры ваших программ. Затем для ввода вашей программы нужно запустить текстовый редактор, например, vi или emacs.

sh-2.05a\$ mkdir Progs	<i>Создание папки для сохранения программ</i>
sh-2.05a\$ cd Progs	<i>Переход в новую папку</i>
sh-2.05a\$ vi prog1.m	<i>Запуск текстового редактора для ввода программы</i>

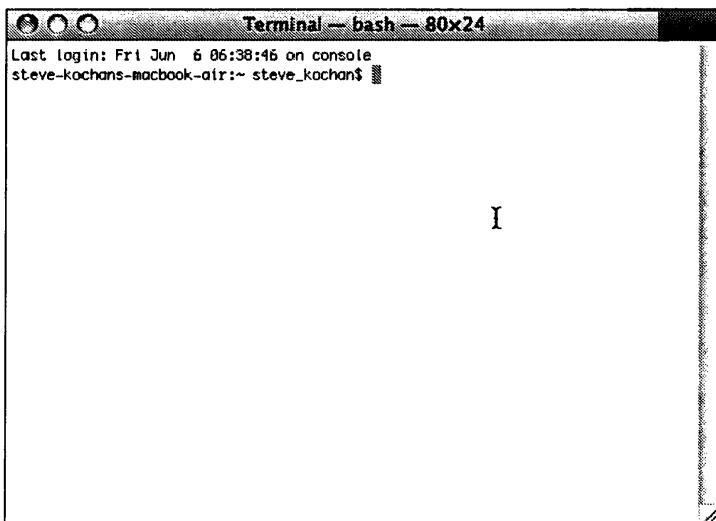


Рис. 2.10. Окно Terminal

Примечание. В показанном примере и в остальной части книги команды, которые вводятся пользователем, выделены полужирным начертанием.

Для файлов Objective-C можно выбрать любое имя, но в конце имени должны стоять символы .m, которые указывают компилятору, что используется Objective-C.

Для файла с текстом программы можно выполнять компиляцию и сборку с помощью компилятора GNU Objective-C, который называется gcc. Команда gcc имеет следующий формат:

```
gcc -framework Foundation файлы -o имя_программы
```

Опция -framework Foundation указывает, что используется информация о Foundation framework.

Эта опция должна быть указана в командной строке. *Файлы* – это список файлов для компиляции. В нашем примере такой файл только один (с именем prog1.m). *Имя_программы* – это имя исполняемого файла, который будет создан, если файл скомпилируется без ошибок.

Эта программа называется prog1. Ниже приводится командная строка для компиляции этой программы:

```
$ gcc -framework Foundation prog1.m -o prog1 Компилировать prog1.m и  
назвать ее prog1  
$
```

Возврат символа командной строки без каких-либо сообщений означает, что в программе не обнаружено ошибок. Вы можете выполнить такую программу, введя имя prog1 в командной строке:

```
$ prog1      Выполнение prog1
```

```
sh: prog1: command not found (команда не найдена)
$
```

Вы можете получить этот результат, если раньше не использовали Terminal. Оболочка UNIX (то есть приложение, выполняющее вашу программу), «не знает», где находится `prog1` (мы не будем подробно объяснять это здесь). Есть два варианта решения этой проблемы. Первый – поставить перед именем программы символы `./`, чтобы оболочка искала программу в текущей папке. Второй – добавить папку, в которой хранится ваша программа (или просто текущей папки), к переменной оболочки PATH. Мы используем первый способ:

```
$ ./prog1      Выполнение prog1
2008-06-08 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

Написание и отладка программ Objective-C из среды Terminal – вполне допустимый, но стратегически ошибочный подход. Приложения Mac OS X или iPhone – это не просто исполняемый файл, который должен быть «упакован» в пакет приложения. Создать такие приложения из приложения Terminal очень нелегко, в отличие от специализированной среды Xcode. Поэтому далее мы займемся изучением Xcode как среды для разработки ваших программ.

Объяснение вашей первой программы

Познакомившись с компиляцией и выполнением программы Objective-C, рассмотрим эту программу более подробно. Еще раз приведем ее текст.

```
// First program example (Первый пример программы)

#import <Foundation/Foundation.h>

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

    NSLog(@"%@", @"Programming is fun!");
    [pool drain];
    return 0;
}
```

В Objective-C прописные и строчные буквы различаются. Кроме того, для Objective-C не имеет значения, с какого места строки вы начинаете ввод: вводить символы можно с любой позиции строки. Это позволяет сделать программы более удобными для чтения.

В первой строке программы мы встречаем понятие *комментария*:

```
// First program example
```

Комментарий используется для документирования программы и удобства ее чтения. Из комментариев читатель программы (программист или человек, поддерживающий программу) видит назначение и логику определенной программы или последовательности операторов.

Комментарии можно вставлять в программу Objective-C двумя способами. Один из них – это использование двух последовательных слэшей (//). Компилятор игнорирует все символы после слэшей, вплоть до конца строки.

Комментарий можно также инициировать с помощью символов / и *. Комментарий начинается с символов /* и завершается символами */ с любым числом предшествующих пробелов. Все символы между открывающими /* и закрывающими */ считаются частью комментария и игнорируются компилятором Objective-C. Эта форма комментария часто используется, если комментарий занимает несколько строк:

```
/*
```

Этот файл реализует класс с именем Fraction, который представляет дробные числа. Используются методы, которые позволяют работать с дробями, например, addition (сложение), subtraction (вычитание) и т.д.

Более подробную информацию см. в документе:

/usr/docs/classes/fractions.pdf

```
*/
```

Вы сами выбираете стиль комментариев. Вложенность комментариев в стиле /* не допускается.

Имеются три причины для того, чтобы вставлять комментарии в программу. Во-первых, намного проще документировать программу, когда вы еще держите в уме ее логику, чем восстанавливать ее логику после того, как программа завершена. Во-вторых, при вставке комментариев на ранних стадиях вы облегчаете себе этап отладки, когда происходит выявление и отладка ошибок. Комментарий не только помогает вам (и другим) в чтении программы, но и помогает выяснить путь к источнику логической ошибки. И последнее – я не встречал ни одного программиста, которому нравилось бы документирование программы, так что после окончания отладки программы вы вряд ли займетесь вставкой комментариев. Менее скучно – вставлять их при разработке программы.

В следующей строке программы 2.1 вы указываете компилятору, что нужно найти и обработать файл с именем Foundation.h:

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

Это системный файл. #import указывает, что нужно импортировать или включить в программу информацию из этого файла так, как если бы ввели ее в этом месте вручную. Импорт файла Foundation.h выполняется потому, что он содержит классы и функции, используемые в этой программе.

Следующая строка указывает, что программа имеет имя `main`:

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

Это специальное имя, которое указывает, где должно начаться выполнение программы. Зарезервированное слово `int`, которое поставлено перед `main`, указывает тип значения, которое возвращает `main` – целое (более подробно об этом см. ниже). Между открывающей и закрывающей круглыми скобками находятся *аргументы командной строки*, которые мы будем рассматривать в главе 13.

Теперь, после идентификации `main` для системы, вы можете задать, какие действия выполняет программа. Для этого все *программные операторы* должны быть заключены в фигурные скобки. В самом простом случае *оператор (statement)* – это просто выражение, которое заканчивается символом «точка с запятой». Система интерпретирует все программные операторы между фигурными скобками как часть процедуры `main`. Программа 2.1 содержит четыре оператора. Первый оператор в программе 2.1 имеет следующий вид:

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

Он резервирует пространство в памяти для *автоматически высвобождаемого пула (autorelease pool)*. Мы опишем этот оператор в главе 17. Xcode помещает эту строку в вашу программу автоматически как часть соответствующего шаблона, поэтому не обращайте пока на это внимание.

Следующий оператор указывает, что должна быть *вызвана (call)* процедура `NSLog`. Параметр, или *аргумент (argument)*, должен быть передан процедуре `NSLog` в виде следующей строки символов:

```
@"Programming is fun!"
```

Здесь символ @ стоит непосредственно перед строкой символов, заключенной в кавычки. Все вместе это называется объектом-константой типа `NSString`.

Примечание. Если вы программируете на C, то вам может быть непонятен смысл символа-префикса @. Без этого символа вы получаете строку-константу в стиле C; с ним вы получаете объект-строку типа `NSString`.

Процедура `NSLog` – это функция библиотеки Objective-C, которая просто выводит на экран свой аргумент (или аргументы). Но прежде чем сделать это, она выводит дату и время выполнения процедуры, имя программы и некоторые другие числовые данные. В следующих главах мы не будем показывать текст, который вставляется функцией `NSLog` перед выходными результатами.

Вы должны заканчивать все операторы программы в Objective-C символом «точка с запятой» (;). Точка с запятой ставится непосредственно после закрывающей круглой скобки при вызове `NSLog`. Прежде чем выполнить выход из программы, вы должны освободить выделенный пул памяти (и связанные с ним объекты). Для этого вводится следующая строка:

```
[pool drain];
```

В данном случае Xcode тоже автоматически вставляет в программу эту строку. Ее подробное описание будет приведено позже. Последний оператор программы в процедуре `main` имеет следующий вид:

```
return 0;
```

С его помощью завершается выполнение `main` и *возвращается (return)* значение состояния, равное 0, которое указывает на нормальное завершение программы. Любое ненулевое значение обычно означает, что возникла проблема, например, программа не смогла найти нужный файл.

На рис. 2.8. в окне Debug Console после строки результатов `NSLog` было выведено следующее сообщение:

```
The Debugger has exited with status 0. (Выход из отладчика со значением состояния 0)
```

Теперь вы понимаете, что означает это сообщение.

Теперь внесем в программу изменения. Пусть она выводит также фразу «*And programming in Objective-C is even more fun!*» (А программировать Objective-C еще интереснее!). Это можно сделать, добавив еще один вызов процедуры `NSLog`, как показано ниже в программе 2.2. Напомним, что каждый оператор программы на Objective-C должен заканчиваться символом «точка с запятой».

Программа 2.2

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

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

    NSLog(@"%@", @"Programming is fun!");
    NSLog(@"%@", @"Programming in Objective-C is even more fun!");

    [pool drain];
    return 0;
}
```

Если ввести программу 2.2 с последующей компиляцией и выполнением, то получится следующий вывод (мы опустили текст, который обычно выводится процедурой `NSLog` перед результатами программы):

Вывод программы 2.2

```
Programming is fun!  
Programming in Objective-C is even more fun!
```

Как можно видеть из следующего примера, вы не обязаны вызывать процедуру `NSLog` для каждой строки вывода.

Рассмотрим специальный набор из двух символов. Обратный слэш в сочетании с буквой `n` используются как признак новой строки. Этот символ указывает системе, что нужно перейти на новую строку. Все символы, которые выводятся после символа новой строки, появятся в следующей строке вывода. Символ новой строки аналогичен клавише возврата каретки на пишущей машинке. Изучите листинг программы 2.3 и попытайтесь предсказать результаты, прежде чем посмотреть вывод этой программы.

Программа 2.3

```
#import <Foundation/Foundation.h>  
  
int main (int argc, const char *argv[])  
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
    NSLog (@"Testing...\n..1\n...2\n....3");  
    [pool drain];  
    return 0;  
}
```

Вывод программы 2.3

```
Testing...  
.1  
.2  
.3
```

Вывод значений переменных

С помощью `NSLog` можно выводить не только простые фразы, но и значения переменных или результаты вычислений. В программе 2.4 для вывода результатов сложения двух чисел, 50 и 25, используется процедура `NSLog`.

Программа 2.4

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

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
int sum;  
  
sum = 50 + 25;  
NSLog (@"The sum of 50 and 25 is %i", sum);  
[pool drain];  
  
return 0;  
}
```

Вывод программы 2.4

The sum of 50 and 25 is 75 (Сумма 50 и 25 равна 75)

Первый оператор программы внутри `main` после `autorelease pool` определяет переменную `sum` типа `integer`. Вы должны определить все переменные программы перед использованием. Определение переменной указывает компилятору Objective-C, как переменная должна использоваться программой. Эта информация требуется компилятору, чтобы он сформировал соответствующие инструкции для сохранения и считывания значений в этой переменной. Переменную, которая определена с типом `int`, можно использовать для хранения только целых значений, то есть значений без цифр после десятичной точки. Примеры целых значений: 3, 5, -20 и 0. Числа с цифрами после десятичной точки, например, 2.14, 2.455 и 27.0, называются числами *с плавающей запятой* (*floating-point*), или вещественными числами.

В целой переменной `sum` сохраняется результат сложения двух целых чисел, 50 и 25. Мы преднамеренно оставили пустую строку после определения этой переменной, чтобы визуально отделить объявление переменной от операторов программы. Иногда добавление одной пустой строки в программе может сделать программу более удобной для чтения.

Соответствующий оператор программы выглядит так же, как в большинстве языков программирования:

```
sum = 50 + 25;
```

Число 50 добавляется (что указано знаком «плюс») к числу 25, и результат сохраняется (что указано оператором присваивания, то есть знаком «равно») в переменной `sum`.

Вызов процедуры `NSLog` в программе 2.4 теперь содержит два аргумента, заключенных в круглые скобки. Эти аргументы разделены запятой. Первый аргумент при обращении к процедуре `NSLog` — это всегда символьная строка, которая должна быть выведена на экран. Но вместе с символьной строкой часто требуется выводить значения определенных переменных. В данном случае нужно вывести значение переменной `sum` после вывода следующей строки символов:

The sum of 50 and 25 is

Символ процента внутри первого аргумента – это специальный символ, распознаваемый функцией `NSLog`. Символ, который следует непосредственно за символом процента, указывает тип отображаемого значения. В приведенной программе процедура `NSLog` интерпретирует букву `i` как указание на вывод целого значения.

Если процедура `NSLog` обнаруживает символы `%i` внутри строки символов, она автоматически выводит значение следующего переданного ей аргумента. Поскольку следующим аргументом для `NSLog` является переменная `sum`, ее значение автоматически выводится после текста «`The sum of 50 and 25 is`».

Теперь попытайтесь предсказать, что выведет программа 2.5.

Программа 2.5

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

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

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int value1, value2, sum;

    value1 = 50;
    value2 = 25;
    sum = value1 + value2;

    NSLog (@"The sum of %i and %i is %i", value1, value2, sum);

    [pool drain];
    return 0;
}
```

Вывод программы 2.5

`The sum of 50 and 25 is 75`

Второй оператор программы внутри `main` определяет три переменные с именами `value1`, `value2` и `sum`, и все они имеют тип `int`. Этот оператор эквивалентен трем операторам:

```
int value1;
int value2;
int sum;
```

После определения этих переменных программа присваивает значение 50 переменной `value1` и значение 25 переменной `value2`. Затем вычисляется сумма этих переменных, и результат присваивается переменной `sum`.

Вызов процедуры `NSLog` теперь содержит четыре аргумента. Здесь тоже первый аргумент, который обычно называют *строкой формата* (*format string*), указывает системе, как выводить остальные аргументы. Значение `value1` должно быть выведено непосредственно после фразы «*The sum of*». Аналогичным образом, значения `value2` и `sum` должны быть выведены в местах, указанных следующими двумя экземплярами символов `%i` в строке формата.

Упражнения

1. Введите и выполните пять программ, описанных в этой главе. Сравните выходные результаты каждой программы с выводом, показанным после каждой программы.
2. Напишите программу, которая выводит следующий текст:

In Objective-C, lowercase letters are significant.
main is where program execution begins.
Open and closed braces enclose program statements in a routine.
All program statements must be terminated by a semicolon.
(В Objective-C строчные буквы отличаются от прописных.)

Выполнение программы происходит в `main`.

Операторы программы находятся между открывающей и закрывающей фигурными скобками.

Все операторы программы должны заканчиваться символом «точка с запятой».

3. Какие результаты выведет следующая программа?

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

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

    i = 1;
    NSLog (@"Testing...");
    NSLog (@"....%i", i);
    NSLog (@"...%i", i + 1);
    NSLog (@"..%i", i + 2);

    [pool drain];
    return 0;
}
```

4. Напишите программу, которая вычитает значение 15 из 87 и выводит результат вместе с сообщением.
5. Укажите синтаксические ошибки в программе, затем введите и выполните исправленную программу, чтобы убедиться, что выявлены все ошибки.

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

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

    INT sum;
    /* COMPUTE RESULT //
    sum = 25 + 37 - 19
    / DISPLAY RESULTS /
    NSLog (@'The answer is %i' sum);

    [pool drain];
    return 0;
}
```

6. Какие результаты выведет следующая программа?

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

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

    int answer, result;

    answer = 100;
    result = answer - 10;

    NSLog (@"The result is %i\n", result + 5);

    [pool drain];
    return 0;
}
```

Глава 3

Классы, объекты и методы

В этой главе вы узнаете о ключевых концепциях в объектно-ориентированном программировании, начнете работать с классами в Objective-C и познакомитесь с основной терминологией. Мы опишем только базовые терминов, чтобы не перегружать вас излишней информацией. Подробное и точное определение терминов см. в «Словаре» (приложение A).

Понятие объекта

Объект (object) – это определенная «вещь». В объектно-ориентированном программировании объект – это и вещь, и действия, которые с ней можно делать. Этим объектно-ориентированные языки программирования отличаются, например, от языка C, который является процедурным языком. В C сначала думают о том, что хотят сделать, и только затем заботятся об объектах, что почти противоположно объектно-ориентированному подходу.

Рассмотрим пример из повседневной жизни. Предположим, у вас есть автомобиль. Это объект, которым вы владеете. Это не какой-либо произвольный автомобиль; это определенный автомобиль, который собран на заводе в Детройте или в Японии, или где-то еще. У него есть идентификационный номер (vehicle identification number, VIN), который уникально идентифицирует этот автомобиль.

В объектно-ориентированной терминологии ваш автомобиль является экземпляром (*instance*) автомобиля. Развивая эту терминологию, автомобиль (*car*) – имя класса, из которого этот экземпляр был создан. При выпуске каждого нового автомобиля создается новый экземпляр данного класса автомобилей, и каждый экземпляр автомобиля называется объектом.

Ваш автомобиль может иметь серебристую окраску, может быть с откидным или жестким верхом, и т.д. Вы выполняете определенные действия со своим автомобилем – водите его, заправляете бензином, моете, выполняете техническое обслуживание и т.д.

Действия, приведенные в таблице 3.1, можно выполнять с вашим автомобилем и с любыми другими автомобилями. Например, ваша сестра водит свой автомобиль, моет его, заправляет его бензином и т.д.

Табл. 3.1. Действия над объектами

Объект	Что вы делаете с ним
Ваш автомобиль	Водите. Заправляете бензином. Моете. Выполняете техническое обслуживание.

Экземпляры и методы

Уникальная реализация класса – это экземпляр. Действия, которые выполняются над экземпляром, называются *методами* (*method*). В некоторых случаях метод может применяться к экземпляру класса или к самому классу. Например, мытье автомобиля применяется к экземпляру (все методы, перечисленные в таблице 3.1, являются методами для экземпляра). Определение количества типов автомобилей, выпускаемых изготовителем, применяется к классу, то есть это метод для класса.

Предположим, что у нас имеются два автомобиля, которые сошли со сборочного конвейера и, казалось бы, идентичны: они имеют одинаковый салон, одинаковую окраску и т.д. Они могут одинаково запускаться, но поскольку каждый автомобиль используется его собственным владельцем, он приобретает свои уникальные характеристики – на одном автомобиле появится царапина, а другой проедет больше километров. Каждый экземпляр, или объект, содержит не только информацию о своих собственных характеристиках, полученных на заводе, но и свои текущие характеристики. Эти характеристики могут изменяться динамически. Когда вы ведете автомобиль, бензина в баке становится меньше, автомобиль загрязняется, шины изнашиваются.

Применение метода к объекту может влиять на *состояние* (*state*) этого объекта. Если метод определяется как «заправить мой автомобиль бензином», то после выполнения этого метода будет заполнен бензобак вашего автомобиля. Таким образом, метод повлияет на состояние бака автомобиля.

Объекты являются уникальными представителями класса, и каждый объект содержит некоторую информацию (данные), которые обычно являются частными для этого объекта. Методы – это средства доступа и изменения этих данных.

Язык программирования Objective-C имеет следующий синтаксис для применения методов к классам и экземплярам:

[Класс-или-Экземпляр метод];

В этом синтаксисе после левой прямоугольной скобки следует имя класса или экземпляра этого класса, а затем (после одного или нескольких пробелов) – метод, который нужно выполнить. В конце ставится правая прямоугольная скобка и завершающая точка с запятой. Если вы обращаетесь к классу или экземпляру для выполнения некоторого действия, можно сказать, что вы отправляете

ему *сообщение* (*message*); это сообщение принимает *получатель* (*receiver*). Поэтому показанный выше формат можно описать следующим образом:

[получатель сообщение] ;

Вернемся к списку таблицы 3.1 и перепишем ее с помощью этого синтаксиса. Но прежде чем сделать это, потребуется получить ваш новый (new) автомобиль (*yourCar*). Для этого нужно обратиться на завод, например, в такой форме:

yourCar = [Car new]; получение нового автомобиля

Вы отправляете сообщение классу Car (получателю сообщения), запрашивая у него получение нового автомобиля. Результирующий объект (представляющий ваш уникальный автомобиль) сохраняется в переменной *yourCar*. С этого момента *yourCar* можно использовать для ссылки на ваш экземпляр автомобиля.

Поскольку вы обращаетесь на завод для получения автомобиля, метод new называется методом *завода* или методом *класса*. Остальные действия для вашего нового автомобиля будут методами экземпляра, поскольку они применяются именно к вашему автомобилю. Ниже приводятся примеры выражений с сообщениями, которые можно написать для вашего автомобиля.

[<i>yourCar prep</i>];	<i>подготовка для первого использования</i>
[<i>yourCar drive</i>];	<i>вождение вашего автомобиля</i>
[<i>yourCar wash</i>];	<i>мытье вашего автомобиля</i>
[<i>yourCar getGas</i>];	<i>заправка бензином вашего автомобиля, если это требуется</i>
[<i>yourCar service</i>];	<i>обслуживание вашего автомобиля</i>

[*yourCar topDown*]; *если он с откидным верхом*
[*yourCar topUp*];
currentMileage = [suesCar currentOdometer];

В последнем примере используется метод экземпляра, который возвращает определенную информацию: текущий километраж (в милях), показанный одометром. Эта информация сохраняется в переменной *currentMileage*.

Ваша сестра Сью (Sue) может использовать те же методы для своего экземпляра автомобиля:

[*suesCar drive*];
[*suesCar wash*];
[*suesCar getGas*];

Применение одних и тех же методов к разным объектам является одной из ключевых концепций объектно-ориентированного программирования.

Работа с автомобилями вряд ли потребуется в ваших программах. Вашиими объектами будут компьютерно-ориентированные элементы, такие как окна, прямоугольники, фрагменты текста, калькулятор или список воспроизведения. Ваши методы будут выглядеть так же, как методы, используемые для автомобилей:

[<i>myWindow erase</i>];	<i>Очистка окна</i>
[<i>myRect getArea</i>];	<i>Расчет площади прямоугольника</i>

[userText spellCheck];	<i>Проверка правописания в определенном тексте</i>
[deskCalculator clearEntry];	<i>Стирание последней записи</i>
[favoritePlaylist showSongs];	<i>Показ песен в списке воспроизведения избранного</i>
[phoneNumber dial];	<i>Набор номера телефона</i>

Класс Objective-C для работы с дробями

Теперь мы можем определить какой-либо конкретный класс в Objective-C и научиться работать с экземплярами этого класса.

Как всегда, начнем с изучения процедуры. Предположим, что вам нужно написать программу для работы с дробями. Вам могут потребоваться операции сложения, вычитания, умножения и т.д. Если вы не работали с классами, то можете начать с простой программы, которая приводится ниже.

Программа 3.1

```
// Simple program to work with fractions (Простая программа для работы с дробями)
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int numerator = 1;
    int denominator = 3;
    NSLog (@"The fraction is %i/%i", numerator, denominator);

    [pool drain];
    return 0;
}
```

Вывод программы 3.1

The fraction is 1/3 (Дробь равна 1/3)

В программе 3.1 дробь представлена своими числителем (`numerator`) и знаменателем (`denominator`). После создания автоматически высвобождаемого пула (`autorelease pool`) в двух строках процедуры `main` объявляются две переменные целого типа — `numerator` и `denominator`, которым присваиваются целые значения — соответственно 1 и 3. Это эквивалентно следующим строкам:

```
int numerator, denominator;
numerator = 1;
denominator = 3;
```

Мы представили дробь 1/3, сохранив значение 1 в переменной `numerator` и значение 3 — в переменной `denominator`. Если нужно сохранить много дробей, то при таком подходе это потребует больших усилий. Каждый раз для ссылки на

дробь придется ссылаться на соответствующие числитель и знаменатель. А выполнение операций с этими дробями будет очень трудным.

Намного лучше определять дробь (fraction) как один элемент и совместно ссылаться на ее числитель и знаменатель по одному имени, например, myFraction. Начнем с определения нового класса.

В программе 3.2 дублируются функции программы 3.1 с помощью нового класса с именем Fraction. Ниже приводится эта программа с подробным описанием ее работы.

Программа 3.2

```
// Program to work with fractions – class version (Программа для работы с дробями – версия с классом)

#import <Foundation/Foundation.h>

//---- @interface section ---- (секция @interface)

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//---- @implementation section ---- (секция @implementation)

@implementation Fraction
-(void) print
{
    NSLog(@"%@", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}
```

```
@end

//---- program section ---- (секция program)

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

    // Create an instance of a Fraction (Создание экземпляра Fraction)
    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    // Set fraction to 1/3 (Присваивание дроби значения 1/3)
    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Display the fraction using the print method (Вывод дроби с помощью метода print)
    NSLog(@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
    return 0;
}
```

Вывод программы 3.2

```
The value of myFraction is: (Значение myFraction)
1/3
```

Как можно видеть из комментариев, программа 3.2 логически разделена на три секции:

- @interface
- @implementation
- program

В секции @interface описывается класс, компоненты его данных и его методы, в секции @implementation содержится конкретный код, который реализует эти методы, секция program содержит программный код для выполнения цели программы.

Каждая из этих секций является частью любой программы на Objective-C, но вам не всегда придется писать каждую секцию самостоятельно. Каждая секция обычно помещается в свой собственный файл. Пока мы держим их в одном файле.

Секция @interface

Определяя новый класс, вы должны сделать несколько вещей. Во-первых, вы должны сообщить компилятору Objective-C, откуда поступил этот класс, то есть указать имя его *родительского (parent)* класса. Во-вторых, вы должны указать, какой тип данных должен сохраняться в объектах этого класса, то есть описать данные, которые будут содержать члены этого класса. Эти члены называются *переменными экземпляров (instance variable)*. И, наконец, нужно определить тип операций, или *методов (method)*, которые можно использовать при работе с объектами из этого класса. Все это делается в специальной секции программы, которая называется секцией интерфейса (@interface). В общем виде эта секция имеет следующий формат:

```
@interface Имя-Нового-Класса: Имя-Родительского-Класса
{
    объявления-членов;
}

объявления-методов;
@end
```

Принято начинать имена классов с прописной буквы, хотя это не обязательное требование. Это позволяет человеку, читающему программу, отличать имена классов от других типов переменных по первому символу имени.

Выбор имен

В главе 2 мы использовали несколько переменных для хранения целых значений. Например, в программе 2.4 переменная sum использовалась для сохранения результата сложения двух целых чисел, 50 и 25.

В языке Objective-C в переменных сохраняются не только данные, но и их типы, поскольку нужное объявление переменной выполняется до того, как она используется в программе. Переменные можно использовать для хранения чисел с плавающей точкой, символов и даже объектов (точнее, ссылки на объекты).

Правила формирования имен достаточно просты: имена должны начинаться с буквы или символа подчеркивания (_), и затем может следовать любое сочетание букв (прописных или строчных), символов подчеркивания или цифр от 0 до 9. Ниже приводятся примеры допустимых имен:

- sum
- pieceFlag

- myLocation
- numberOfRows
- _sysFlag
- ChessBoard

Следующие имена недопустимы по указанным выше причинам:

- sum\$value \$ – недопустимый символ
- piece flag – внутренние пробелы не допускаются
- 3Spencer – имена не могут начинаться с цифры
- int – это зарезервированное слово

Такие имена, как int, нельзя использовать в качестве имени переменной, поскольку они имеют специальное применение в компиляторе Objective-C. Их называют *зарезервированными именами* (*reserved name*) или *зарезервированными словами* (*reserved word*). Любое имя, которое имеет специальный смысл для компилятора Objective-C, нельзя использовать как имя переменной. В приложении В приводится полный список таких зарезервированных имен.

Не забывайте, что прописные и строчные буквы в Objective-C различаются. Например, sum, Sum и SUM – это разные переменные. Как уже говорилось, имя класса рекомендуется начинать с прописной буквы. Имена переменных экземпляров, объектов и методов обычно начинают со строчных букв. Для удобства чтения внутри имен используют прописные буквы, чтобы показать начало нового слова, как в следующих примерах:

- AddressBook – это может быть имя класса;
- currentEntry – это может быть объект;
- current_entry – некоторые программисты используют символ подчеркивания как разделитель слов;
- addNewEntry – это может быть имя метода.

Подбирайте имена, которые отражают назначение соответствующей переменной или объекта. Аналогично операторам комментария, продуманные имена могут намного повысить удобочитаемость программы, и приложенные усилия окупятся на этапах отладки и документирования. Практика показывает, что задача документирования становится намного проще, поскольку программа становится самоочевидной, не требующей пояснений.

Рассмотрим еще раз секцию @interface из программы 3.2:

```
//---- @interface section ----  
@interface Fraction: NSObject  
{  
    int numerator;
```

```
int denominator;
}
-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
@end
```

Имя нового класса – Fraction, его родительский класс – NSObject. (Подробнее о родительских классах см. главе 8.) Класс NSObject определен в файле NSObject.h, который автоматически включается в программу, когда вы импортируете Foundation.h.

Переменные экземпляров

В секции «объявления-членов» указывается, какие типы данных содержатся в классе Fraction, и имена этих типов данных. Эта секция находится внутри собственной пары фигурных скобок. Для класса Fraction эти объявления указывают, что объект Fraction содержит два целых элемента, которые называются numerator (числитель) и denominator (знаменатель):

```
int numerator;
int denominator;
```

Члены, которые объявляются в этой секции, называются переменными экземпляров. Каждый раз, когда вы создаете новый объект, создается новый уникальный набор переменных экземпляра. Поэтому, если у вас имеются два объекта Fraction, один из которых называется fracA, а второй – fracB, каждый из них содержит свой собственный набор переменных экземпляра. Таким образом, fracA и fracB будут иметь свои собственные числитель и знаменатель (numerator и denominator). Система Objective-C автоматически следит за этим, и это – одно из наиболее важных преимуществ работы с объектами.

Методы класса и методы экземпляра

Определим методы для работы с дробями. Нам нужно задавать для дроби определенное значение. Поскольку у нас нет непосредственного доступа к внутреннему представлению дроби (иначе говоря, непосредственного доступа к переменным экземпляра), нужно написать методы, которые задают числитель и знаменатель, и написать метод с именем print, который будет выводить значение дроби. Объявление метода print в файле интерфейса выглядит следующим образом:

```
-(void) print;
```

Знак «минус» (-) указывает компилятору Objective-C, что данный метод является методом экземпляра. Знак «плюс» (+) указывает на метод класса. Метод класса выполняет определенную операцию над самим классом, например, со-

здаст новый экземпляр данного класса. Это аналогично производству нового автомобиля в том смысле, что автомобиль (*car*) – это класс, и создание нового автомобиля является методом класса.

Метод экземпляра выполняет определенную операцию для определенного экземпляра класса, например, задает его значение, читает его значение, выводит на экран его значение и т.д.

Возвращаемые значения

При объявлении нового метода вы должны сообщить компилятору Objective-C, будет ли метод возвращать значение, и если да, то указать тип возвращаемого значения. Для этого нужно заключить тип возвращаемого значения в круглые скобки после ведущего символа «минус» или «плюс». Приведенное ниже объявление указывает, что метод экземпляра с именем `retrieveNumerator` возвращает целое значение.

```
- (int) retrieveNumerator;
```

В следующей строке объявляется метод, который возвращает значение с двойной точностью. (Подробнее об этом типе данных см. в главе 4.)

```
- (double) retrieveDoubleValue;
```

Значение возвращается из метода с помощью оператора Objective-C `return` аналогично значению, возвращаемому из `main` в примерах предыдущей программы. Если метод не возвращает никакого значения, нужно указать это с помощью типа `void`, как в следующем примере:

```
- (void) print;
```

Это объявление метода экземпляра `print`, этот метод не возвращает никакого значения. В подобных случаях вам не нужно выполнять оператор `return` в конце метода. Можно выполнить `return` без указания какого-либо значения:

```
return;
```

Указывать тип возвращаемого значения для методов необязательно, хотя практика программирования рекомендует это делать. Если тип не указан, то по умолчанию используется тип `id`. Подробнее о типе данных `id` см. в главе 9. По сути, тип `id` можно использовать для ссылки на любой тип объекта.

Аргументы для метода

В секции `@interface` программы 3.2 объявляется еще два метода.

```
- (void) setNumerator: (int) n;  
- (void) setDenominator: (int) d;
```

Оба метода экземпляра не возвращают никакого значения. Каждому методу передается целый аргумент (параметр), который указывается типом `(int)` перед именем аргумента. В случае `setNumerator` имя аргумента – `n`. Это имя выбирается произвольно и используется методом для ссылки на аргумент. В объявлении

`setNumerator` устанавливается, что один целый аргумент с именем `n` будет передаваться методу, а метод не будет возвращать значение. Метод `setDenominator` лейтсвует так же, но его аргумент имеет имя `d`.

Обратите внимание на синтаксис объявления методов. Имя каждого метода заканчивается двоеточием, указывая компилятору Objective-C, что для метода должен быть указан аргумент. Затем указывается тип аргумента, заключенный в круглые скобки. Так же указывается тип возвращаемого значения для самого метода. И, наконец, символическое имя указывает, что для метода задан аргумент. В конце объявления ставится точка с запятой. Этот синтаксис показан на рис. 3.1.

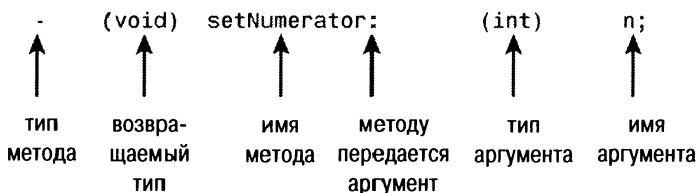


Рис. 3.1. Объявление метода

Если методу передается аргумент, то при ссылке на этот метод нужно добавить двоеточие после имени метода. В нашем случае необходимо указать `setNumerator:` и `setDenominator::`. Ссылка на метод `print` без двоеточия в конце имени указывает, что этому методу не передаются аргументы. В главе 7 рассматриваются методы, которым передается несколько аргументов.

Секция @implementation

Как говорилось выше, секция `@implementation` содержит код для методов, объявленных в секции `@interface`. В секции `@implementation` вы *объявляете* (*declare*) методы в секции `@interface` и *определяете* их (*define*), то есть пишете для них конкретный код.

В общем виде секция `@implementation` имеет следующий формат.

```
@implementation Имя-Нового-Класса
    Определения-методов;
@end
```

Имя-Нового-Класса – то же самое имя, которое использовалось для данного класса в секции `@interface`. Можно использовать двоеточие в конце имени и затем имя родительского класса, как мы делали в секции `@interface`:

```
@implementation Fraction: NSObject
```

Однако это не обязательно, и обычно не делается.

Часть *Определения-методов* секции `@implementation` содержит код для каждого метода, указанного в секции `@interface`. Аналогично секции `@interface`, определение каждого метода начинается с указания типа метода (для класса или экзем-

пляра), типа возвращаемого значения, аргументов метода и их типов. Но вместе завершения строки символом «точка с запятой» следует код для этого метода, заключенный в фигурные скобки.

Рассмотрим секцию @implementation из программы 3.2.

```
//---- @implementation section ----
@implementation Fraction
-(void) print
{
    NSLog ("%i/%i", numerator, denominator);
}
-(void) setNumerator: (int) n
{
    numerator = n;
}
-(void) setDenominator: (int) d
{
    denominator = d;
}
@end
```

В методе `print` используется `NSLog` для вывода значений переменных экземпляра `numerator` и `denominator`. `numerator` и `denominator` – это переменные экземпляра в объекте, который является получателем сообщения. Это важная концепция, и мы еще вернемся к ней.

Метод `setNumerator:` сохраняет значение целого аргумента, которому мы присвоили имя `n`, в переменной экземпляра `numerator`. Метод `setDenominator:` сохраняет значение аргумента `d` в переменной экземпляра `denominator`.

Секция program

Секция `program` содержит код для решения конкретной задачи, которая может включать много файлов. Как говорилось выше, наличие процедуры `main` является обязательным. В ней начинается выполнение программы. Ниже приводится секция `program` из программы 3.2.

```
//---- program section ----
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

```
Fraction *myFraction;  
  
// Create an instance of a Fraction (Создание экземпляра класса Fraction)  
  
myFraction = [Fraction alloc];  
myFraction = [myFraction init];  
  
// Set fraction to 1/3 (Задание значения дроби 1/3)  
  
[myFraction setNumerator: 1];  
[myFraction setDenominator: 3];  
  
// Display the fraction using the print method (Вывод дроби с помощью метода print)  
  
NSLog (@"The value of myFraction is:");  
[myFraction print];  
  
[myFraction release];  
[pool drain];  
  
return 0;  
}
```

Внутри процедуры `main` вы определяете переменную с именем `myFraction`:

```
Fraction *myFraction;
```

Здесь указывается, что `myFraction` является объектом типа `Fraction`, то есть `myFraction` используется для хранения значений из вашего нового класса `Fraction`. Звездочка (*) перед `myFraction` является обязательным элементом. Технически она обозначает, что `myFraction` является на самом деле ссылкой (или *указателем — pointer*) на `Fraction`. Мы рассмотрим ссылки позднее.

Теперь, когда у нас имеется объект для хранения дроби (`Fraction`), нужно создать саму дробь.

```
myFraction = [Fraction alloc];
```

`alloc` — это сокращение от *allocate* (*выделить*). Вам нужно выделить пространство в памяти для новой дроби. С помощью этого выражения происходит отправка сообщения новому классу `Fraction`:

```
[Fraction alloc]
```

Вы обращаетесь к классу `Fraction` для применения метода `alloc`, но вы не определяли метод `alloc`. Он был унаследован из родительского класса. Эта тема рассматривается в главе 8.

Отправив классу сообщение `alloc`, вы получаете в ответ новый экземпляр этого класса. В программе 3.2 возвращаемое значение сохраняется внутри переменной `myFraction`. Метод `alloc` очищает все переменные экземпляра объекта. Затем вам нужно инициализировать объект после выделения памяти для него.

Это выполняет оператор:

```
myFraction = [myFraction init];
```

Здесь используется метод `init`, который инициализирует экземпляр класса. Сообщение `init` отправляется в `myFraction`. Вам нужно инициализировать конкретный объект `Fraction`, поэтому вы отправляете сообщение не классу, а экземпляру этого класса.

Метод `init` возвращает также значение — инициализированный объект. Это возвращаемое значение сохраняется в переменной `myFraction`. Эта последовательность из двух строк (выделение памяти для нового экземпляра класса и последующая его инициализация) выполняется в Objective-C так часто, что два сообщения обычно объединяются:

```
myFraction = [[Fraction alloc] init];
```

Сначала выполняется оценка внутреннего выражения:

```
[Fraction alloc]
```

Результатом этого выражения является конкретный выделяемый элемент типа `Fraction`. Вместо сохранения результата выделения в переменной, как это было сделано выше, к нему непосредственно применяется метод `init`. Таким образом, здесь снова выделяется сначала новый элемент типа `Fraction`, и затем происходит его инициализация. Результат инициализации присваивается переменной `myFraction`.

Для сокращения записи выделение памяти и инициализация часто встраиваются непосредственно в строку объявления:

```
Fraction *myFraction = [[Fraction alloc] init];
```

Мы часто используем этот стиль в книге, поэтому важно, чтобы вы понимали его. Вы видели выше, как действует автоматически высвобождаемый пул (`autorelease pool`):

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

Здесь сообщение `alloc` отправляется классу `NSAutoreleasePool` для запроса о создании нового экземпляра. Затем этому новому созданному объекту отправляется сообщение `init` для его инициализации. Теперь, вернувшись к программе 3.2, вы можете задать значение дроби.

```
// Set fraction to 1/3  
[myFraction setNumerator: 1];  
[myFraction setDenominator: 3];
```

В первом операторе выполняется отправка сообщения `setNumerator:` переменной `myFraction`. Для аргумента указывается значение 1, и управление передается методу `setNumerator:`, который был определен для класса `Fraction`. Система Objective-C «понимает», что это метод из данного класса, поскольку «знает», что `myFraction` является объектом из класса `Fraction`.

Внутри метода `setNumerator:` переданное значение 1 сохраняется в переменной `n`. В единственной программной строке метода это значение сохраняется в переменной экземпляра `numerator`. Фактически вы присваиваете элементу `numerator` в `myFraction` значение 1.

Затем следует сообщение для обращения к методу `setDenominator:` в `myFraction`. Значение аргумента 3 присваивается переменной `d` внутри метода `setDenominator:`. Это значение затем сохраняется в переменной экземпляра `denominator`, что завершает присваивание значения 1/3 переменной `myFraction`. Теперь можно вывести на экран значение дроби.

```
// display the fraction using the print method  
NSLog(@"The value of myFraction is:");  
[myFraction print];
```

В результате вызова `NSLog` выводится текст:

The value of myFraction is: (Значение myFraction:)

В следующем выражении для сообщения вызывается метод `print:`

```
[myFraction print];
```

Внутри метода `print` выполняется вывод значений переменных экземпляра `numerator` и `denominator`, разделенных наклонной чертой.

Следующее сообщение в программе освобождает память, которая использовалась для этого объекта `Fraction`:

```
[myFraction release];
```

Это критически важная часть в практике программирования. При создании каждого нового объекта вы запрашиваете память, выделяемую для объекта. Закончив работу с объектом, вы обязаны освободить эту память. Память освобождается автоматически, когда происходит завершение программы, но при разработке сложных приложений вам придется работать с сотнями (или тысячами) объектов, которые используют много памяти. Если не освобождать память во время работы программы, то это может замедлить ее выполнение. Поэтому возмите за правило освобождать память, как только это можно сделать.

В системе выполнения программ (runtime) Apple обеспечивается механизм очистки памяти, который называется *сборкой мусора* (*garbage collection*), но лучше самому управлять использованием памяти, а не полагаться на этот автоматизированный механизм.

Вы не можете полагаться на сборку мусора для тех платформ, где она не поддерживается, например, iPhone. У вас может создаться впечатление, что вам пришлось написать намного больше кода, чтобы сделать в программе 3.2 то же самое, что и в программе 3.1. Для данного простого примера это верно, но такой способ делает упрощает написание, поддержку и расширение больших и сложных программ. Вы поймете это позже.

В последнем примере этой главы показано, как работать с несколькими дробями. В программе 3.3 одной дроби присваивается значение $2/3$, второй дроби – $3/7$, и затем выполняется вывод обеих дробей.

Программа 3.3

```
// Program to work with fractions – cont'd (Программа для работы с дробями – продолжение)

#import <Foundation/Foundation.h>

//---- @interface section ----

@interface Fraction: NSObject

{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
@end

//---- @implementation section ----

@implementation Fraction
-(void) print
{
    NSLog(@"%@", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}
```

```
@end

//---- program section ----

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

    Fraction *frac1 = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // Set 1st fraction to 2/3 (Задание 1-й дроби 2/3)

    [frac1 setNumerator: 2];
    [frac1 setDenominator: 3];

    // Set 2nd fraction to 3/7 (Задание 2-й дроби 3/7)

    [frac2 setNumerator: 3];
    [frac2 setDenominator: 7];

    // Display the fractions (Вывод дробей)

    NSLog (@"First fraction is:");

    [frac1 print];

    NSLog (@"Second fraction is:");
    [frac2 print];

    [frac1 release];
    [frac2 release];

    [pool drain];
    return 0;
}
```

Вывод программы 3.3

```
First fraction is: (Первая дробь:)
2/3
Second fraction is: (Вторая дробь:)
3/7
```

Секции `@interface` и `@implementation` остались такими же, как в программе 3.2. В программе создаются *два* объекта типа `Fraction`, — `frac1` и `frac2`, затем им присваиваются соответственно значения $2/3$ и $3/7$. Когда метод `setNumerator:` применяется к `frac1`, чтобы задать значение 2 для его числителя (`numerator`), выполняется присваивание значения 2 переменной экземпляра `numerator`. Аналогичным образом, когда для `frac2` применяется тот же метод, чтобы задать значение 3 для его числителя, выполняется присваивание значения 3 его отдельной переменной экземпляра `numerator`. Новый объект получает свой собственный отдельный набор переменных экземпляра при каждом создании (рис. 3.2).

В зависимости от объекта, которому отправляется сообщение, происходит ссылка на соответствующие переменные экземпляра. В следующем примере выполняется ссылка на numerator объекта frac1, если внутри метода setNumerator: используется имя numerator:

```
[frac1 setNumerator: 2];
```

Это происходит потому, что получателем сообщения является frac1.

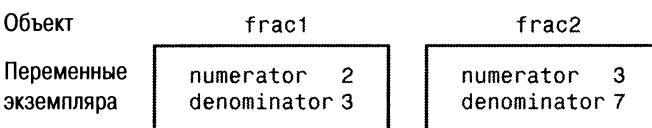


Рис. 3.2. Уникальные переменные экземпляра

Доступ к переменным экземпляра и инкапсуляция данных

Мы увидели, каким образом методы, используемые для работы с дробями, выполняют доступ по имени к двум переменным экземпляра: numerator и denominator. Метод экземпляра всегда может выполнять непосредственный доступ к переменным экземпляра. Однако это не может делать метод класса, поскольку он применяется только к самому классу, а не к экземплярам этого класса. Но что делать, если нужно выполнять доступ к переменным экземпляра из какого-либо другого места, например, изнутри процедуры main? Это нельзя сделать напрямую, поскольку переменные экземпляра скрыты. Это еще одна ключевая концепция, которая называется *инкапсуляцией данных* (*data encapsulation*). Это позволяет расширять и изменять определения класса, не заботясь о том, что пользователи данного класса будут работать с внутренними деталями класса. Инкапсуляция данных обеспечивает необходимый уровень изоляции между программистом и разработчиком класса.

Вы можете выполнять доступ к своим переменным экземпляра, написав специальные методы для считывания их значений. Например, можно создать два новых метода (numerator и denominator) для доступа к соответствующим переменным экземпляра Fraction, который является получателем сообщения. Результатом будет возвращаемое целое значение. Ниже приводятся объявления для двух новых методов:

```
- (int) numerator;  
- (int) denominator;
```

Их определения:

```
- (int) numerator  
{  
    return numerator;  
}
```

```
-(int) denominator
{
    return denominator;
}
```

Имена методов и переменных экземпляра, к которым они выполняют доступ, совпадают. Это распространенная практика. В программе 3.4 выполняется проверка этих методов.

Программа 3.4

```
// Program to access instance variables – cont'd (Программа доступа к переменным
экземпляра – продолжение)

#import <Foundation/Foundation.h>
//---- @interface section ----

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;

@end

//---- @implementation section ----

@implementation Fraction
-(void) print
{
    NSLog (@»%i/%i», numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
```

```

{
    return numerator;
}

-(int) denominator
{
    return denominator;
}

@end

//---- program section ----

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

    // Set fraction to 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Display the fraction using our two new methods

    NSLog (@"The value of myFraction is: %i/%i",
           [myFraction numerator], [myFraction denominator]);
    [myFraction release];
    [pool drain];

    return 0;
}

```

Вывод программы 3.4

The value of myFraction is 1/3 (Значение myFraction 1/3)

Оператор `NSLog` выводит результаты отправки двух сообщений экземпляру `myFraction`: первого – для считывания значения его переменной `numerator`, второго – для считывания значения `denominator`.

```
NSLog (@"The value of myFraction is: %i/%i",
       [myFraction numerator], [myFraction denominator]);
```

Методы, которые задают значения переменных экземпляра, называют *установщиками (setters)*, а методы, которые используются для считывания значений переменных экземпляра, называются *получателями (getters)*. Для класса `Fraction` методы `setNumerator:` и `setDenominator:` являются установщиками, а `numerator` и `denominator` являются получателями.

Примечание. Вскоре вы познакомитесь с удобным средством Objective-C 2.0, которое позволяет автоматически создавать методы-установщики и методы-получатели.

Мы должны указать, что метод с именем `new` объединяет действия методов `alloc` и `init`. Поэтому следующую строку можно использовать для выделения памяти и инициализации нового объекта `Fraction`:

```
Fraction *myFraction = [Fraction new];
```

Но обычно применяется двухшаговый способ выделения памяти и инициализации, поскольку в действительности происходят два отдельных события: сначала создается новый объект, а затем выполняется его инициализация.

Упражнения

1. Какие из следующих имен не являются допустимыми? Почему?

Int	playNextSong	6_05
_calloc	Xx	alphaBetaRoutine
clearScreen_1312	z	
Reinitialize	-	A\$

2. По аналогии с автомобилем, рассмотрите объект, который вы используете каждый день. Определите класс для этого объекта и напишите пять действий для этого объекта.
3. Для списка из упражнения 2 используйте следующий синтаксис, чтобы переписать ваш список в указанном формате:
[экземпляр метод];
4. Предположим, что у вас, помимо автомобиля, есть лодка и мотоцикл. Напишите список действий, которые вы можете выполнять с каждым из них. Есть ли какое-то перекрытие между этими действиями?
5. В соответствии с вопросом 4 предположим, что у вас есть класс с именем `Vehicle` (Средство передвижения) и объект с именем `myVehicle`, которым может быть `Car` (Автомобиль), `Motorcycle` (Мотоцикл) или `Boat` (Лодка). Предположим, вы написали следующее:

```
[myVehicle prep];
[myVehicle getGas];
[myVehicle service];
```

Видите ли вы преимущества в применении действия к объекту, который относится к одному из нескольких классов?

6. В процедурном языке, таком как C, вы продумываете действия и затем пишете код для применения этого действия к различным объектам. Если взять пример автомобиля, то вы могли бы написать на C процедуру мытья средства передвижения и затем внутри этой процедуры написать код для мытья автомобиля, мытья лодки, мытья мотоцикла и т.д. Если принять этот процедурный подход и затем добавить новый тип средства передвижения (см. предыдущее упражнение), будет ли он иметь преимущества или недостатки по сравнению с объектно-ориентированным подходом?
7. Определите класс с именем `XYPoint` для декартовых координат (x, y), где x и y – целые значения. Определите методы, позволяющие по отдельности задавать координаты точки и считывать их значения. Напишите программу на Objective-C, чтобы реализовать этот новый класс.

Глава 4

Типы данных и выражения

В этой главе мы рассмотрим базовые типы данных и основные правила формирования арифметических выражений в Objective-C.

Типы данных и константы

Вы уже видели, как в Objective-C используется один из основных типов данных – `int`. Переменную, объявленную с типом `int`, можно использовать только для хранения целых значений.

В языке программирования Objective-C имеются три основных типа данных: `float`, `double` и `char`. Переменную, которая объявлена с типом `float`, можно использовать для хранения чисел с плавающей точкой (то есть значений, имеющих дробную часть). Тип `double` тоже хранит числа с плавающей точкой, но с двойной точностью. Тип данных `char` можно использовать для хранения одного символа, например, буквы «`a`», цифры «`6`», знака «`;`» (подробнее об этом см. ниже).

В Objective-C любое число, символ или строка символов называется *константой* (*constant*). Например, число `58` является целым значением-константой. Стока `@"Programming in Objective-C is fun.\n"` – это пример объекта-константы со строкой символов. Выражение, содержащее только значения-константы, называется *выражением-константой*, или *константным выражением* (*constant expression*). Следующее выражение является константным, поскольку каждый из членов этого выражения является константой:

`128 + 7 - 17`

Но если `i` объявлена как переменная целого типа, то это выражение уже не будет константным:

`128 + 7 - i`

Тип `int`

В Objective-C целая константа состоит из одной или нескольких цифр. Знак «минус» перед цифрами указывает, что это отрицательное значение. `158`, `-10` и `0` – примеры целых констант. Между цифрами не допускаются пробелы, и значения между тройками цифр нельзя разделять запятыми. (То есть число `12,000` должно быть записано как `12000`.)

В Objective-C имеется два специальных формата записи целых констант с основанием, отличным от 10. Если первая цифра целого значения равна 0, это значение считается *восьмеричным*, *octal* (по основанию 8). В этом случае остальные цифры должны быть допустимыми восьмеричными цифрами от 0 до 7. Например, чтобы представить в Objective-C восьмеричное значение 50 (эквивалентное десятичному значению 40), используется форма записи 050. Восьмеричная константа 0177 эквивалентна десятичному значению 127 ($1 \times 64 + 7 \times 8 + 7$). Целое значение можно вывести на экран в восьмеричной записи, вызвав NSLog и указав символы формата %o в строке форматирования. Значение выводится в восьмеричной записи без ведущего нуля. Символы формата %#o выводят ведущий нуль перед восьмеричным значением.

Если перед целой константой ставится 0 и буква x (прописная или строчная), значение считается шестнадцатеричным (по основанию 16). После буквы x следуют цифры шестнадцатеричного значения (это цифры 0-9 и буквы a-f (или A-F)). Буквы представляют, соответственно, десятичные числа от 10 до 15. Например, чтобы присвоить шестнадцатеричное значение FFEF0D целой переменной с именем rgBColor, можно использовать оператор:

```
rgBColor = 0xFFEF0D;
```

Символы форматирования %x выводят значение в шестнадцатеричном формате без ведущих символов 0x и с использованием строчных букв a-f. Чтобы вывести значение с ведущими символами 0x, нужно использовать символы форматирования %#x:

```
NSLog ("Color is %#x\n", rgBColor);
```

Для вывода ведущего символа x и шестнадцатеричных цифр прописными буквами в символах форматирования ставится прописная буква X: %X или %#X.

Каждое значение, будь то символ, целое число или число с плавающей точкой, имеет связанный с ним диапазон значений. Этот диапазон связан с количеством памяти, выделяемой для хранения определенного типа данных. Обычно он зависит от компьютера, на котором выполняется работа, и поэтому называется *реализацией* или *машинно-зависимым диапазоном* (*implementation* или *machine dependent*). Так, целое значение может занимать на вашем компьютере 32 или 64 бита.

Никогда не пишите программы, в которых предполагается определенный размер типов данных. Для каждого базового типа данных гарантируются определенные размеры памяти, например, для целого значения отводится не менее 32 бит памяти. Однако на некоторых машинах это не выполняется (см. таблицу B.3 в приложении B).

Тип float

Для хранения значений, содержащих знаки после запятой (точки), можно использовать переменные с типом *float*. Признаком константы с плавающей точкой является присутствие десятичной точки. Можно не указывать цифры до

десятичной точки или после нее, но, очевидно, нельзя не указывать одновременно обе составляющие. 3., 125.8 и -0.001 являются примерами констант с плавающей точкой. В NSLog для вывода значения с плавающей точкой используются символы формата %f.

Константы с плавающей точкой можно записывать в виде так называемого экспоненциального представления (*scientific notation*). 1.7e4 – это значение с плавающей точкой для 1.7×10^4 . Значение до буквы e называется мантиссой (*mantissa*), а значение после e называется порядком (*exponent*). Порядок, который может быть дополнительно снабжен знаком «плюс» или «минус», представляет степень числа 10 для умножения на мантиссу. Например, в константе 2.25e-3 значение 2.25 – это мантисса, а -3 – порядок. Эта константа представляет значение 2.25×10^{-3} , или 0.00225. Букву e, которая отделяет мантиссу от порядка, можно записывать и как прописную, и как строчную букву.

Чтобы вывести значение в экспоненциальном представлении, в строке формата NSLog должны быть заданы символы форматирования %e. Если использовать символы формата %g, то процедура NSLog сама определит, как выводить значение с плавающей точкой: в обычном виде записи с плавающей точкой или в экспоненциальном представлении. Решение основывается на значении порядка: если он меньше -4 или больше 5, то используется формат %e (экспоненциальное представление), если нет – используется формат %f.

Шестнадцатеричная константа с плавающей точкой содержит ведущие символы 0x или 0X, после которых следует одна или несколько шестнадцатеричных цифр, затем p или P, после чего следует двоичный порядок. Например, 0x0.3p10 представляет значение $3/16 \times 2^{10} = 192$.

Тип double

Тип double аналогичен типу float. Он используется, если диапазон, предоставляемый переменной типа float, не дает достаточной точности. В переменных, которые объявлены с типом double, можно сохранять примерно вдвое больше цифр, чем с типом float. На большинстве компьютеров для значений типа double используются 64 бита.

Если не указано особо, то компилятор Objective-C обрабатывает все константы с плавающей точкой как значения типа double. Чтобы явно задать константу типа float, нужно добавить f или F в конце числа, например,

12.5f.

Чтобы вывести значение типа @double, можно использовать символы формата %f, %e или %g, как и для значений типа float.

Тип char

Переменную типа char можно использовать для хранения одного символа. Чтобы задать символ-константу, нужно заключить символ в апострофы, например, 'a', ';' и '0'. Не следует путать символ-константу с символьной строкой в стиле C, которая содержит любое число символов, заключенных в кавычки. Стока

символов, заключенная в кавычки, перед которой поставлен символ @, – это объект-строка типа `NSString`.

Примечание. В приложении В описываются методы сохранения символов из расширенных наборов с помощью специальных escape-последовательностей, уникальных символов и так называемых «широких» символов.

Символ-константа '\n' (символ новой строки) является допустимым символом-константой. Обратный слэш (обратная наклонная черта) – это специальный символ в системе Objective-C, который не учитывается как отдельный символ. Компилятор Objective-C интерпретирует '\n' как один символ. Имеются и другие специальные символы, которые инициируются с помощью обратного слэша. Полный список приводится в приложении В. Для вывода значения переменной типа `char` можно использовать символы формата %c при вызове `NSLog`.

В программе 4.1 используются базовые типы данных Objective-C.

Программа 4.1

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

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

    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';

    NSLog (@"integerVar = %i", integerVar);
    NSLog (@"floatingVar = %f", floatingVar);
    NSLog (@"doubleVar = %e", doubleVar);
    NSLog (@"doubleVar = %g", doubleVar);
    NSLog (@"charVar = %c", charVar);

    [pool drain];
    return 0;
}
```

Вывод программы 4.1

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = 'W'
```

Во второй строке вывода значение 331.79, присвоенное переменной `floatingVar`, выводится как 331.790009. Причиной этой неточности является внутреннее представление чисел в компьютере. С таким же типом неточности вы сталкиваетесь

при работе с числами на своем калькуляторе. Если разделить 1 на 3 на калькуляторе, в результате получится .33333333 (в конце, возможно, будет еще несколько троек). Эта строка является аппроксимацией одной трети калькулятором. Теоретически число троек бесконечно, но калькулятор может содержать столько цифр, сколько соответствует внутренней точности устройства. Тот же тип неточности возникает и здесь: определенные значения с плавающей точкой не могут быть точно представлены в памяти компьютера.

Квалификаторы: `long`, `long long`, `short`, `unsigned` и `signed`

Если квалификатор `long` помещен непосредственно перед объявлением `int`, то соответствующая целая переменная на некоторых компьютерах имеет расширенный диапазон. Пример объявления `long int`:

```
long int factorial;
```

Это объявление целой переменной `factorial` типа `long`. Как и в случае переменных типа `float` и `double`, конкретная точность переменной типа `long` зависит от компьютерной системы. Во многих системах `int` и `long int` имеют одинаковый диапазон и могут использоваться для хранения целых значений до 32 бит ($2^{31} - 1$, или 2147483647).

Значение константы типа `long int` формируется путем добавления буквы `L` (прописной или строчной) в конец целой константы. Между числом и буквой `L` не допускаются никакие пробелы. Ниже приводится пример объявления переменной `numberOfPoints` типа `long int` с начальным значением 131071100:

```
long int numberOfPoints = 131071100L;
```

Чтобы вывести значение типа `long int` с помощью `NSLog`, перед символами форматирования `i`, `o` и `x` в качестве модификатора ставится буква `l`. Это означает, что символы форматирования `%li` выводят значения типа `long int` в десятичном формате, символы `%lo` — в восьмеричном формате, символы `%lx` — в шестнадцатеричном формате.

Пример применения типа данных `long long`:

```
long long int maxAllowedStorage;
```

Указанная переменная будет иметь заданную увеличенную точность — не менее 64 бит. Для вывода целых значений типа `long long` вместо одной буквы `l` в строке `NSLog` используются две буквы `ll`, как в `"%lli"`.

Квалификатор `long` разрешается также использовать перед типом `double`, например:

```
long double US_deficit_2004;
```

Константа типа `long double` записывается в виде константы с плавающей точкой с добавлением в конце `l` или `L`, например:

```
1.234e+7L
```

Чтобы вывести значение типа `long double`, нужно указать модификатор `L`. `%Lf` выводит значение типа `long double` с десятичной точкой, `%Le` выводит значение в экспоненциальном представлении, при `%Lg` процедура `NSLog` выбирает между `%Lf` и `%Le`.

Квалификатор `short`, помещенный перед объявлением `int`, указывает компилятору Objective-C, что определенная переменная используется для хранения относительно небольших целых значений. Переменные типа `short` экономят память. Это существенно, если программа использует большой объем памяти и количество доступной памяти ограничено.

На некоторых машинах переменная `short int` занимает половину памяти, которая требуется переменной типа `int`. В любом случае для переменных типа `short int` выделяется не меньше 16 бит.

В Objective-C не существует способа записать константу типа `short int`. Чтобы вывести переменную типа `short int`, нужно поместить букву `h` перед любым из символов преобразования в целое значение: `%hi`, `%ho` или `%hx`. Для отображения значений `short int` можно также использовать любой из символов преобразования в целое значение, поскольку эти символы можно преобразовать в целые значения, когда они передаются в процедуру `NSLog` как аргументы.

Квалификатор, который можно помещать перед переменной типа `int`, применяется для переменной, которая должна содержать только положительные числа. Это квалификатор `unsigned`.

```
unsigned int counter;
```

Этот квалификатор расширяет диапазон значений целой переменной.

Константа типа `unsigned int` формируется путем добавления `u` или `U` после константы.

```
0x00ffU
```

Для целой константы можно сочетать `u` (или `U`) и `l` (или `L`). Например, в следующем случае компилятор будет интерпретировать константу `20000` как `unsigned long`.

```
20000UL
```

Целая константа, которая не заканчивается буквами `u`, `U`, `l` или `L` и слишком велика, чтобы уместиться в обычный размер типа `int`, интерпретируется компилятором как `unsigned int`. Если она слишком велика, чтобы уместиться в `unsigned int`, то компилятор обрабатывает ее как `long int`. Если она не умещается даже в `long int`, компилятор обрабатывает ее как `unsigned long int`.

При объявлении переменных с типом `long int`, `short int` или `unsigned int` вы можете опустить ключевое слово `int`. Переменную `counter` типа `unsigned int` можно объявить следующим образом:

```
unsigned counter;
```

Переменные типа `char` также можно объявлять как `unsigned`.

Квалификатор `signed` позволяет явно указать компилятору, что определенная переменная содержит значения со знаком. В основном его используют пе-

ред объявлением `char`. Подробное описание этого квалификатора выходит за рамки этой книги.

Тип `id`

Тип данных `id` используется для хранения объекта любого типа. Его можно называть обобщенным типом объекта. Ниже объявляется переменная `number` с типом `id`.

```
id number;
```

Можно объявлять методы, которые возвращают значения типа `id`.

```
-(id) newObject: (int) type;
```

Здесь объявляется метод экземпляра с именем `newObject`, который принимает один целый аргумент с именем `type` и возвращает значение типа `id`. `id` – это тип по умолчанию для типа возвращаемого значения и типа аргумента. Ниже приводится объявление метода класса, который возвращает значение типа `id`.

```
+allocInit;
```

Тип данных `id` часто используется в этой книге. Тип `id` является основой таких важных средств в Objective-C, как *полиморфизм (polymorphism)* и *динамическое связывание (dynamic binding)*. (Подробнее см. главу 9.)

В таблице 4.1 приводится сводка базовых типов и квалификаторов.

Табл. 4.1. Базовые типы данных

Тип	Примеры констант	Символы формата <code>NSLog</code>
<code>char</code>	'a', '\n'	%c
<code>short int</code>	–	%hi, %hx, %ho
<code>unsigned short int</code>	–	%hu, %hx, %ho
<code>int</code>	12, -97, 0xFFE0, 0177	%i, %x, %o
<code>unsigned int</code>	12u, 100U, 0xFFFFu	%u, %x, %o
<code>long int</code>	12L, -2001, 0xfffffL	%li, %lx, %lo
<code>unsigned long int</code>	12UL, 100ul, 0xffffeUL	%lu, %lx, %lo
<code>long long int</code>	0xe5e5e5e5LL, 500ll	%lli, %llx, &lllo
<code>unsigned long long int</code>	12ull, 0xffeeULL	%llu, %llx, %lllo
<code>float</code>	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
<code>double</code>	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
<code>long double</code>	12.341, 3.1e-5l	%Lf, \$Le, %Lg
<code>id</code>	<code>nil</code>	%p

Арифметические выражения

В Objective-C, как практически во всех языках программирования, знак «плюс» (+) используется для сложения двух значений, знак «минус» – для вычитания двух значений, звездочка (*) – для умножения двух значений, а слэш (/) – для деления двух значений. Эти операторы называются *бинарными* (*binary*) арифметическими операторами, поскольку они работают с двумя значениями, или членами.

Старшинство операторов

Вы уже видели, как в Objective-C выполняется простая операция, например, сложение. В следующей программе показаны операции вычитания, умножения и деления. В последних двух операторах этой программы показано, что оператор может иметь более высокий *приоритет*, или *старшинство* (*precedence*) по сравнению с другим оператором. Любой оператор в Objective-C имеет свой приоритет.

Приоритет определяет порядок вычисления выражения, содержащего более одного оператора: оператор с более высоким приоритетом вычисляется первым. Выражения, содержащие операторы с одинаковым приоритетом, вычисляются слева направо или справа налево (в зависимости от оператора). Это свойство *ассоциативности* (*associative*) оператора. В приложении B приводится полный список старшинства операторов и правил ассоциативности.

Программа 4.2

```
// Использование арифметических операторов
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

    result = a - b; //вычитание
    NSLog (@"a - b = %i", result);

    result = b * c; //умножение
    NSLog (@"b * c = %i", result);

    result = a / c; //деление
    NSLog (@"a / c = %i", result);
```

```
result = a + b * c; //старшинство
NSLog(@"a + b * c = %i", result);

NSLog(@"a * b + c * d = %i", a * b + c * d);

[pool drain];
return 0;
}
```

Вывод программы 4.2

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

После объявления целых переменных `a`, `b`, `c`, `d` и `result` программа присваивает результат вычитания `b` из `a` переменной `result` и затем выводит ее значение с помощью вызова `NSLog`.

В следующей строке значение `b` умножается на значение `c`, и произведение сохраняется в переменной `result`:

```
result = b * c;
```

Результат умножения выводится с помощью вызова `NSLog`. В следующей строке программы используется оператор деления и с помощью `NSLog` выводится результат деления `a` на `c`, то есть 100 на 25.

Попытка деления на ноль вызывает аварийное завершение. Даже если программа не будет прекращена, результат такого деления не имеет смысла. В главе 6 вы научитесь проверять, не равен ли делитель нулю, прежде чем выполнить операцию деления. Если делитель равен нулю, можно обойти операцию деления.

Результат следующего выражения не равен 2550 (102×25); вместо этого `NSLog` выводит значение 150.

```
a + b * c
```

Objective-C, как и большинство других языков программирования, выполняет несколько операций над членами выражения по определенному порядку. Вычисление выражения обычно происходит слева направо, но операции умножения и деления имеют приоритет над операциями сложения и вычитания. Поэтому система вычисляет выражение

```
a + b * c
```

следующим образом:

$$a + (b * c)$$

Для изменения порядка вычисления членов внутри выражения можно использовать круглые скобки. Показанное выше выражение является допустимым выражением Objective-C. Следующую строку можно подставить в программу 4.2, чтобы получить те же результаты:

$$\text{result} = a + (b * c);$$

Но переменной `result` будет присвоено значение 2550, если использовать следующее выражение:

$$\text{result} = (a + b) * c;$$

Значение `a` (100) будет сложено со значением `b` (2) до умножения на `c` (25). Круглые скобки можно вкладывать друг в друга; выражение будет вычисляться, начиная с внутренних круглых скобок. Убедитесь, что число закрывающих скобок равно числу открывающих. В последней выполняемой строке программы 4.2 выражение передается как аргумент процедуре `NSLog` без присваивания этого выражения какой-либо переменной. Выражение

$$a * b + c * d$$

вычисляется по описанным выше правилам как

$$(a * b) + (c * d)$$

то есть

$$(100 * 2) + (25 * 4)$$

Результат 300 передается процедуре `NSLog`.

Целочисленная арифметика и унарный оператор «минус»

В программе 4.3 демонстрируется то, что мы только что обсуждали, и вводится понятие целочисленной арифметики.

Программа 4.3

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

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
int a = 25;  
int b = 2;  
int result;  
float c = 25.0;  
float d = 2.0;  
  
NSLog(@"%@", 6 + a / 5 * b = %i", 6 + a / 5 * b);  
NSLog(@"%@", a / b * b = %i", a / b * b);  
NSLog(@"%@", c / d * d = %f", c / d * d);  
NSLog(@"%@", -a = %i", -a);  
  
[pool drain];  
return 0;  
}
```

Вывод программы 4.3

```
6 + a / 5 * b = 16  
a / b * b = 24  
c / d * d = 25.000000  
-a = -25
```

Мы вставили дополнительные пробелы между `int` и объявлением переменных `a`, `b` и `result` в первых трех выполняемых строках, чтобы выровнять объявление каждой переменной. Это делает программу более удобной для чтения. Вы могли заметить, что каждый арифметический оператор окружен пробелами. Обычно вы можете добавлять дополнительные пробелы там, где разрешен один пробел. Это не обязательно, но упрощает чтение программы.

Вычисление выражения в первом вызове `NSLog` программы 4.3 происходит следующим образом.

1. Поскольку деление имеет более высокий приоритет, чем сложение, сначала значение `a` (25) делится на 5. Это дает промежуточный результат 5.
2. Поскольку умножение имеет более высокий приоритет, чем сложение, промежуточный результат 5 умножается на 2 (значение `b`), что дает новый промежуточный результат 10.
3. Наконец, выполняется сложение 6 и 10, что дает конечный результат 16.

Во второй строке, казалось бы, деление на `b` и последующее умножение на `b` должно дать значение переменной `a`, то есть присвоенной ей значение 25. Но результаты вывода дают значение 24. Дело в том, что выражение было вычислено с помощью целочисленной арифметики.

Переменные *a* и *b* были объявлены с типом *int*. Если вычисляемый член в выражении содержит два целых значения, Objective-C выполняет операцию, используя целочисленную арифметику, при которой дробная часть отбрасывается. Поэтому при делении значения *a* на значение *b* (25 делится на 2) мы получаем промежуточный результат 12, а не 12.5, как можно было бы ожидать. Умножение этого промежуточного результата на 2 дает конечный результат 24.

В предпоследней строке показано, что выполнение той же операции для значений с плавающей точкой дает ожидаемый результат.

Решение о выборе между типом переменной *float* или *int* нужно принимать, исходя из предполагаемого использования переменной. Если вам не нужна дробная часть, используйте целую переменную. Результирующая программа будет работать быстрее на многих компьютерах. С другой стороны, если вам нужна точность со знаками после десятичной точки, то выбор очевиден. Остается только решить, какой тип использовать: *float* или *double*. Ответ на этот вопрос зависит от точности чисел, с которыми вы работаете, и от их величины.

В последнем операторе *NSLog* значение переменной берется со знаком «минус» с помощью унарного (одноместного) оператора. *Унарный (unary)* оператор применяется к одному значению – в отличие от бинарного оператора, который применяется к двум значениям. Знак «минус» как бинарный оператор применяется для вычитания одного значения из другого, а как унарный оператор – для изменения знака значения.

Унарные операторы «минус» и «плюс» имеют более высокий приоритет, чем все остальные арифметические операторы. Таким образом, следующее выражение дает результат умножения *-a* на *b*:

c = *-a* * *b*;

В таблице из приложения В приводится сводка операторов с их приоритетами.

Оператор остатка от деления

Последний арифметический оператор, представленный в этой главе – это оператор остатка от деления, который указывается знаком процента (%). Чтобы определить, как работает этот оператор, рассмотрим программу 4.4.

Программа 4.4

```
// Оператор остатка от деления
#import <Foundation/Foundation.h>
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int a = 25, b = 5, c = 10, d = 7;
```

```
NSLog(@"a %% b = %i", a % b);
NSLog(@"a %% c = %i", a % c);
NSLog(@"a %% d = %i", a % d);
NSLog(@"a / d * d + a %% d = %i", a / d * d + a % d);

[pool drain];
return 0;
}
```

Вывод программы 4.4

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

В процедуре `main` переменные `a`, `b`, `c` и `d` определяются в одной строке.

Символ после знака «`%%`» указывает, как должен выводиться следующий аргумент. Если следующим символом является еще один знак процента, процедура `NSLog` выводит этот знак и вставляет его в соответствующую позицию вывода программы.

Из результатов вывода можно заключить, что оператор `%` вычисляет остаток от деления первого значения на второе. В первом примере остаток от деления 25 на 5 равен 0. Если разделить 25 на 10, то мы получим в остатке 5, что подтверждается второй строкой вывода. Деление 25 на 7 дает в остатке 4, что показано в третьей строке вывода.

Теперь рассмотрим арифметическое выражение, вычисляемое в последней строке. Операции между двумя целыми значениями выполняются с помощью целочисленной арифметики, поэтому остаток от деления двух целых значений просто отбрасывается. Деление 25 на 7 (выражение `a / d`) дает промежуточный результат 3. Умножение этого значения на значение `d` (оно равно 7) дает промежуточный результат 21. И, наконец, остаток от деления `a` на `d` (в выражении `a % d`) дает конечный результат 25 ($21 + 4$). Результат равен значению переменной `a`, и это не является случайным совпадением. Следующее выражение всегда равно значению `a`, если `a` и `b` являются целыми значениями.

`a / b * b + a % b`

На самом деле оператор остатка от деления (%) предназначен для работы только с целыми значениями.

Что касается приоритета операций, то оператор остатка от деления имеет одинаковый приоритет с операторами умножения и деления. Отсюда следует, что выражение

`table + value % TABLE_SIZE`

будет вычисляться как

`table + (value % TABLE_SIZE)`

Преобразования между целыми значениями и значениями с плавающей точкой

Чтобы эффективно разрабатывать программы на Objective-C, необходимо знать правила, применяемые в Objective-C для неявного преобразования между целыми значениями и значениями с плавающей точкой. В программе 4.5 показаны примеры некоторых простых преобразований между числовыми типами данных.

Программа 4.5

```
// Основные преобразования в Objective-C
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    float f1 = 123.125, f2;
    int i1, i2 = -150;

    i1 = f1; // преобразование типа float в целый тип int
    NSLog (@"%f assigned to an int produces %i", f1, i1);

    f1 = i2; // преобразование целого типа в тип float
    NSLog (@"%i assigned to a float produces %f", i2, f1);

    f1 = i2 / 100; // деление целого значения на целое
    NSLog (@"%i divided by 100 produces %f", i2, f1);

    f2 = i2 / 100.0; // деление целого значения на тип float
    NSLog (@"%i divided by 100.0 produces %f", i2, f2);

    f2 = (float) i2 / 100; // оператор приведения типа
    NSLog (@"(float) %i divided by 100 produces %f", i2, f2);

    [pool drain];
    return 0;
}
```

Вывод программы 4.5

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

Если значение с плавающей точкой присваивается переменной целого типа, дробная часть этого числа отбрасывается. Тем самым, если в приведенной про-

грамме значение `f1` присваивается `i1`, то дробная часть числа 123.125 отбрасывается, то есть в `i1` сохраняется его целая часть (123). Это показано в первой строке вывода программы.

Если значение целой переменной присваивается переменной с плавающей точкой, это не вызывает какого-либо изменения в значении числа; система просто преобразует это значение и сохраняет его в переменной с плавающей точкой. Во второй строке вывода программы показано, что значение `i2` (-150) правильно преобразовано и сохранено в переменной `f1` типа `float`.

В следующих двух строках вывода программы показаны две ситуации, о которых нужно помнить при формировании арифметических выражений. В первом случае действует целочисленная арифметика, о которой мы говорили в предыдущем разделе. Если два операнда выражения являются целыми (это относится и к целым переменным с модераторами `short`, `unsigned` и `long`), то операция выполняется по правилам целочисленной арифметики. Поэтому любая дробная часть, полученная в операции деления, отбрасывается, даже если результат присваивается переменной с плавающей точкой (как в этой программе). Если целая переменная `i2` делится на целую константу 100, то система выполняет целочисленное деление. Поэтому результат деления -150 на 100, равный -1, сохраняется в переменной типа `float` `f1`.

Следующее деление применяется к целой переменной и константе с плавающей точкой. Любая операция, выполняемая с двумя значениями в Objective-C, выполняется как операция с плавающей точкой, если хотя бы одно из значений является переменной или константой с плавающей точкой. Поэтому при делении значения `i2` на 100.0 система выполняет деление с плавающей точкой и дает результат -1.5, который присваивается переменной типа `f1 float`.

Оператор приведения типа

При объявлении и определении методов для объявления типов возвращаемого значения и аргументов тип включается в круглые скобки. Внутри выражений этот способ применяется для другой цели.

В последней операции деления программы 4.5 появляется оператор приведения типа:

```
f2 = (float) i2 / 100; // оператор приведения типа
```

В данном случае для вычисления выражения оператор приведения типа используется для преобразования значения переменной `i2` в тип `float`. Это оператор не влияет на значение переменной `i2`; это унарный оператор. Выражение `(float) a` не влияет на значение `a` – аналогично выражению `-a`.

Оператор приведения типа имеет более высокий приоритет, чем все арифметические операторы, за исключением унарных операций «плюс» и «минус». И, конечно, вы можете использовать в выражении круглые скобки, чтобы вычисления выполнялись в нужном порядке.

В этом примере выражение

```
(int) 29.55 + (int) 21.99
```

вычисляется в Objective-C как

$29 + 21$

поскольку приведение значения с плавающей точкой к целому типу приводит к отбрасыванию дробной части. Выражение

$(\text{float}) 6 / (\text{float}) 4$

дает значение 1.5, как и в следующем выражении:

$(\text{float}) 6 / 4$

Оператор приведения типа часто используется для принудительного приведения объекта, имеющего обобщенный тип `id`, к объекту определенного класса. Например, в следующих строках выполняется преобразование значения переменной `myNumber` типа `id` к объекту класса `Fraction`:

```
id myNumber;
Fraction *myFraction;
...
myFraction = (Fraction *) myNumber;
```

Результат этого преобразования присваивается переменной `myFraction` типа `Fraction`.

Операторы присваивания

Язык Objective-C позволяет объединять арифметические операторы с оператором присваивания в обобщенном формате

`op=`

В этом формате `op` – любой из арифметических операторов, включая `+`, `-`, `*`, `/` или `%`. Кроме того, `op` может быть любым из битовых операторов для смещения и маскирования, которые описываются ниже.

Рассмотрим строку

`count += 10;`

Оператор «плюс равно» `+=` добавляет выражение, находящееся справа от этого оператора, к переменной, находящейся слева от оператора, и сохраняет результат в этой переменной. Таким образом, приведенная выше строка эквивалентна следующей строке:

`count = count + 10;`

В следующем выражении используется оператор «минус равно» для вычитания 5 из значения переменной `counter`:

`counter -= 5`

Это эквивалентно выражению

```
counter = counter - 5
```

А теперь чуть более сложное выражение:

```
a /= b + c
```

Выполняется деление a на то, что находится справа от знака равенства (сумму b и c), и сохранение результата в a. Сначала выполняется сложение, поскольку оператор сложения имеет более высокий приоритет, чем оператор присваивания. Все операторы, кроме точкой, имеют более высокий приоритет, чем операторы присваивания, которые имеют одинаковый приоритет.

Это выражение эквивалентно следующему:

```
a = a / (b + c)
```

Операторы присваивания используются по трем причинам. Во-первых, программу проще писать: то, что находится слева от оператора присваивания, не нужно повторять с правой стороны. Во-вторых, результирующее выражение проще читать. В третьих, программы могут выполняться быстрее, поскольку компилятор иногда генерирует более короткий код для вычисления выражения.

Класс Calculator

Теперь можно определить новый класс. Мы создадим класс `Calculator` для сложения, умножения, вычитания и деления чисел. Как и обычный калькулятор, он должен следить за промежуточной суммой, которую называют *накапливающим сумматором, или просто сумматором (accumulator)*. Поэтому соответствующие методы должны позволять вам задавать для сумматора определенное значение, выполнять его сброс (задавать равным нулю) и считывать значение, когда закончатся вычисления. В программе 4.6 содержится определение этого класса и тестовая программа.

Программа 4.6

```
// Реализация класса Calculator
#import <Foundation/Foundation.h>

@interface Calculator: NSObject
{
    double accumulator;
}

// методы для сумматора (accumulator)
-(void)    setAccumulator: (double) value;
-(void)    clear;
-(double)  accumulator;

// арифметические методы
-(void)    add: (double) value;
```

```
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
-(void)    setAccumulator: (double) value
{
    accumulator = value;
}

-(void) clear
{
    accumulator = 0;
}

-(double) accumulator
{
    return accumulator;
}

-(void) add: (double) value
{
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
@end

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

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

deskCalc = [[Calculator alloc] init];

[deskCalc clear];
[deskCalc setAccumulator: 100.0];
[deskCalc add: 200.];
[deskCalc divide: 15.0];
[deskCalc subtract: 10.0];
[deskCalc multiply: 5];
NSLog(@"The result is %g", [deskCalc accumulator]);
[deskCalc release];

[pool drain];
return 0;
}
```

Вывод программы 4.6

The result is 50

Класс `Calculator` имеет только одну переменную экземпляра типа `double`, содержащую значение сумматора. Сами определения методов достаточно просты.

Обратите внимание на сообщение, которое вызывает метод `multiply`:

```
[deskCalc multiply: 5];
```

Этому методу передается целый аргумент, хотя в методе предполагается значение типа `double`. Здесь нет никакой проблемы, поскольку числовые аргументы, передаваемые методам, автоматически преобразуются в соответствующий предполагаемый тип. Метод `multiply:` предполагает значение типа `double`, поэтому при вызове функции целое значение 5 автоматически преобразуется в значение с плавающей точкой двойной точности. Здесь преобразование происходит автоматически, по практике программирования все же рекомендуется задавать соответствующие типы аргументов при вызове методов.

В отличие от класса `Fraction`, где можно работать со многими дробями, в вашей программе, возможно, потребуется работа только с одним объектом `Calculator`. Имеет смысл определить новый класс, чтобы упростить работу с этим объектом. В какой-то момент вам захочется добавить графический интерфейс к своему калькулятору, чтобы пользователь мог реально щелкать кнопками на экране, как в широко распространенном приложении `calculator`.

Битовые операторы

Некоторые операторы в языке Objective-C работают с определенными битами внутри числа. Эти операторы представлены в таблице 4.2.

Табл. 4.2. Битовые операторы

Символ	Операция
&	Побитовое И
	Побитовое включающее ИЛИ
^	Побитовое исключающее ИЛИ
~	Дополнение до единицы
<<	Сдвиг влево
>>	Сдвиг вправо

Все операторы из таблицы 4.2, за исключением оператора дополнения до единицы (~), являются бинарными операторами, то есть предполагают два операнда. Битовые операции можно выполнять с любым типом целого значения, но нельзя выполнять для значений с плавающей точкой.

Побитовый оператор И

Если два значения связаны оператором И (&), то двоичные представления этих значений сравниваются бит за битом. Если какой-либо бит первого значения равен 1 и соответствующий бит второго значения равен 1, то соответствующий бит результата равен 1; во всех остальных случаях результат равен 0. Пусть b1 и b2 представляют соответствующие биты в двух операндах. В следующей таблице, которая называется *таблицей истинности (truth table)*, представлены результаты операции И для всех возможных значений b1 и b2.

b1	b2	b1 & b2
0	0	0
0	1	0
1	0	0
1	1	1

Например, если w1 и w2 определены с типом short int, w1 присвоено шестнадцатеричное значение 15 и w2 присвоено шестнадцатеричное значение 0c, то в результате следующего оператора переменной w3 присваивается значение 0x04:

w3 = w1 & w2;

Чтобы разобраться в этом, нужно представить значения w1, w2 и w3 в виде двоичных чисел. Предполагается, что мы работаем с размером типа short int в 16 бит.

w1	0000	0000	0001	0101	0x15
w2	0000	0000	0000	1100	& 0x0c
w3	0000	0000	0000	0100	0x04

Побитовые операции И часто используются для операций маскирования. Этот оператор можно использовать, чтобы задавать нулевые значения для определенных битов элемента данных. Например, в следующей строке переменной `w3` присваивается результат применения операции И к переменной `w1` и константе 3.

```
w3 = w1 & 3;
```

В результате все биты `w3`, кроме двух правых битов, становятся равными 0, а два правых бита берутся из `w1`.

Как и для всех бинарных арифметических операторов в Objective-C, бинарные битовые операторы можно использовать как операторы присваивания, добавляя знак равенства. Например, строка

```
word &= 15;
```

дает такой же результат, как

```
word = word & 15;
```

то есть присваивает значение 0 всем битам переменной `word`, кроме четырех правых битов.

Оператор побитового включающего ИЛИ

Если два значения связываются операцией побитового включающего ИЛИ (Inclusive-OR) в Objective-C, то происходит сравнение этих двух значений бит за битом. Но на этот раз, если бит первого значения равен 1 *или* соответствующий бит второго значения равен 1, то соответствующий бит результата равен 1. Ниже показана таблица истинности для оператора включающего ИЛИ.

b1	b2	b1 b2
0	0	0
0	1	1
1	0	1
1	1	1

Если переменным `w1` и `w2` типа `short int` присвоены соответственно шестнадцатеричные значения 19 и 6a, то операция включающего ИЛИ, примененная к `w1` и `w2`, даст в результате шестнадцатеричное значение 7b, как показано ниже.

w1	0000	0000	0001	1001	0x19
w2	0000	0000	0110	1010	0x6a
	0000	0000	0111	1011	0x7b

Эта операция часто используется, чтобы задавать определенные биты слова равными 1. Например, в следующей строке три правых бита переменной `w1` задаются равными 1 независимо от состояния этих битов до выполнения операции.

`w1 = w1 | 07;`

Вместо этой строки можно написать оператор присваивания:

`w1 |= 07;`

Далее мы приведем пример программы с оператором включающего ИЛИ.

Оператор побитового исключающего ИЛИ

Оператор побитового исключающего ИЛИ (Exclusive-OR), или оператор XOR, действует следующим образом: если один из соответствующих битов двух операндов равен 1 (но не оба), то соответствующий бит результата равен 1; в противном случае он равен 0. Ниже приводится таблица истинности для этого оператора.

b1	b2	$b1 \wedge b2$
0	0	0
0	1	1
1	0	1
1	1	0

Если для `w1` и `w2` заданы соответственно шестнадцатеричные значения `5e` и `d6`, то операция исключающего ИЛИ, примененная к `w1` и `w2`, даст в результате шестнадцатеричное значение `e8`, как показано ниже.

<code>w1</code>	0000	0000	0101	1110	0x5e
<code>w2</code>	0000	0000	1011	0110	\wedge 0xd6

	0000	0000	1110	1000	0xe8

Оператор дополнения до единицы

Оператор дополнения до единицы (ones complement) – это унарный оператор, который меняет биты операнда на противоположные. Каждый бит, равный 1, изменяется на 0, и каждый бит, равный 0, изменяется на 1. Таблица истинности приводится здесь просто для полноты изложения.

b1	$\sim b1$
0	1
1	0

Если переменная `w1` имеет тип `short int`, имеет длину 16 бит и для нее задано шестнадцатеричное значение `a52f`, то применение к этому значению операции дополнения до единицы даст в результате шестнадцатеричное значение `5ab0`.

<code>w1</code>	1010	0101	0010	1111	0xa52f
$\sim w1$	0101	1010	1101	0000	0x5ab0

Операцию дополнения до единицы полезно использовать, если вы не знаете размера в битах значения, к которому применяется какая-либо операция, и его использование может сделать программу менее зависимой от конкретного размера целого типа данных. Например, чтобы задать значение 0 для младшего бита переменной `w1` типа `int`, можно применить операцию И к `w1` и переменной типа `int`, содержащей все единицы, за исключением 0 в крайнем правом бите. Следующую строку на С можно применить на машинах, где целое значение представляется 32 битами:

```
w1 &= 0xFFFFFFFF;
```

Если заменить ее следующей строкой, то операция И будет применена к `w1` и нужному значению на любой машине.

```
w1 &= ~1;
```

Операция дополнения до единицы, примененная к значению `1`, даст `0` в крайнем правом бите, а все остальные биты слева будут равны `1` для значения типа любой длины (31 левый бит для 32-битных целых значений).

Приведем пример конкретной программы, где показано использование различных битовых операторов.

Программа 4.7

```
// Примеры битовых операторов
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    unsigned int w1 = 0xA0A0A0A0, w2 = 0xFFFF0000, w3 = 0x00007777;
    NSLog(@"%@", w1 & w2, w1 | w2, w1 ^ w2);
    NSLog(@"%@", ~w1, ~w2, ~w3);
    NSLog(@"%@", w1 ^ w1, w1 & ~w2, w1 | w2 | w3);
    NSLog(@"%@", w1 | w2 & w3, w1 | w2 & ~w3);
    NSLog(@"%@", ~(~w1 & ~w2), ~(~w1 | ~w2));

    [pool drain];
    return 0;
}
```

Вывод программы 4.7

```
a0a00000 fffffa0a0 5f5fa0a0
5f5f5f5f ffff ffff8888
0 a0a0 ffffff7f7
a0a0a0a0 fffffa0a0
fffffa0a0 a0a00000
```

Проработайте каждую из операций программы 4.7, чтобы убедиться, что вы понимаете, как получены результаты.

В четвертом вызове NSLog важно отметить, что побитовый оператор И имеет более высокий приоритет, чем побитовый оператор ИЛИ. Сводку приоритетов операторов см. в приложении В.

В пятом вызове NSLog иллюстрируется правило ДеМоргана: $\sim(a \& \sim b)$ равно $a \mid b$, и $\sim(\sim a \mid \sim b)$ равно $a \& b$. Последовательность операторов показывает, что операция обмена действует так, как описано в разделе по оператору исключающего ИЛИ.

Оператор левого сдвига

Если к значению применяется операция левого сдвига, то биты этого значения буквально сдвигаются влево. Для этой операции указывается число позиций (битов), на которое должно быть сдвинуто значение. Биты, которые выходят за старший бит элемента данных, утрачиваются, а младшие биты значения замещаются нулями. Например, если значение w1 равно 3, то выражение

```
w1 = w1 << 1;
```

которое можно также представить как

```
w1 <<= 1;
```

даст в результате смещение 3 на одну позицию влево, то есть w1 будет присвоено значение 6:

w1	...	00000011	0x03
w1 << 1	...	00000110	0x06

Операнд слева от оператора `<<` – это значение, к которому применяется сдвиг, а оператор справа – это количество битовых позиций, на которое должно быть сдвинуто значение. Например, если сдвинуть w1 еще на одну позицию влево, то мы получим шестнадцатеричное значение 0c.

w1	...	00000110	0x06
w1 << 1	...	00001100	0xc

Оператор правого сдвига

Как следует из этого названия, оператор правого сдвига `>>` смещает биты значения вправо. Биты, смещаемые за позицию младшего бита, теряются. Смещение вправо значения без знака (`unsigned`) приводит к замещению нулями старших битов. Замещение левой позиции для значений со знаком (`signed`) зависит от знака смещаемого вправо значения, а также от реализации этой операции в вашей компьютерной системе. Если бит знака равен 0 (для положительного значения), то происходит замещение нулями независимо от используемой машины. Но если бит знака равен 1, то на некоторых машинах происходит замещение единицами, а на других машинах – нулями. Первый тип операции на-

зывается *арифметическим правым сдвигом*, а второй тип — *логическим правым сдвигом*.

Внимание. Никогда не делайте предположений о типе правого сдвига (арифметическом или логическом), реализуемом в вашей системе. Программа, которая выполняет правый сдвиг значений со знаком, может работать правильно на одном компьютере и давать сбой на другом.

Если переменная `w1` имеет тип `unsigned int`, представленный 32 битами, и `w1` присвоено шестнадцатеричное значение `F777EE22`, то смещение `w1` на одну позицию вправо с помощью оператора

```
w1 >>= 1;
```

даст в результате шестнадцатеричное значение `7BBBBF711`, как показано ниже.

```
w1      1111 0111 0111 1110 1110 0010 0010 0xF777EE22  
w1 >> 1  0111 1011 1011 1011 1111 0111 0001 0001 0x7BBBF711
```

Если `w1` объявлена как (`signed`) `short int`, то на некоторых компьютерах будет получен тот же результат, а на других будет получено значение `FBBBBF711`, если операция выполняется как арифметический правый сдвиг.

Если выполняется сдвиг значения влево или вправо на число позиций, равное или превышающее количество бит этого значения, то Objective-C не дает определенный результат. Например, на машине, представляющей целое значение 32 битами, сдвиг целого значения влево или вправо на 32 или больше битов не обязательно даст определенный результат. Если указана отрицательная величина смещения, то результат тоже неопределенный.

Типы: `_Bool`, `_Complex` и `_Imaginary`

Завершая эту главу, упомянем еще три типа: `_Bool`, для работы с булевыми значениями (0 или 1), `_Complex` и `_Imaginary` — для работы с комплексными и мнимыми числами соответственно.

Программисты Objective-C обычно используют тип данных `BOOL` вместо `_Bool` для работы с булевыми значениями. В действительности это не тип данных, а еще одно имя для типа данных `char`. Его определяют с помощью специального ключевого слова `typedef`, которое описывается в главе 10.

Упражнения

1. Какие из следующих констант являются недопустимыми? Почему?

123.456	0x10.5	0X0G1
0001	0xFFFF	123L
0Xab05	0L	-597.25
123.5e2	.0001	+12

```

98.6F      98.7U    17777s
0996       -12E-12   07777
1234uL     1.2Fe-7   15,000
1.234L     197u      100U
0XABCDEFLoabcu +123

```

2. Напишите программу, которая преобразует 27° по Фаренгейту (F) в градусы Цельсия (C) с помощью следующей формулы:

$$C = (F - 32) / 1.8$$

3. Какой вывод даст следующая программа?

```

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    char c, d;
    c = 'd';
    d = c;
    NSLog (@"d = %c", d);

    [pool drain];
    return 0;
}

```

4. Напишите программу для вычисления полинома.

$$3x^3 - 5x^2 + 6, \text{ для } x = 2.55.$$

5. Напишите программу для вычисления выражения и вывода результатов (для вывода результатов используйте формат экспоненциального представления).

$$(3.31 \times 10^{-8} + 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$$

6. *Комплексные (Complex)* числа содержат две части: *вещественную (real)* и *мнимую (imaginary)*. Если a – это вещественная часть, а b – мнимая часть, то для представления такого числа используется форма записи

$$a + bi$$

Напишите программу на Objective-C, которая определяет новый класс с именем Complex. Соблюдая принцип, описанный для класса Fraction, определите для своего нового класса следующие методы.

```

-(void) setReal: (double) a;
-(void) setImaginary: (double) b;
-(void) print; // вывод как a + bi
-(double) real;
-(double) imaginary;

```

Напишите программу, чтобы протестировать новый класс и методы.

7. Вы разрабатываете библиотеку процедур для работы с графическими объектами. Начните с определения нового класса с именем `Rectangle` (Прямоугольник). Разработайте методы задания ширины и высоты прямоугольника, считывания их значений, а также расчета площади (`area`) и периметра (`perimeter`) прямоугольника. Предполагается, что объекты-прямоугольники описываются прямоугольниками на целочисленной сетке, например, на экране компьютера. В данном случае ширина и высота прямоугольника являются целыми значениями.

Ниже приводится секция `@interface` для класса `Rectangle`:

```
@interface Rectangle: NSObject
{
    int width;
    int height;
}

-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(int) width;
-(int) height;
-(int) area;
-(int) perimeter;

@end
```

Напишите секцию `implementation` и программу для тестирования вашего нового класса и методов.

8. Модифицируйте методы `add:`, `subtract:`, `multiply:` и `divide:` для программы 4.6, чтобы считывать результирующее значение сумматора (`accumulator`). Выполните тестирование этих новых методов.
9. Закончив упражнение 8, добавьте следующие методы к классу `Calculator` и выполните их тестирование.

```
-(double) changeSign; // изменение знака сумматора
-(double) reciprocal; // 1/(значение сумматора)
-(double) xSquared; // квадрат сумматора
```

10. Добавьте методы работы с памятью для класса `Calculator` в программе 4.6. Реализуйте следующие объявления методов и выполните их тестирование.

```
-(double) memoryClear; // очистка памяти
-(double) memoryStore; // запись в память значения сумматора
-(double) memoryRecall; // чтение из памяти значения сумматора
-(double) memoryAdd; // добавление значения сумматора к памяти
-(double) memorySubtract // вычитание значения сумматора из памяти
```

Каждый из этих методов должен возвращать значение сумматора.

Глава 5

Циклы в программах

Язык Objective-C позволяет повторять последовательности кода различными способами. Темой этой главы являются циклы. Они могут формироваться с помощью следующих операторов.

- Оператор `for`
- Оператор `while`
- Оператор `do`

Начнем с простого примера: подсчет суммы.

Составим треугольник из 15 шаров (рис. 5.1).

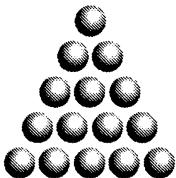


Рис. 5.1. Пример треугольной компоновки

Первый ряд состоит из одного шара, второй ряд – из двух, и т.д. Количество шаров в треугольнике, состоящем из n рядов, равно сумме чисел от 1 до n . Эта сумма называется *треугольным числом*.

Если начать с 1, то четвертое треугольное число будет равно сумме последовательных чисел от 1 до 4 ($1 + 2 + 3 + 4$), то есть 10. Напишем программу, которая вычисляет и выводит на экран значение восьмого треугольного числа. Это число легко сосчитать в уме, но можно написать программу на Objective-C, чтобы выполнить эту задачу (программа 5.1).

Программа 5.1

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

// Программа вычисления восьмого треугольного числа (triangularNumber)

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

    triangularNumber = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

    NSLog (@"The eighth triangular number is %i", triangularNumber);

    [pool drain];
    return 0;
}
```

Вывод программы 5.1

The eighth triangular number is 36 (восьмое треугольное число = 36)

Способ, применяемый в программе 5.1, вполне подходит для относительно небольших треугольных чисел, но что делать, если нам нужно вычислить, например, значение 200-го треугольного числа? Не складывать же в явном виде все целые числа от 1 до 200? К счастью, имеется более простой способ – возможность выполнения операций в цикле. Для реализации циклов в языке Objective-C имеется три оператора.

Оператор for

В программе 5.2 вычисляется 200-е треугольное число. На ее примере мы рассмотрим, как работает оператор for.

Программа 5.2

```
// Программа вычисления 200-го треугольного числа (triangularNumber)
// Пример использования оператора for

#import <Foundation/Foundation.h>

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

```

```
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
    int n, triangularNumber;  
  
    triangularNumber = 0;  
  
    for ( n = 1; n <= 200; n = n + 1 )  
        triangularNumber += n;  
  
    NSLog (@"The 200th triangular number is %i", triangularNumber);  
  
    [pool drain];  
    return 0;  
}
```

Вывод программы 5.2

The 200th triangular number is 20100 (200-е треугольное число = 20100)

Программа 5.2 требует некоторых пояснений. Для расчета 200-го треугольного числа используется тот же метод, что и для 8-го треугольного числа: вычисляется сумма чисел от 1 до 200.

Перед оператором `for` переменной `triangularNumber` присваивается значение 0. Все переменные необходимо инициализировать (присвоить им значение) до того, как они будут использованы в программе. Определенным типам переменных начальные значения присваиваются по умолчанию, но надежнее присвоить переменным нужные значения явно.

Оператор `for` позволяет обойтись без явного написания всех чисел от 1 до 200. В общем виде оператор `for` имеет формат

```
for (начальное_выражение; условие_цикла; выражение_цикла)  
    программный оператор
```

Три выражения, заключенные в круглые скобки: *начальное_выражение*, *условие_цикла* и *выражение_цикла* — задают среду выполнения программного цикла. Следующий за ними *программный оператор* (который заканчивается символом «точка с запятой») может быть любым оператором Objective-C. Количество выполнений этого оператора определяется параметрами, заданными в операторе `for`.

Первый компонент оператора `for`, *начальное_выражение*, задает начальные значения до выполнения цикла. В программе 5.2 эта часть оператора `for` задает начальное значение `n`, равное 1. Второй компонент оператора `for` указывает условия, необходимые для продолжения цикла. Цикл повторяется, пока это условие истинно. В программе 5.2 *условие_цикла* указывается следующим выражением отношения:

`n <= 200`

Это выражение означает «*n* меньше или равно 200». Оператор «меньше или равно» (знак «меньше» [$<$], после которого сразу следует знак «равно» [=]) – это оператор отношения. Операторы отношения используются для проверки условий. Результатом проверки является ответ «да» (или TRUE), если условие выполняется, или «нет» (или FALSE), если условие не выполняется.

В таблице 5.1 приводится список всех операторов отношения Objective-C.

Табл. 5.1. Операторы отношения

Оператор	Описание	Пример
$==$	Равно	<code>count == 10</code>
$!=$	Не равно	<code>flag != DONE</code>
$<$	Меньше, чем	<code>a < b</code>
\leq	Меньше или равно	<code>low \leq high</code>
$>$	Больше, чем	<code>points > POINT_MAX</code>
\geq	Больше или равно	<code>j \geq 0</code>

Операторы отношения имеют меньший приоритет, чем все арифметические операторы. Выражение

`a < b + c`

выполняется так:

`a < (b + c)`

Оно истинно (TRUE), если значение *a* меньше, чем значение *b* + *c*, и ложно (FALSE) в противном случае.

Обратите внимание на оператор «равно» (==); его не следует путать с оператором присваивания (=). Выражение

`a == 2`

проверяет, равно ли значение *a* значению 2, а выражение

`a = 2`

присваивает значение 2 переменной *a*.

Выбор оператора отношения зависит от типа проверки и от ваших предпочтений. Например, выражение отношения

`n \leq 200`

эквивалентно

`n < 201`

В приведенном выше примере выполнение программного оператора, который образует тело цикла `for (triangularNumber += n;)`, повторяется *до тех пор*, пока результат проверки выражения отношения равен TRUE (в данном случае – пока значение `n` меньше или равно 200). В этом операторе значение `n` прибавляется к значению переменной `triangularNumber`.

Если условие_цикла не выполнено, то программа продолжает выполняться со следующего оператора. В этой программе выполнение продолжается с оператора `NSLog`.

Последний компонент оператора `for` содержит выражение, которое вычисляется каждый раз после выполнения тела цикла. В программе 5.2 выражение_цикла к значению `n` прибавляется 1. Значение `n` увеличивается на 1 каждый раз после прибавления этого значения к значению `triangularNumber`, и `n` изменяется от 1 до 201.

Последнее значение `n` (201) не прибавляется к значению `triangularNumber`, поскольку цикл заканчивается, как только перестает выполняться условие цикла (`n` становится равным 201).

Итак, оператор `for` выполняется следующим образом.

1. Сначала вычисляется начальное выражение. В этом выражении обычно задается (инициализируется) переменная, используемая внутри цикла, которую называют *индексной переменной*. Ей присваивается некоторое начальное значение (например, 0 или 1).
2. Проверяется условие цикла. Если это условие не удовлетворяется (выражение имеет значение FALSE), то цикл немедленно прекращается, и выполнение продолжается с оператора, расположенного непосредственно после цикла.
3. Выполняется программный оператор, содержащий тело цикла.
4. Вычисляется выражение цикла. Это выражение обычно используется, чтобы изменить значение индексной переменной. Довольно часто это прибавление к переменной 1 или вычитание из нее 1.
5. Возврат к шагу 2.

Условие цикла проверяется сразу после входа в цикл, до первого выполнения тела цикла. Нельзя ставить символ «точка с запятой» после закрывающей круглой скобки, поскольку это сразу прекращает цикл.

Программа 5.2 генерирует все 200 первых треугольных чисел. Создадим таблицу этих чисел. Для экономии места мы выведем таблицу, содержащую только первые 10 треугольных чисел. Эта задача выполняется в программе 5.3.

Программа 5.3

```
// Программа, генерирующая таблицу треугольных чисел  
  
#import <Foundation/Foundation.h>
```

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

    NSLog (@"TABLE OF TRIANGULAR NUMBERS");
    NSLog (@" n Sum from 1 to n");
    NSLog (@"--- -----");

    triangularNumber = 0;

    for ( n = 1; n <= 10; ++n ) {
        triangularNumber += n;
        NSLog (@" %i      %i", n, triangularNumber);
    }

    [pool drain];
    return 0;
}
```

Вывод программы 5.3

TABLE OF TRIANGULAR NUMBERS (Таблица треугольных чисел)

n Сумма от 1 до n

— -----

1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

В программе 5.3 первые три оператора `NSLog` выводят заголовок и названий колонок вывода.

После этого вычисляются первые 10 треугольных чисел. Переменная `n` используется, чтобы определить текущий номер, для которого вычисляется сумма от 1 до `n`, в переменной `triangularNumber` сохраняется значение `n`-го треугольного числа.

Выполнение оператора `for` начинается с присваивания переменной `n` значения 1. Программный оператор, следующий непосредственно после оператора `for`, содержит тело цикла. Чтобы выполнить в цикле не один, а несколько программных оператор, нужно заключить их в фигурные скобки. Система интерпретирует этот блок (*block*) как один элемент. Вообще говоря, в любом месте программы на Objective-C вместо одного оператора можно использовать блок операторов, если заключить этот блок в фигурные скобки.

В программе 5.3 тело цикла состоит из выражения, в котором значение `n` добавляется к значению `triangularNumber`, и следующего за ним оператора `NSLog`. Обратите внимание, что эти операторы расположены с отступом, поскольку они образуют часть цикла `for`. Существуют разные стили программирования; некоторые из них записывают цикл так, чтобы открывающая фигурная скобка располагалась в отдельной строке после `for`.

```
for ( n = 1; n <= 10; ++n )
{
    triangularNumber += n;
    NSLog(@"%@", n, triangularNumber);
}
```

Следующее треугольное число вычисляется путем добавления значения `n` к предыдущему треугольному числу. При первом входе в цикл треугольное число равно 0, поэтому новое значение `triangularNumber` при `n`, равном 1, равно 1. Затем выводятся значения `n` и `triangularNumber` с соответствующим числом пробелов, вставляемых в строку формата, чтобы значения этих переменных были выровнены под заголовками соответствующих колонок.

После выполнения тела цикла вычисляется выражение цикла. Но это выражение в данном операторе `for` выглядит странно:

`++n`

Однако `++n` – это вполне допустимое выражение Objective-C. Оно представляет новый оператор Objective-C: *оператор приращения, или увеличения (increment operator)*. Этот оператор добавляет 1 к своему операнду. Приращение на 1 весьма распространено, и для этой цели существует специальный оператор. Выражение `++n` эквивалентно выражению `n = n + 1`.

Конечно, ни один язык программирования, содержащий оператор прибавления 1, не будет полным без соответствующего оператора вычитания 1. Этот оператор называется *оператором уменьшения (decrement operator)*, и для него используется двойной знак «минус». Выражение

```
bean_counter = bean_counter - 1
```

эквивалентно выражению

```
--bean_counter
```

Некоторые программисты любят помещать `++` или `--` после имени переменной, как в `n++` или `bean_counter--`.

Вероятно, вы обратили внимание, что последняя строка вывода программы 5.3 не выровнена. Вы можете устраниТЬ это небольшое несоответствие, заменив соответствующий оператор программы 5.3 следующим оператором `NSLog`:

```
NSLog(@"%@", n, triangularNumber);
```

Ниже приводится вывод модифицированной программы (назовем ее программой 5.3A).

Вывод программы 5.3A

TABLE OF TRIANGULAR NUMBERS

n Sum from 1 to n

— -----

1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

Основное изменение, внесенное в оператор `NSLog` — это включение описания ширины поля. Символы `%2i` указывают процедуре `NSLog`, что нужно вывести целое значение в определенной позиции и использовать для вывода не менее двух позиций. Любое целое число, занимающее менее двух позиций, будет выводиться с ведущим пробелом. Это называется *выравниванием по правому краю* (*right justification*).

Ввод с клавиатуры

Программа 5.2 вычисляет только 200-е треугольное число. Как быть, если вам нужно вычислить 50-е или 100-е треугольное число? Нужно изменить программу, чтобы цикл `for` был выполнен соответствующее число раз, и изменить оператор `NSLog`, чтобы вывести соответствующее сообщение.

Наиболее простое решение — сделать так, чтобы программа запрашивала, какое число вы хотите вычислить, а затем вычисляла его. Чтобы реализовать это решение, можно использовать процедуру с именем `scanf`. Процедура `scanf` действует аналогично процедуре `NSLog`, но процедура `NSLog` выводит значения на экран, а процедура `scanf` позволяет вводить значения в программу. Конечно, если в программе на Objective-C используется графический интерфейс пользователя (User Interface, UI) для такого приложения, как Сосоа или iPhone, то ни `NSLog`, ни `scanf` вообще не потребуется.

Программа 5.4 спрашивает у пользователя, какое треугольное число нужно вычислить, вычисляет это число и выводит результаты.

Программа 5.4

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

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

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int n, number, triangularNumber;

    NSLog (@"What triangular number do you want?");
    scanf ("%i", &number);

    triangularNumber = 0;

    for ( n = 1; n <= number; ++n )
        triangularNumber += n;

    NSLog (@"Triangular number %i is %i\n", number, triangularNumber);

    [pool drain];
    return 0;
}
```

В приведенном ниже выводе программы число, которое вводит пользователь (100), выделено полужирным шрифтом.

Вывод программы 5.4

What triangular number do you want? (Какое треугольное число вам нужно?)

100

Triangular number 100 is 5050 (100-е треугольное число равно 5050)

Согласно этому выводу, пользователь ввел число 100. Программа вычислила 100-е треугольное число и вывела результат 5050 на терминал. В первом опера-

тоте NSLog программы 5.4 у пользователя запрашивается ввод числа. Напомним пользователю, что конкретно нужно ввести. После вывода сообщения вызывается процедура scanf. Первым аргументом для scanf является строка формата, которая *не* начинается с символа @. У NSLog первым аргументом всегда является объект NSString, а у scanf – С-строка. Перед строками в стиле C не ставится символ @.

Строка формата указывает scanf, какие типы значений должны считываться с консоли (из окна терминала, если вы компилируете программу с помощью приложения Terminal). Как и в случае NSLog, для указания целого значения используются символы %i.

Второй аргумент для процедуры scanf указывает, где должно быть сохранено значение, которое вводит пользователь. В этом случае перед именем переменной необходим символ &. Этот символ, который является на самом деле оператором, рассматривается в главе 13.

При вызове scanf указывается, что целое значение должно быть прочитано и сохранено в переменной number. Это значение представляет номер того треугольного числа, которое требуется вычислить.

После ввода этого значения пользователем (и нажатия клавиши Enter на клавиатуре, указывающей на завершение ввода) программа вычисляет запрошенное треугольное число. Это происходит так же, как; Вместо использования в программе 5.2 предела, равного 200, в этой программе используется значение number.

Затем выводятся результаты, и выполнение программы заканчивается.

Вложенные циклы for

Программа 5.4 позволяет вычислить любое треугольное число. Теперь предположим, что нам надо вычислить пять треугольных чисел. Пользователь мог бы запускать эту программу пять раз, вычисляя все числа по очереди.

С точки зрения изучения Objective-C, нужно, чтобы программа сама обрабатывала эту ситуацию. Для этого достаточно вставить в программу цикл для повторения всей последовательности вычислений. Вы можете использовать оператор for, чтобы задать такой цикл. Этот способ показан на примере программы 5.5 и вывода ее результатов.

Программа 5.5

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

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

```
int n, number, triangularNumber, counter;

for ( counter = 1; counter <= 5; ++counter ) {
    NSLog (@"What triangular number do you want?");
    scanf ("%i", &number);

    triangularNumber = 0;

    for ( n = 1; n <= number; ++n )
        triangularNumber += n;

    NSLog (@"Triangular number %i is %i", number, triangularNumber);
}

[pool drain];
return 0;
}
```

Вывод программы 5.5

What triangular number do you want? (Какое треугольное число вам нужно?)

12

Triangular number 12 is 78 (12-е треугольное число равно 78)

What triangular number do you want?

25

Triangular number 25 is 325

What triangular number do you want?

50

Triangular number 50 is 1275

What triangular number do you want?

75

Triangular number 75 is 2850

What triangular number do you want?

83

Triangular number 83 is 3486

Эта программа содержит два уровня операторов for. Внешний оператор for:

```
for ( counter = 1; counter <= 5; ++counter )
```

Он указывает, что цикл программы должен быть выполнен пять раз. Значение счетчика (*counter*) задается равным 1 и затем наращивается с шагом 1 и условием, что он меньше или равен 5 (т.е. пока он не достигнет 6).

В отличие от предыдущей программы, переменная *counter* в других местах программы не используется. Она применяется исключительно как счетчик цикла в операторе *for*. Тем не менее, поскольку это переменная, вы должны объявить ее в программе.

В цикл этой программы включены все остальные программные операторы, заключенные в фигурные скобки. Эту программу можно описать следующим образом.

```
Пять раз
{
    Получение числа от пользователя.
    Вычисление запрашиваемого треугольного числа.
    Вывод результатов.
}
```

Часть «Вычисление запрашиваемого треугольного числа» фактически состоит из задания значения 0 для переменной *triangularNumber* и цикла *for*, в котором вычисляется треугольное число. Таким образом, оператор *for* оказывается внутри другого оператора *for*. Вложение циклов может продолжаться до любого уровня.

Использование отступов становится крайне важным при работе с усложненными программными конструкциями, такими как вложенные операторы *for*. Так вы сможете быстро определить, какие операторы содержатся внутри каждого оператора *for*.

Варианты цикла *for*

Прежде чем закончить рассмотрение цикла *for*, рассмотрим синтаксические вариации, допустимые при формировании цикла. При написании цикла *for* может оказаться, что вам нужно инициализировать более одной переменной, прежде чем начнется цикл, или нужно проверять несколько условий на каждом шаге цикла. Вы можете включать несколько выражений в любые поля цикла *for*, разделяя выражения запятыми. Например, в операторе, который начинается с

```
for ( i = 0, j = 0; i < 10; ++i )
    ...

```

значения *i* и *j* до начала цикла задаются равными 0. Эти два выражения, *i=0* и *j=0*, разделены запятой, и оба выражения считаются частью поля *начальное выражение* этого цикла. Рассмотрим еще один пример. Он начинается с оператора

```
for ( i = 0, j = 100; i < 10; ++i, j -= 10 )
    ...

```

Здесь задаются две индексные переменные, *i* и *j*, которым присваиваются до начала цикла начальные значения: 0 и 100. После каждого выполнения тела цикла значение *i* увеличивается на 1, значение *j* уменьшается на 10.

Вы можете включать несколько выражений в определенное поле оператора `for` или пропускать несколько полей в этом операторе. Пропустив соответствующее поле, на его месте нужно оставить точку с запятой. Наиболее распространенный случай пропуска поля в операторе `for` возникает, когда не нужно инициализировать начальное выражение. Вы можете просто оставить поле `начальное выражение` пустым, как в следующем случае, оставив только символ «точка с запятой».

```
for ( ; j != 100; ++j )
```

...

Этот оператор можно использовать, например, если для переменной *j* до входа в цикл уже задано некоторое начальное значение.

Цикл `for`, в котором опущено поле `условие_цикла`, на самом деле является бесконечным циклом. Такой цикл можно использовать, если для выхода из цикла используются другие средства (например, операторы `return`, `break` или `goto`, которые описываются далее).

Вы можете определять переменные в составе начального выражения цикла `for` с помощью рассмотренных ранее типичных способов определения переменных. Например, следующий оператор задает цикл `for` с целой переменной `counter`, которая одновременно определяется и инициализируется со значением 1.

```
for ( int counter = 1; counter <= 5; ++counter )
```

Переменная `counter` действует только при выполнении этого цикла (она называется *локальной переменной цикла*) и недоступна извне данного цикла.

В следующем цикле определяются две целые переменные и задаются их начальные значения.

```
for ( int n = 1, triangularNumber = 0; n <= 200; ++n )
    triangularNumber += n;
```

Последний вариант цикла, который называется *быстрым перечислением* (*fast enumerations*) коллекции объектов, подробно описывается в главе 15.

Оператор `while`

Оператор `while` еще больше расширяет возможности циклов в языке Objective-C. Эта часто применяемая конструкция имеет следующий синтаксис:

```
while (выражение )
    программный оператор
```

Выполняется оценка *выражения*, указанного в круглых скобках. Если результатом оценки выражения является значение `TRUE`, то выполняется *программный оператор*. После выполнения этого оператора (или операторов, заключенных в фигурные скобки) снова выполняется оценка *выражения*. Если результатом оценки

выражения является значение TRUE, то снова выполняется *программный оператор*. Этот процесс продолжается, пока результатом оценки выражения не станет значение FALSE, после чего цикл прекращается. Выполнение программы затем продолжается, начиная с оператора, который следует за программным оператором.

В качестве примера в следующей программе задается цикл while, используемый просто для счета от 1 до 5.

Программа 5.6

```
// В этой программе вводится оператор while

#import <Foundation/Foundation.h>

#import <stdio.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int count = 1;

    while ( count <= 5 ) {
        NSLog (@"%i", count);
        ++count;
    }

    [pool drain];
    return 0;
}
```

Вывод программы 5.6

```
1
2
3
4
5
```

В этой программе сначала задается значение переменной count, равное 1; затем начинается выполнение цикла while. Поскольку значение count меньше 5, выполняется следующий оператор. В фигурных скобках содержится тело цикла с оператором NSLog и оператором, который наращивает значение переменной count. Из результатов вывода этой программы видно, что цикл выполняется пять раз, то есть до тех пор, пока значение count не достигнет 5.

Эту же задачу можно выполнить с помощью оператора for. Оператор for всегда можно преобразовать в эквивалентный оператор while, и наоборот. Общую форму оператора for

```
for (начальное_выражение; условие_цикла; выражение_цикла )  
    программный оператор
```

можно выразить в эквивалентной форме оператора `while`:

```
начальное_выражение;  
while (условие_цикла )  
{  
    программный оператор  
    выражение_цикла;  
}
```

Со временем вы поймете, когда удобнее использовать оператор `while`, а когда – оператор `for`. Обычно для цикла, который должен быть выполнен определенное количество раз, подходит оператор `for`. Если начальное выражение, выражение цикла и условие цикла включают одну и ту же переменную, то обычно правильным выбором будет оператор `for`.

В следующей программе содержится еще один пример использования оператора `while`. Эта программа вычисляет наибольший общий делитель двух целых значений. Наибольший общий делитель (далее мы будем применять для него сокращение `gcd` – greatest common divisor) – это наибольшее целое число, на которое делятся без остатка два целых числа. Например, `gcd` чисел 10 и 15 равен 5, так как 5 – наибольшее целое число, на которое делятся без остатка 10 и 15.

Алгоритм получения `gcd` двух произвольных целых чисел основывается на процедуре, впервые разработанной Евклидом примерно в 300 г. до нашей эры. Ее можно сформулировать следующим образом.

Задача. Найти наибольший общий делитель двух неотрицательных целых чисел `u` и `v`.

Шаг 1. Если `v` равно 0, то процедура завершена и `gcd` равен `u`.

Шаг 2. Вычислить `temp = u % v`, `u = v`, `v = temp` и вернуться к шагу 1.

Анализ шагов этого алгоритма показывает, что шаг 2 повторяется, пока значение `v` не станет равным 0. Из этого следует, что на Objective-C этот алгоритм следует реализовать опредевством оператора `while`.

Программа 5.7 выполняет поиск `gcd` двух неотрицательных целых значений, вводимых пользователем.

Программа 5.7

```
// Эта программа ищет наибольший общий делитель  
// двух неотрицательных целых значений
```

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

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

```

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    unsigned int u, v, temp;

    NSLog(@"Please type in two nonnegative integers.");
    scanf ("%u%u", &u, &v);

    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    NSLog(@"Their greatest common divisor is %u", u);

    [pool drain];

    return 0;
}

```

Вывод программы 5.7

Please type in two nonnegative integers. (Введите два неотрицательных целых числа)

150 35

Their greatest common divisor is 5 (Их наибольший общий делитель равен 5)

Вывод программы 5.7 (Повторный запуск)

Please type in two nonnegative integers. (Введите два неотрицательных числа.)

1026 540

Their greatest common divisor is 54 (Их наибольший общий делитель равен 54.)

После ввода двух целых значений и сохранения в переменных `u` и `v` (с использованием символов формата `%u` для чтения целого значения без знака) начинается цикл `while`, где вычисляется наибольший общий делитель. После выхода из цикла `while` выводится значение `u`, представляющее `gcd` значения переменной `v` и исходного значения переменной `u` с соответствующим сообщением.

Тот же алгоритм будет использоваться для поиска наибольшего общего делителя в главе 7. В следующей программе с оператором `while` решается задача обращения порядка цифр целого числа, которое вводится с терминала. Например, если пользователь вводит число 1234, программа должна вывести результат 4321.

Примечание. При использовании вызовов `NSLog` каждая цифра будет выводиться в отдельной строке вывода. Программисты на C, которые знакомы с функцией `printf`, могут использовать эту процедуру вместо последовательного вывода цифр.

Чтобы написать такую программу, нужно сначала составить алгоритм, который выполняет поставленную задачу. Часто анализ метода решения задачи приводит к определенному алгоритму. Для задачи обращения цифр числа решение можно сформулировать как «успешное чтение цифр числа справа налево». Вы можете написать компьютерную программу, читающую числа, разработав процедуру извлечения каждой цифры числа, начиная с правой. Полученную цифру можно затем вывести на терминал как следующую цифру обращенного числа.

Чтобы извлечь правую цифру целого числа, можно взять остаток от деления этого числа на 10. Например, $1234 \% 10$ дает значение 4. Это правая цифра числа 1234 и первая цифра обращенного числа. Следующую цифру можно получить так же, если сначала выполнить целочисленное деление этого числа на 10. Деление $1234 / 10$ дает в результате 123, а операция $123 \% 10$ дает 3, что является следующей цифрой обращенного числа.

Вы можете продолжать эту процедуру, пока не будет извлечена последняя цифра. Если результат последнего деления на 10 равен 0, значит, все цифры извлечены.

Программа 5.8 запрашивает у пользователя ввод числа и затем выводит цифры этого числа, начиная с крайней правой и заканчивая крайней левой.

Программа 5.8

```
// Программа обращения цифр числа

#import <Foundation/Foundation.h>

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

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int number, right_digit;

    NSLog (@"Enter your number.");
    scanf ("%i", &number);

    while ( number != 0 ) {
        right_digit = number % 10;
        NSLog (@"%i", right_digit);
        number /= 10;
    }

    [pool drain];
    return 0;
}
```

Вывод программы 5.8

Enter your number. (Введите свое число)

13579

9

7

5

3

1

Оператор do

В двух конструкциях циклов, которые рассматривались выше, перед выполнением цикла выполняется проверка условий. Тело цикла не будет выполнено ни разу, если не удовлетворяются эти условия. При разработке программ иногда требуется, чтобы проверка условий выполнялась в конце цикла, а не в начале. В языке Objective-C для такой ситуации имеется отдельная конструкция, которая называется оператором **do**. Этот оператор имеет следующий синтаксис.

```
do
    программный оператор
    while ( выражение );
```

Оператор **do** выполняется следующим образом. Первым выполняется *программный оператор*. Затем оценивается *выражение* в круглых скобках. Если результатом оценки *выражения* является значение **TRUE**, то цикл продолжается и снова выполняется *программный оператор*. Выполнение *программного оператора* повторяется до тех пор, пока результатом оценки *выражения* является значение **TRUE**. Если результатом оценки *выражения* оказывается значение **FALSE**, цикл заканчивается и выполняется следующий по порядку оператор программы.

В операторе **do**, в отличие от оператора **while**, условия цикла помещаются в конец цикла, а не в начало.

В программе 5.8 оператор **while** использовался для вывода цифр числа в обратном порядке. Вернемся к этой программе и определим, что произойдет, если пользователь введет число 0 вместо 13579. Цикл оператора **while** не будет выполнен ни разу, и ничего не будет выведено на экран. Если использовать оператор **do** вместо оператора **while**, цикл программы будет выполнен хотя бы один раз, что гарантирует вывод хотя бы одной цифры. Использование оператора **do** показано в программе 5.9.

Программа 5.9

```
// Программа обращения цифр числа

#import <Foundation/Foundation.h>

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

```
{  
  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
    int number, right_digit;  
  
    NSLog(@"Enter your number.");  
    scanf ("%i", &number);  
  
    do {  
        right_digit = number % 10;  
        NSLog(@"%@", right_digit);  
        number /= 10;  
    }  
    while ( number != 0 );  
  
    [pool drain];  
    return 0;  
}
```

Вывод программы 5.9

```
Enter your number. (Введите свое число)  
135  
5  
3  
1
```

Вывод программы 5.9 (Повторный запуск)

```
Enter your number.  
0  
0
```

При вводе 0 программа выводит цифру 0.

Оператор break

Иногда при выполнении цикла требуется выйти из цикла, как только возникает определенное условие, например, если определено состояние ошибки или преждевременно достигнут конец данных. Для этой цели можно использовать оператор `break`. При выполнении оператора `break` программа сразу выходит из цикла любого типа (`for`, `while` или `do`). Все остальные операторы цикла пропускаются, и выполнение цикла прекращается. Выполнение продолжается с первого оператора, следующего после цикла.

Если оператор `break` выполняется из набора вложенных циклов, то прекращается выполнение только вложенного цикла, в котором выполнен этот оператор `break`.

Оператор `break` – это просто ключевое слово `break`, после которого следует символ «точка с запятой»:

```
break;
```

Оператор `continue`

Оператор `continue` не прекращает выполнение цикла. Если выполнен оператор `continue`, то пропускаются все следующие операторы до конца цикла, в противном случае выполнение цикла продолжается обычным образом.

Оператор `continue` чаще всего используется для обхода группы операторов внутри цикла в зависимости от некоторого условия. Его формат:

```
continue;
```

Не используйте операторы `break` или `continue`, пока не освоитесь с написанием программных циклов и выхода из них. Эти операторы нужно применять с осторожностью, поскольку они затрудняют чтение и отслеживание работы программ.

Упражнения

1. Напишите программу создания и вывода таблицы значений n и n^2 для целых значений n от 1 до 10. Обязательно выведите заголовки колонок.
2. Треугольное число (`triangularNumber`) можно вычислить для любого целого числа по формуле $\text{triangularNumber} = n(n + 1)/2$

Например, 10-е треугольное число (55) можно вычислить путем подстановки в эту формулу значения 10 вместо n . Напишите программу, которая создает таблицу треугольных чисел с помощью этой формулы. Эта программа должна вычислять каждое пятое треугольное число для значений n от 5 до 50 (то есть 5, 10, 15, ..., 50).

3. Факториал целого числа n (записывается как $n!$) – это произведение последовательных чисел от 1 до n . Например, 5 факториал рассчитывается следующим образом: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Напишите программу расчета и вывода первых десяти факториалов.

4. Если перед описателем ширины поля поставлен знак «минус», это поле выводится с выравниванием по левому краю. Замените следующим оператором `NSLog` соответствующий оператор программы 5.2, выполните программу и сравните вывод обеих программ:

```
NSLog(@"%@", @"%-2i %i", n, triangularNumber);
```

5. Программа 5.5 позволяет пользователю ввести только пять чисел. Модифицируйте эту программу, чтобы пользователь мог ввести количество треугольных чисел для расчета.
6. Перепишите программы 5.2–5.5, заменив все случаи использования оператора `for` на эквивалентные операторы `while`. Выполните каждую из этих программ, чтобы проверить идентичность результатов обеих версий.
7. Что произойдет, если ввести отрицательное число в программу 5.8? Попробуйте выполнить это и посмотрите, что получится.
8. Напишите программу, которая вычисляет сумму цифр целого числа. Например, сумма цифр числа 2155 равна $2 + 1 + 5 + 5$, то есть 13. Программа должна допускать любое число, введенное пользователем.

Глава 6

Принятие решений

В любом языке программирования имеется возможность принятия решений. В языке программирования Objective-C имеются несколько конструкций для принятия решений.

- Оператор `if`
- Оператор `switch`
- Условный оператор

Оператор `if`

В общем виде оператор `if` имеет следующий формат.

```
if ( выражение )
    программный оператор
```

Представим, что нам нужно преобразовать утверждение «Если нет дождя, я пойду купаться» в операторы языка Objective-C. Используя показанный выше формат, это можно «записать» следующим образом.

```
if ( нет дождя )
    я пойду купаться
```

Оператор `if` используется, чтобы выполнить программный оператор в зависимости от указанных условий (или операторы, если они заключены в фигурные скобки). Аналогичным образом, в программном операторе

```
if ( count > MAXIMUM_SONGS )
    [playlist maxExceeded];
```

сообщение `maxExceeded` отправляется `playlist` (список воспроизведения), *только* если значение переменной `count` больше значения `MAXIMUM_SONGS`, иначе этот оператор игнорируется.

Рассмотрим пример. Предположим, что вам нужно написать программу, которая принимает целое значение, введенное с клавиатуры, и затем выводит абсолютное значение этого целого числа. Проще всего вычислить абсолютное значение числа – взять его с противоположным знаком, если оно меньше нуля. Фраза «если оно меньше нуля» означает, что программа должна принять решение. Для этого можно использовать оператор if, как показано в программе 6.1.

Программа 6.1

```
// Вычисление абсолютного значения целого числа

#import <Foundation/Foundation.h>

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

    int number;

    NSLog (@"Type in your number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        number = -number;

    NSLog (@"The absolute value is %i", number);

    [pool drain];
    return 0;
}
```

Вывод программы 6.1

```
Type in your number: (Введите свое число)
-100
The absolute value is 100 (Абсолютное значение равно 100)
```

Вывод программы 6.1 (повторный запуск)

```
Type in your number:
2000
The absolute value is 2000
```

Эта программа запускалась два раза, чтобы проверить правильность ее работы. Мы проверили оба возможных результата принятия решений.

После вывода сообщения для пользователя и сохранения введенного целого значения в переменной `number` программа проверяет, не является ли значение `number` отрицательным. Если да, то выполняется следующий программный оператор, который изменяет знак `number` на противоположный. Если значение `number` не меньше нуля, этот оператор автоматически пропускается. (В этом случае не нужно изменять знак на противоположный.) Затем выполняется вывод значения `number` и завершается выполнение программы.

Рассмотрим еще одну программу с оператором `if`. Мы добавим в класс `Fraction` еще метод `convertToNum`. Этот метод будет представлять значение дроби в виде вещественного числа — выполнять деление числителя (`numerator`) на знаменатель (`denominator`). Результат будет возвращаться как значение двойной точности (`double`). Так, для дроби $1/2$ нужно возвращать значение 0.5.

Для такого метода можно использовать, например, следующее объявление.

```
-(double) convertToNum;
```

Его определение можно записать в следующем виде:

```
-(double) convertToNum
{
    return numerator / denominator;
}
```

Хорошо, но недостаточно. При таком определении этот метод имеет две серьезные проблемы. Первая относится к работе с арифметическими преобразованиями. Напомним, что `numerator` и `denominator` — это целые переменные экземпляра. А что происходит при делении целых значений? Оно выполняется как целое деление! Если нужно преобразовать дробь $1/2$, то представленный код даст значение нуль! Это легко исправить с помощью оператора приведения типа, преобразующего один или оба операнда в значение с плавающей точкой до операции деления:

```
(double) numerator / denominator
```

Поскольку этот оператор имеет относительно высокий приоритет, значение `numerator` преобразуется в тип `double` до выполнения деления. Вам не обязательно преобразовывать `denominator`, достаточно сделать это для одного из operandов.

Вторая проблема заключается в том, что нужно проверять деление на нуль (вы должны всегда выполнять эту проверку). Ниже показана модифицированная версия метода `convertToNum`.

```
-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
```

```
    else
        return 0.0;
}
```

Мы решили возвращать значение 0.0, если знаменатель дроби равен нулю. Можно использовать и другие варианты (например, вывод сообщения об ошибке, создание исключительного состояния и т.д.).

Введем этот метод в программе 6.2.

Программа 6.2

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

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void)    print;
-(void)    setNumerator: (int) n;
-(void)    setDenominator: (int) d;
-(int)     numerator;
-(int)     denominator;
-(double)  convertToNum;
@end

@implementation Fraction
-(void) print
{
    NSLog(@"%@", numerator / denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

-(int) numerator
{
    return numerator;
```

```
}

-(int) denominator
{
    return denominator;
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 0.0;
}
@end

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

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

    [aFraction setNumerator: 1]; // 1-я дробь 1/4
    [aFraction setDenominator: 4];

    [aFraction print];
    NSLog (@» =«);
    NSLog (@»%g», [aFraction convertToNum]);

    [bFraction print]; // значение не было присвоено
    NSLog (@" =");
    NSLog (@"%g", [bFraction convertToNum]);
    [aFraction release];
    [bFraction release];

    [pool drain];
    return 0;
}
```

Вывод программы 6.2

```
1/4
=
0.25
0/0
```

```
=  
0
```

После присваивания aFraction значения $1/4$ используется метод convertToNum, чтобы преобразовать эту дробь в десятичное значение. Затем это значение выводится как 0.25.

Во втором случае значение bFraction не задано явным образом, поэтому его числителю и знаменателю присваивается значение 0, что происходит по умолчанию для переменных экземпляра. Это объясняет результат метода print, а оператор if в методе convertToNum возвращает значение 0.

Конструкция if-else

Если нужно сказать, является ли число четным или нечетным, вы, скорее всего, определите это по последней цифре числа. Если эта цифра равна 0, 2, 4, 6 или 8, значит, число является четным.

На компьютере это можно сделать проще: достаточно проверить, делится ли это число на 2 без остатка. Если да, то число является четным, иначе оно является нечетным.

Вы уже видели, как использовать оператор взятия по модулю (%) для вычисления остатка от деления одного целого числа на другое. Его удобно использовать, чтобы определить, делится ли число на 2 без остатка. Если остаток от деления на 2 равен 0, значит, это четное число, если нет – нечетное.

Напишем программу, которая определяет, является ли целое значение, которое вводит пользователь, четным или нечетным, и выводит соответствующее значение на терминал (см. программу 6.3).

Программа 6.3

```
// Программа, определяющая, является ли число четным или нечетным  
  
#import <Foundation/Foundation.h>  
  
int main (int argc, char *argv[])  
  
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
    int number_to_test, remainder;  
  
    NSLog (@"Enter your number to be tested: ");  
    scanf ("%i", &number_to_test);  
  
    remainder = number_to_test % 2;  
  
    if ( remainder == 0 )
```

```
 NSLog(@"The number is even.");  
  
if ( remainder != 0 )  
    NSLog(@"The number is odd.");  
  
[pool drain];  
return 0;  
}
```

Вывод программы 6.3

Enter your number to be tested: (Ведите ваше число для проверки)

2455

The number is odd. (Число является нечетным)

Вывод программы 6.3 (повторный запуск)

Enter your number to be tested:

1210

The number is even. (Число является четным)

После ввода числа вычисляется остаток от деления на 2. Первый оператор if проверяет, равно ли нулю значение этого остатка. Если да, то выводится сообщение «The number is even.» (Число является четным).

Второй оператор if проверяет значение этого остатка на *неравенство* нулю. Если да, то выводится сообщение, что число является нечетным.

Если выполняется первый оператор if, то второй оператор не должен выполняться, и наоборот. При написании программ концепция «иначе» («else») требуется настолько часто, что почти во всех современных языках программирования введена специальная конструкция для этой ситуации. В Objective-C это конструкция if-else, которая имеет следующий формат.

```
if (выражение)  
    программный оператор 1  
else  
    программный оператор 2
```

На самом деле if-else является просто расширением общего формата оператора if. Если результатом оценки *выражения* является значение TRUE, то выполняется *программный оператор 1*; иначе выполняется *программный оператор 2*. Во всех случаях выполняется *программный оператор 1* или *программный оператор 2*, но не оба оператора.

Оператор if-else можно вставить в предыдущую программу, заменив два оператора if на один оператор if-else. Эта новая конструкция в некоторой степени упрощает программу и делает ее более удобной для чтения.

Программа 6.4

```
// Программа, определяющая, является ли число четным или нечетным (Версия 2)

#import <Foundation/Foundation.h>

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

{

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

    int number_to_test, remainder;

    NSLog (@"Enter your number to be tested:");
    scanf ("%i", &number_to_test);

    remainder = number_to_test % 2;

    if ( remainder == 0 )
        NSLog (@"The number is even.");
    else
        NSLog (@"The number is odd.");

    [pool drain];
    return 0;
}
```

Вывод программы 6.4

```
Enter your number to be tested: (Введите ваше число для проверки)
1234
The number is even. (Число является четным)
```

Вывод программы 6.4 (повторный запуск)

```
Enter your number to be tested:
6551
The number is odd. (Число является нечетным)
```

Не забывайте, что двойной знак равенства (==) – это проверка на равенство, а одинарный знак равенства – это оператор присваивания. Если случайно использовать оператор присваивания внутри оператора if, то может возникнуть множество проблем.

Составные операции сравнения

До сих пор в операторах if использовались простые операции сравнения между двумя числами. В программе 6.1 сравнивалось значение числа с нулем, а в про-

грамме 6.2 выполнялось сравнение с нулем знаменателя дроби. Иногда необходимы более сложные проверки. Предположим, что нужно вычислить количество экзаменационных оценок в диапазоне от 70 по 79. В данном случае нужно сравнить значение оценки не с одним пределом, а с двумя (70 и 79), чтобы проверить, попадает ли она в указанный диапазон.

В Objective-C имеется механизм для составных операторов сравнения этого типа. *Составная операция сравнения (compound relational test)* – это несколько операций сравнения, объединенных логическим оператором AND или OR. Эти операторы представлены парой символов `&&` или `||` (два вертикальных штриха). Например, следующий оператор увеличивает значение переменной `grades_70_to_79` (оценки от 70 по 79), только если значение `grade` (оценка) больше или равно 70 и меньше или равно 79.

```
if ( grade >= 70 && grade <= 79 )
    ++grades_70_to_79;
```

А следующий оператор вызывает выполнение оператора `NSLog`, если `index` меньше 0 или больше 99.

```
if ( index < 0 || index > 99 )
    NSLog(@"%@", @"Error - index out of range"); // Ошибка - индекс вне диапазона
```

Составные операторы можно использовать в Objective-C для формирования очень сложных выражений. Не злоупотребляйте этим. Более простые выражения всегда проще читать и отлаживать. При формировании составных выражений сравнения можно неограниченно использовать круглые скобки, чтобы повысить удобство чтения выражений и избежать осложнений из-за ошибочных предположений о старшинстве операторов в выражении. (Оператор `&&` имеет более низкий приоритет, чем любой арифметический оператор или оператор отношения, но более высокий приоритет, чем оператор `||`.) Пробелы также повышают удобочитаемость выражения. Дополнительные пробелы вокруг операторов `&&` и `||` визуально отделяют эти операторы от выражений, которые они связывают.

Чтобы показать использование составных операций сравнения на конкретном примере, мы напишем программу, которая проверяет, является ли определенный год високосным, если он делится на 4 без остатка. Год, который делится на 100, не является високосным годом, если он не делится также на 400.

Необходимо продумать, как проверять эти условия. Во-первых, можно считать остатки (*remainder*) от деления года на 4, 100 и 400 и присвоить эти значения соответствующим переменным, например, `rem_4`, `rem_100` и `rem_400`. Затем можно проверить эти переменные, чтобы выяснить, отвечают ли эти остатки критериям високосного года.

Можно сказать, что год является високосным, если он делится без остатка на 4 и не делится на 100, либо он делится без остатка на 400. На всякий случай убедитесь, что это определение эквивалентно предыдущему определению. Теперь, когда мы переформулировали определение, его реализация в виде программного оператора становится достаточно простой.

```
if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
    NSLog (@"It's a leap year."); ("Это високосный год")
```

Круглые скобки вокруг следующего подвыражения не обязательны.

```
rem_4 == 0 && rem_100 != 0
```

Оно будет оцениваться в таком порядке в любом случае, поскольку оператор `&&` имеет более высокий приоритет, чем `||`.

В этом примере подходит и следующая проверка.

```
if ( rem_4 == 0 && ( rem_100 != 0 || rem_400 == 0 ) )
```

Можно добавить перед этим оператором `if` несколько операторов объявления переменных и ввод пользователем значения года с терминала. Это программа 6.5, которая определяет, является ли год високосным.

Программа 6.5

```
// Эта программа определяет, является ли год високосным

#import <Foundation/Foundation.h>

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

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

    int year, rem_4, rem_100, rem_400;

    NSLog (@"Enter the year to be tested: ");
    scanf ("%i", &year);

    rem_4 = year % 4;
    rem_100 = year % 100;
    rem_400 = year % 400;

    if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
        NSLog (@"It's a leap year.");
    else
        NSLog (@"Nope, it's not a leap year.");

    [pool drain];
    return 0;
}
```

Вывод программы 6.5

Enter the year to be tested: (Введите год для проверки)

1955

Nope, it's not a leap year. (Нет, это не високосный год)

Вывод программы 6.5 (повторный запуск)

Enter the year to be tested:

2000

It's a leap year. (Это високосный год)

Вывод программы 6.5 (повторный запуск)

Enter the year to be tested:

1800

Nope, it's not a leap year. (Нет, это не високосный год)

В приведенных примерах вводится 1955-й год, который не является високосным, поскольку он не делится нацело на 4, 2000-й год, который является високосным, поскольку он делится нацело на 400, и 1800-й год, который не является високосным, поскольку он делится нацело на 100, но не на 400. В качестве последнего варианта следует проверить год, который делится нацело на 4, но не на 100. Выполните эту проверку самостоятельно.

Objective-C дает программисту очень высокий уровень гибкости при формировании выражений. Например, в приведенной выше программе вместо вычисления промежуточных результатов `rem_4`, `rem_100` и `rem_400` можно было бы сделать все непосредственно в операторе `if`:

```
if ( ( year % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 )
```

Использование пробелов для разделения отдельных операторов делает это выражение более удобным для чтения. Если игнорировать эти возможности и удалить необязательный набор круглых скобок, то получится следующее выражение:

```
if(year%4==0&&year%100!=0)||year%400==0)
```

Это выражение действует точно так же, как предыдущее выражение, однако дополнительные пробелы очень упрощают восприятие составных выражений.

Вложенные операторы if

При рассмотрении общего формата оператора `if` мы указывали, что если результатом оценки выражения в круглых скобках является значение `TRUE`, то выполняется следующий оператор. Этим программным оператором может быть еще один оператор `if`, как в следующих строках.

```
if ( [chessGame isOver] == NO ) ([игра в шахматы окончена] == НЕТ)
    if ( [chessGame whoseTurn] == YOU ) (... чей ход] == ВАШ)
        [chessGame yourMove]; (... ваш ход])
```

Если значение, возвращаемое после отправки сообщения `isOver` объекту `chessGame`, равно `NO`, то выполняется следующий оператор; в свою очередь, этот

оператор является еще одним оператором if. Этот оператор if сравнивает значение, возвращаемое методом `whoseTurn`, со значением YOU. Если эти два значения равны, то объекту `chessGame` отправляется сообщение `yourMove`. Таким образом, сообщение `yourMove` отправляется только в том случае, если игра не окончена, и это ваш ход. Эти операторы можно было бы записать с помощью составной операции сравнения:

```
if ( [chessGame isOver] == NO && [chessGame whoseTurn] == YOU )
    [chessGame yourMove];
```

Обычно с вложенными операторами if используют предложение else, как показано ниже.

```
if ( [chessGame isOver] == NO ) ([игра в шахматы окончена] == НЕТ)
    if ( [chessGame whoseTurn] == YOU ) ([... чей ход] == ВАШ)
        [chessGame yourMove]; ([... ваш ход])
    else
        [chessGame myMove]; ([... мой ход])
```

Сначала все выполняется, как и раньше, но если игра не окончена и ход не ваш, то выполняется предложение else. В результате сообщение `myMove` отправляется объекту `chessGame`. Если игра окончена, то пропускается весь следующий оператор if вместе с присоединенным к нему предложением else.

Предложение else связано с оператором if, который проверяет значение, возвращаемое методом `whoseTurn`, а не с оператором if, который проверяет, окончена ли игра (`chessGame isOver`). Общее правило состоит в том, что предложение else всегда связывается с последним оператором if, не содержащим else.

Можно продвинуться еще на один шаг и добавить предложение else к внешнему оператору if. Это предложение else выполняется, если игра окончена.

```
if ( [chessGame isOver] == NO ) ([игра в шахматы окончена] == НЕТ)
    if ( [chessGame whoseTurn] == YOU ) ([... чей ход] == ВАШ)
        [chessGame yourMove]; ([... ваш ход])
    else
        [chessGame myMove]; ([... мой ход])
else
    [chessGame finish]; (... конец)
```

Конечно, система может интерпретировать этот оператор и по-другому. Например, если в предыдущем примере удалить первое предложение `else`, то оператор не будет интерпретироваться в соответствии с показанными отступами.

```
if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
else
    [chessGame finish];
```

Вместо этого оператор будет интерпретироваться следующим образом:

```
if ( [chessGame isOver] == NO )
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
    else
        [chessGame finish];
```

Предложение `else` связывается с последним оператором `if`, не включающим `else`. Можно использовать фигурные скобки, чтобы связать `else` с внешним `if`, а не с внутренним. Фигурные скобки замыкают оператор `if`, который находится внутри них. Нужная последовательность задается с помощью следующего оператора.

```
if ( [chessGame isOver] == NO ) {
    if ( [chessGame whoseTurn] == YOU )
        [chessGame yourMove];
}
else
    [chessGame finish];
```

Конструкция `else if`

Вы уже видели, как действует оператор `else` при проверке двух возможных условий. Однако решения, которые приходится реализовать в программировании, не всегда ограничиваются выбором между двумя вариантами. Рассмотрим программу, которая выводит `-1`, если пользователь вводит отрицательное число; `0`, если число равно нулю; и `1`, если число больше нуля. (Это реализация функции `sign`.) Очевидно, что в этом случае нужны три проверки: для отрицательных чисел, чисел, равных нулю, и положительных чисел. Простая конструкция `if-else` в данном случае не подходит. Конечно, мы могли бы прибегнуть здесь к трем отдельным операторам `if`, но это решение не всегда можно реализовать, особенно если проверки не являются взаимоисключающими.

Чтобы справиться с этой ситуацией, можно добавить оператор `if` к предложению `else`. Мы уже говорили, что после предложения `else` может следовать любой допустимый программный оператор Objective-C, так почему не использовать еще один `if`?

```
if (выражение 1)
    программный оператор 1
else
    if (выражение 2)
        программный оператор 2
    else
        программный оператор 3
```

В результате оператор `if` расширяется — вместо логического решения с двумя значениями мы получаем логическое решение с тремя значениями. Вы можете продолжить добавление операторов `if` к предложениям `else` в том же стиле, чтобы получить логическое решение с `n` значениями.

Эта конструкция так часто используется, что обычно ее называют конструкцией `else if` и форматируют так.

```
if (выражение 1)
    программный оператор 1
else if (выражение 2)
    программный оператор 2
else
    программный оператор 3
```

Этот способ делает чтение оператора более удобным. В следующей программе показано использование конструкции `else if` с реализацией описанной выше функции `sign`.

Программа 6.6

```
// Реализация функции sign

#import <Foundation/Foundation.h>

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

    int number, sign;

    NSLog (@"Please type in a number: ");
    scanf ("%i", &number);

    if ( number < 0 )
        sign = -1;
    else if ( number == 0 )
        sign = 0;
    else // Положительное число
        sign = 1;

    NSLog (@"Sign = %i", sign);
    [pool drain];
    return 0;
}
```

Вывод программы 6.6

```
Please type in a number: (Введите число)
1121
Sign = 1
```

Вывод программы 6.6 (повторный запуск)

Please type in a number:

-158

Sign = -1

Вывод программы 6.6 (повторный запуск)

Please type in a number:

0

Sign = 0

Если введенное число меньше нуля, переменной sign присваивается значение -1; если число равно нулю, sign присваивается значение 0; в противном случае число больше нуля, поэтому sign присваивается значение 1.

В следующей программе символ, введенный с терминала, анализируется и классифицируется как алфавитный символ (a-z или A-Z), как цифровой символ (0-9) или как специальный символ (все остальные символы). Для чтения одного символа с терминала при вызове scanf используются символы форматирования %c.

Программа 6.7

```
// Эта программа классифицирует символ,  
// введенный с клавиатуры
```

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

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

    NSLog (@"Enter a single character:");
    scanf ("%c", &c);

    if ( (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') )
        NSLog (@"It's an alphabetic character.");
    else if ( c >= '0' && c <= '9' )
        NSLog (@"It's a digit.");
    else
        NSLog (@"It's a special character.");

    [pool drain];
    return 0;
}
```

Вывод программы 6.7

Enter a single character: (Введите один символ)
&
It's a special character. (Это специальный символ)

Вывод программы 6.7 (повторный запуск)

Enter a single character:
8
It's a digit. (Это цифра)

Вывод программы 6.7 (повторный запуск)

Enter a single character:
B
It's an alphabetic character. (Это алфавитный символ)

В первой проверке после чтения символа мы определяем, является ли этот символ алфавитным (в переменной с типа `char`). Сначала проверяется, попадает ли этот символ в диапазон строчных букв:

(`c >= 'a' && c <= 'z'`)

Это выражение имеет значение `TRUE`, если с находится в диапазоне символов от '`a`' до '`z`' (с – строчная латинская буква). Вторая проверка выполняется с помощью следующего выражения:

(`c >= 'A' && c <= "Z"`)

Это выражение имеет значение `TRUE`, если с находится в диапазоне символов от '`A`' до '`Z`' (с – прописная латинская буква). Эти проверки подходят для символов в кодировке ASCII.

Если переменная с содержит алфавитный символ, проверка проходит успешно и выводится сообщение «`It's an alphabetic character`» (Это алфавитный символ). В противном случае выполняется предложение `else if`. Оно определяет, является ли данный символ цифрой. Отметим, что выполняется сравнение с *символами* от '`0`' до '`9`', а *не с цифрами* от `0` до `9`. Дело в том, что с терминала читается символ, а запись символов от '`0`' до '`9`' в компьютере отличается от чисел `0-9`. В кодировке ASCII символ '`0`' представляется во внутренней записи числом `48`, символ '`1`' – числом `49`, и т.д.

Если с – символ цифры, то выводится фраза «`It's a digit`» (Это цифра). В противном случае (если с не является алфавитным символом или символом цифры) выполняется последнее предложение `else`, и на терминале выводится фраза «`It's a special character`» (Это специальный символ). Затем выполнение программы завершается.

Хотя в данном случае `scanf` используется для чтения только одного символа, вы все равно должны нажать клавишу `Enter` после ввода символа, чтобы результат ввода был передан в программу. Обычно при чтении данных с терминала программа «не видит» введенных данных, пока не нажата клавиша `Enter`.

В следующем примере напишем программу, которая позволяет ввести простое выражение в следующей форме:

число оператор число

Программа выполнит оценку выражения и выведет результаты на терминал. Операторы, которые нужно распознавать, – это обычные оператор сложения, вычитания, умножения и деления. Мы будем использовать класс Calculator из программы 4.6 главы 4. Каждое выражение будет передаваться этому калькулятору для вычислений.

В следующей программе используется довольно большой оператор if с несколькими предложениями else if, определяющими, какая операция должна быть выполнена.

Примечание. Чтобы избежать проблем с внутренним представлением, лучше использовать процедуры из стандартной библиотеки `islower` и `isupper`. Для этого включите в программу строку `#import <ctype.h>`.

Программа 6.8

```
// Программа вычисления простых выражений в форме
//   число оператор число

// Реализация класса Calculator

#import <Foundation/Foundation.h>

@interface Calculator: NSObject
{
    double accumulator;
}

// методы для сумматора (accumulator)
-(void) setAccumulator: (double) value;
-(void) clear;
-(double) accumulator;

// арифметические методы
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
-(void) setAccumulator: (double) value
```

```
{  
    accumulator = value;  
}  
  
-(void) clear  
{  
    accumulator = 0;  
}  
  
-(double) accumulator  
{  
    return accumulator;  
}  
  
-(void) add: (double) value  
{  
    accumulator += value;  
}  
  
-(void) subtract: (double) value  
{  
    accumulator -= value;  
}  
  
-(void) multiply: (double) value  
{  
    accumulator *= value;  
}  
  
-(void) divide: (double) value  
{  
    accumulator /= value;  
}  
@end  
  
int main (int argc, char *argv[])  
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
    double value1, value2;  
    char operator;  
    Calculator *deskCalc = [[Calculator alloc] init];  
  
    NSLog (@"Type in your expression.");  
    scanf ("%lf %c %lf", &value1, &operator, &value2);  
}
```

```
[deskCalc setAccumulator: value1];
if ( operator == '+' )
    [deskCalc add: value2];
else if ( operator == '-' )
    [deskCalc subtract: value2];
else if ( operator == '*' )
    [deskCalc multiply: value2];
else if ( operator == '/' )
    [deskCalc divide: value2];

NSLog (@"%.2f", [deskCalc accumulator]);
[deskCalc release];

[pool drain];
return 0;
}
```

Вывод программы 6.8

Type in your expression. (Введите ваше выражение)
123.5 + 59.3
182.80

Вывод программы 6.8 (повторный запуск)

Type in your expression.
198.7 / 26
7.64

Вывод программы 6.8 (повторный запуск)

Type in your expression.
89.3 * 2.5
223.25

При обращении к `scanf` указывается, что должны быть считаны три значения в переменные `value1`, `operator` и `value2`. Значение типа `double` можно прочитать с помощью символов формата `%lf`. Этот формат считывает значения в переменную `value1`, которая является первым операндом выражения.

Затем считывается оператор. Поскольку оператор представляется символом ('+', '-', '*' или '/'), а не числом, он читается в символьную переменную `operator`. Символы формата `%c` указывают, что нужно прочитать с терминала следующий символ. Пробелы внутри строки формата указывают, что при вводе допускается любое число пробелов. Это позволяет при вводе значений отделять операнды от оператора.

После считывания двух значений и оператора программа сохраняет первое значение в сумматоре (accumulator) калькулятора. Затем значение `operator` сравнивается с четырьмя возможными операторами. Если найдено допустимое со-

ответствии, калькулятору передается сообщение для выполнения операции. В последней строке `NSLog` значение `accumulator` считывается для вывода. После этого выполнение программы завершается.

Здесь необходимо сказать несколько слов о законченности программы. Хотя приведенная программа выполняет поставленную задачу, ее нельзя назвать законченной, поскольку в ней не учитываются ошибки пользователя. Например, что произойдет, если пользователь ошибся и ввел `?` в качестве имени оператора? Программа просто пройдет через оператор `if`, и на терминале не появится никаких ошибок, уведомляющих пользователя, что он ввел недопустимое выражение.

Еще один неучтенный случай – это ввод пользователем нулевого значения делителя в операции деления. Программа должна проверять этот случай. Определение ситуаций, при которых возможен отказ программы или получение нежелательных результатов, и принятие предупредительных мер – обязательная часть разработки надежных программ.

Программа 6.8А является модифицированной версией программы 6.8. В ней учитываются деление на нуль и ввод неизвестного оператора.

Программа 6.8А

```
// Программа вычисления простых выражений в форме
//   число оператор число

#import <Foundation/Foundation.h>

// Здесь нужно вставить секции interface и implementation
// для класса Calculator

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    double      value1, value2;
    char        operator;
    Calculator *deskCalc = [[Calculator alloc] init];

    NSLog(@"Type in your expression.");
    scanf ("%lf %c %lf", &value1, &operator, &value2);

    [deskCalc setAccumulator: value1];

    if (operator == '+')
        [deskCalc add: value2];
    else if (operator == '-')
        [deskCalc subtract: value2];
```

```
else if ( operator == '*' )
    [deskCalc multiply: value2];
else if ( operator == '/' )
    if ( value2 == 0 )
        NSLog (@"Division by zero.");
    else
        [deskCalc divide: value2];
else
    NSLog (@"Unknown operator.");

NSLog (@"%.2f", [deskCalc accumulator]);
[deskCalc release];

[pool drain];
return 0;
}
```

Вывод программы 6.8А

Type in your expression. (Введите ваше выражение)
123.5 + 59.3
182.80

Вывод программы 6.8А (повторный запуск)

Type in your expression.
198.7 / 0
Division by zero. (Деление на нуль)
198.7

Вывод программы 6.8А (повторный запуск)

Type in your expression.
125 \$ 28
Unknown operator. (Неизвестный оператор)
125

Если введен оператор деления (слэш), то проверяется, не равно ли нулю значение value2. Если равно, выводится соответствующее сообщение; если нет — выполняется операция деления и выводятся результаты. Обратите внимание, как в данном случае вложены оператор if и предложения else.

Предложение else в конце программы позволяет «ловить» любые непредусмотренные случаи. Поэтому любое значение переменной operator, которое не совпадает с одним из четырех символов вычислений, приводит к выполнению предложения else, и на терминал выводится сообщение «Unknown operator» (Неизвестный оператор).

Более удобный способ обработки ситуации деления на ноль — это выполнение проверки в методе, который работает с операцией деления. Метод `divide:` можно модифицировать следующим образом.

```
- (void) divide: (double) value
{
    if (value != 0.0)
        accumulator /= value;
    else {
        NSLog (@"Division by zero.");
        accumulator = 99999999.;
    }
}
```

Для ненулевого значения выполняется деление; для нулевого — выводится сообщение, и переменной `accumulator` присваивается значение 99999999. Это произвольно выбранное значение; можно было присвоить значение 0 или задать специальную переменную, указывающую состояние ошибки. Лучше включать обработку особых ситуаций в сам метод, чем полагаться на предусмотрительность программиста при использовании метода.

Оператор `switch`

Цепочки `if-else` так часто используются при разработке программ, что в язык Objective-C включен специальный оператор для выполнения именно этой функции. Это оператор `switch`. Он имеет следующий формат.

```
switch (выражение)
{
    case value1:
        программный оператор
        программный оператор
        ...
        break;
    case value2:
        программный оператор
        программный оператор
        ...
        break;
    ...
    case valuen:
        программный оператор
        программный оператор
        ...
        break;
    default:
```

```
    программный оператор  
    программный оператор  
    ...  
    break;  
}
```

Выражение, указанное в круглых скобках, последовательно сравнивается со значениями *value1*, *value2*, ..., *valuen*, которые могут быть простыми константами или константными выражениями. Если найден вариант (case), при котором значение равно значению выражения, выполняются программные операторы, следующие после этого предложения case. Если включено несколько таких операторов, то их можно не заключать в фигурные скобки.

Оператор *break* вызывает прекращение выполнения оператора *switch*. Не забывайте включать оператор *break* в конце каждого предложения *case*, иначе выполнение программы продолжится в следующем предложении *case*. Иногда это делают преднамеренно; в таких случаях обязательно включите комментарии, чтобы предупредить других о ваших целях.

Особый (необязательный) вариант с именем *default* выполняется в том случае, если значение выражения не соответствует ни одному из значений, указанных в предложениях *case*. Это эквивалентно предложению *else*, которое использовалось в предыдущем примере для «улавливания» любых неуказанных вариантов. На самом деле общую форму оператора *switch* можно представить в эквивалентной форме оператора *if*.

```
if (выражение == value1)  
{  
    программный оператор  
    программный оператор  
    ...  
}  
else if (выражение == value2)  
{  
    программный оператор  
    программный оператор  
    ...  
}  
...  
else if (выражение == valuen)  
{  
    программный оператор  
    программный оператор  
    ...  
}  
else  
{
```

программный оператор
программный оператор

...

}

Мы можем теперь преобразовать большой оператор if из программы 6.8A в эквивалентный оператор switch. Это показано в программе 6.9.

Программа 6.9

```
// Программа вычисления простых выражений в форме
//   число оператор число

#import <Foundation/Foundation.h>

// Здесь нужно вставить секции interface и implementation
// для класса Calculator

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

    double value1, value2;
    char operator;
    Calculator *deskCalc = [[Calculator alloc] init];

    NSLog (@"Type in your expression.");
    scanf ("%lf %c %lf", &value1, &operator, &value2);

    [deskCalc setAccumulator: value1];

    switch ( operator ) {
        case '+':
            [deskCalc add: value2];
            break;
        case '-':
            [deskCalc subtract: value2];
            break;
        case '*':
            [deskCalc multiply: value2];
            break;
        case '/':
            [deskCalc divide: value2];
            break;
        default:
            NSLog (@"Unknown operator.");
    }
}
```

```
        break;
    }

    NSLog(@"%@", [deskCalc accumulator]);
    [deskCalc release];

    [pool drain];
    return 0;
}
```

Вывод программы 6.9

Type in your expression. (Введите ваше выражение)
178.99 - 326.8
-147.81

После считывания выражения значение переменной `operator` сравнивается со значениями каждого варианта `case`. Если найдено соответствие, то выполняются операторы внутри этого варианта `case`. Затем оператор `break` выполняет выход из оператора `switch`, после чего завершается выполнение программы. Если ни один из вариантов не соответствует значению переменной `operator`, то используется вариант по умолчанию `default`, в котором выводится сообщение «`Unknown operator`» (Неизвестный оператор).

Оператор `break` для варианта `default` не является необходимым, т.к. после него в данном операторе `switch` не выполняется никаких операторов. Однако практика надежного программирования требует вставки `break` в конце каждого варианта.

При написании оператора `switch` помните, что нельзя указывать одинаковые значения для двух разных предложений `case`. Однако один набор программных операторов можно связывать с несколькими `case`-значениями. Для этого нужно задать список из нескольких `case`-значений (с ключевым словом `case` перед каждым значением и двоеточием после значения) и общий набор программных операторов. Например, в следующем операторе `switch` выполняется метод `multiply:`, если значение `operator` указано символом «звездочка» или строчной буквой `x`.

```
switch ( operator )
{
    ...
    case '*':
    case 'x':
        [deskCalc multiply: value2];
        break;
    ...
}
```

Булевые переменные

Почти все, кто учится программировать, довольно скоро сталкиваются с задачей написания программы, которая создает таблицу простых чисел. Напомним, что положительное целое число p является простым числом, если оно не делится нацело ни на одно из других целых чисел, кроме 1 и самого себя. Первым простым числом является 2. Следующее простое число – 3, так как оно делится нацело только на 1 и 3; 4 не является простым числом, так как оно делится нацело на 2.

Создать таблицу простых чисел можно несколькими способами. Например, чтобы создать таблицу простых чисел, не превышающих 50, достаточно напрямую проверить делимость каждого целого числа p на все целые числа от 2 до $p-1$. Если p делится без остатка на одно из этих чисел, значит, p не является простым числом, иначе оно является простым.

В программе 6.10 создается список простых чисел, не превышающих 50.

Программа 6.10

```
// Программа создания таблицы простых чисел (prime number)

#import <Foundation/Foundation.h>

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

    int p, d, isPrime;

    for ( p = 2; p <= 50; ++p ) {
        isPrime = 1;

        for ( d = 2; d < p; ++d )
            if ( p % d == 0 )
                isPrime = 0;

        if ( isPrime != 0 )
            NSLog(@"%@", p);
    }

    [pool drain];
    return 0;
}
```

Вывод программы 6.10

2
3

5
7
11
13
17
19
23
29
31
37
41
43
47

Внешний оператор `for` задает цикл по целым значениям от 2 до 50. Переменная цикла `r` представляет текущее значение, для которого выполняется проверка, является ли оно простым числом. В первом операторе цикла присваивается значение 1 переменной `isPrime`. Использование этой переменной будет описано чуть позже.

Во втором цикле выполняется деление `r` на целые числа от 2 до `r-1`. Внутри этого цикла проверяется остаток от деления `r` на `d`. Если остаток равен 0, значит, `r` не является простым числом, поскольку оно делится на целое число, отличное от 1 и от него самого. Чтобы указать, что `r` уже не является «кандидатом» на простое число, переменной `isPrime` присваивается значение 0.

После окончания внутреннего цикла проверяется значение `isPrime`. Если оно не равняется нулю, значит, не найдено целого числа, которое делит `r` нацело, и `r` является простым числом, после чего выводится его значение.

Переменная `isPrime` принимает всего два значения: 0 или 1. Ее значение равно 1, пока еще проверяется на принадлежность к множеству простых чисел. Как только находится хотя бы один делитель без остатка, `isPrime` присваивается значение 0, чтобы указать, что `r` не отвечает критерию принадлежности к простым числам. Такие переменные обычно называют *булевыми* (*Boolean*) переменными. Флажок (признак) обычно принимает одно из двух значений. Кроме того, значение флаг-ка обычно проверяется хотя бы один раз в программе, чтобы определить, установлен он (значение `TRUE` или `YES`) или сброшен (`FALSE` или `NO`), а также выполнить определенное действие в зависимости от результата проверки.

В Objective-C состояниям флагка `TRUE` и `FALSE` естественным образом соответствуют значения 1 и 0. Например, когда в программе 6.10 переменной `isPrime` внутри цикла присваивается значение 1, ей фактически присваивается значение `TRUE`, указывающее, что `r` пока считается простым числом («`is prime`»). Если при выполнении внутреннего цикла `for` найден делитель нацело, переменной `isPrime` фактически присваивается значение `FALSE`, чтобы указать, что `r` не является простым числом.

Значение 1 обычно используется для представления значения `TRUE` (или состояния «`on`»), а значение 0 – для представления значения `FALSE` (или состояния

«off»). Это соответствует одному биту в компьютере. Если бит находится в состоянии «on» (установлен), его значение равно 1, если он находится в состоянии «off» (сброшен), его значение равно 0. Но в Objective-C имеется более веская причина для использования именно этих логических значений. В начале этой главы говорилось, что если удовлетворяются условия, указанные в операторе `if`, то выполняется следующий за ним программный оператор. В языке Objective-C это означает условие (состояние) «не ноль». Например,

```
if ( 100 )
    NSLog(@"%@", @"This will always be printed.");
```

вызывает выполнение оператора `NSLog` ввиду ненулевого состояния в операторе `if` (в данном случае – значение 100), то есть удовлетворяется условие «не ноль».

В каждой из программ этой главы мы фактически использовали понятия «не нуль – условие удовлетворяется» и «нуль – условие не удовлетворяется». В Objective-C при оценке выражения отношения ему присваивается значение 1, если соответствующее условие удовлетворяется, и 0, если условие не удовлетворяется. Поэтому оценка оператора

```
if ( number < 0 )
    number = -number;
```

фактически происходит следующим образом. Выполняется оценка выражения отношения `number < 0`. Если это условие удовлетворяется (`number` меньше 0), то значение выражения равно 1, иначе его значение равно 0.

В операторе `if` проверяется результат оценки выражения. В случае ненулевого результата выполняется следующий оператор. То же самое относится к оценке условий в операторах `for`, `while` и `do`. Оценка составных выражений отношения (сравнения) выполняется таким же образом.

```
while ( char != 'e' && count != 80 )
```

Если верны оба условия, то результат равен 1, но если неверно хотя бы одно из условий, результат оценки равен 0. Затем выполняется проверка результатов оценки. Если результат равен 0, то выполнение цикла `while` прекращается, иначе его выполнение продолжается.

Вернемся к программе 6.10 и понятию флагков. В Objective-C допустимо проверять значение флагка `TRUE`, например, с помощью следующего выражения.

```
if ( isPrime )
```

Оно эквивалентно следующему выражению.

```
if ( isPrime != 0 )
```

Можно проверить значение флагка `FALSE` с помощью логического оператора отрицания `!`. В следующем выражении логический оператор отрицания используется, чтобы проверить переменную `isPrime` на значение `FALSE` (его можно прочитать как «если не `isPrime`»).

```
if ( ! isPrime )
```

В общем виде логическое отрицание *выражения* записывается как
! выражение

Так, если *выражение* равно 0, то оператор логического отрицания дает значение 1. Если оценка выражения не равна 0, то оператор отрицания дает значение 0.

Оператор логического отрицания можно использовать для изменения значения флагка, как в следующем выражении.

```
my_move = ! my_move;
```

Этот оператор имеет такой же уровень приоритета, как и унарный оператор «минус», т.е. имеет более высокий приоритет, чем все бинарные арифметические операторы и все операторы отношения. В следующем выражении задано условие на то, чтобы значение переменной *x* было не меньше значения переменной *y*, поэтому здесь необходимы круглые скобки.

```
!( x < y )
```

Этот оператор можно записать в эквивалентной форме:

```
x >= y
```

В Objective-C имеется пара встроенных средств, упрощающих работу с булевыми переменными. Одно из них – специальный тип **BOOL**, который можно использовать для объявления переменных со значениями **true** или **false**. Второе средство – это встроенные значения **YES** и **NO**. Использование этих значений упрощает написание и чтение программ. Ниже приводится версия программы 6.10, переписанной с использованием этих средств.

Примечание. Тип **BOOL** добавляется механизмом, который называется препроцессором.

Программа 6.10A

```
// Программа создания таблицы простых чисел.  
// Вторая версия с использованием типа BOOL и готовых значений  
  
#import <Foundation/Foundation.h>  
  
int main (int argc, char *argv[])  
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
  
    int      p, d;  
    BOOL    isPrime;
```

```
for ( p = 2; p <= 50; ++p ) {
    isPrime = YES;

    for ( d = 2; d < p; ++d )
        if ( p % d == 0 )
            isPrime = NO;

    if ( isPrime == YES )
        NSLog (@»%i «, p);
}

[pool drain];
return 0;
}
```

Вывод программы 6.10A

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

Условный оператор

Возможно, самым необычным оператором в языке Objective-C является *условный* (conditional) оператор. В отличие от остальных операторов в Objective-C, которые являются унарными или бинарными, условный оператор является *тернарным, или трехместным* (*ternary*), то есть в нем требуются три операнда. В этом операторе для обозначений используются два символа: вопросительный знак (?) и двоеточие (:). Первый operand ставят перед знаком ?, второй operand – между знаками ? и :, третий operand ставится после :. Ниже показан формат условного оператора в общем виде.

условие ? *выражение_1* : *выражение_2*

В этом описании синтаксиса *условие* – это выражение (обычно выражение отношения), которое сначала оценивается системой Objective-C в условном операторе. Если результатом оценки является значение TRUE (то есть не нуль), то вычисляется *выражение_1* и результат вычисления становится результатом данной операции. Если результатом оценки *условия* является значение FALSE (то есть нуль), то вычисляется *выражение_2* и результат вычисления становится результатом данной операции.

Условное выражение чаще всего используется, чтобы в зависимости от некоторого условия присвоить переменной одно из двух значений. Например, предположим, что у нас есть две целые переменные - *x* и *s*. Чтобы присвоить *s* значение -1, если *x* меньше 0, или значение x^2 в противном случае, можно использовать следующий оператор.

```
s = ( x < 0 ) ? -1 : x * x;
```

Сначала проверяется условие $x < 0$. Обычно условие помещается в круглые скобки, но это не обязательно, поскольку условный оператор имеет очень низкий приоритет – ниже всех остальных операторов, за исключением операторов присваивания и оператора «запятая».

Если значение *x* меньше нуля, то вычисляется выражение, непосредственно следующее за символом ?. Это выражение представлено просто целой константой -1, которая присваивается переменной *s*, если *x* меньше нуля.

Если значение *x* не меньше нуля, то вычисляется выражение, непосредственно следующее за символом :, оно присваивается переменной *s*. Таким образом, если *x* больше или равно нулю, то переменной *s* присваивается значение $x * x$, или x^2 .

Еще один пример условного оператора: оператор, где переменной *max_value* присваивается максимум из двух значений: *a* и *b*.

```
max_value = ( a > b ) ? a : b;
```

Если выражение после : (часть «иначе») содержит еще один условный оператор, то мы получаем возможности предложения else if. Например, функцию *sign*, которая реализована в программе 6.6, можно записать в одной программной строке с помощью двух условных операторов.

```
sign = ( number < 0 ) ? -1 : (( number == 0 ) ? 0 : 1);
```

Если *number* меньше нуля, то переменной *sign* присваивается значение -1; если *number* равно нулю, то переменной *sign* присваивается значение 0; иначе ей присваивается значение 1. Круглые скобки вокруг части «иначе» на самом деле не обязательны. Дело в том, что условный оператор объединяется справа налево, то есть когда этот оператор используется несколько раз в одном выражении, например, в

```
e1 ? e2 : e3 ? e4 : e5
```

группирование происходит справа налево и поэтому оценка выполняется следующим образом:

`e1 ? e2 : (e3 ? e4 : e5)`

Условные выражения не обязаны стоять в правой части операции присваивания — их можно применять в любой ситуации, где используются выражения. С помощью оператора `NSLog` вы можете вывести на дисплей знак (`sign`) переменной `number` без присваивания ее какой-либо переменной:

```
NSLog(@"Sign = %i", ( number < 0 ) ? -1 : ( number == 0 ) ? 0 : 1);
```

Условный оператор очень удобно использовать в Objective-C при написании макросов препроцессора (см. главу 12).

Упражнения

- Напишите программу, которая запрашивает у пользователя ввод двух целых значений. Проверьте, делится ли первое число без остатка на второе число, и выведите на терминал соответствующее сообщение.
- Программа 6.8A выводит значение, содержащееся в сумматоре, даже если введен неверный оператор или пользователь пытается выполнить деление на нуль. Решите эту проблему.
- Внесите изменения в метод `print` из класса `Fraction`, чтобы целые числа выводились в обычном виде (например, дробь `5/1` должна выводиться как `5`). Внесите изменения в этот метод, чтобы дроби с делителем, равным `0`, выводились просто как `0`.
- Напишите программу, которая действует как простой калькулятор. Эта программа должна позволять пользователю вводить выражения в следующей форме:

число оператор

Программа должна распознавать следующие операторы:

`+ - * / S E`

Оператор `S` указывает, что программа должна присвоить сумматору (`accumulator`) введенное число, и оператор `E` указывает программе, что нужно закончить выполнение. Арифметические операции выполняются для содержимого сумматора с числом, которое введено для операции как второй operand. Ниже приводится пример, показывающий, как должна работать эта программа.

Начало вычислений

<code>10 S</code>	<i>Сумматору (Accumulator) присваивается значение 10</i>
<code>= 10.000000</code>	<i>Содержимое сумматора</i>
<code>2 /</code>	<i>Divide by 2</i>
<code>= 5.000000</code>	<i>Содержимое сумматора</i>
<code>55 -</code>	<i>Вычитание 55</i>
<code>= -50.000000</code>	
<code>100.25 S</code>	<i>Сумматору присваивается значение 100.25</i>
<code>= 100.250000</code>	

```
4 *          Умножение на 4  
= 401.000000  
0 E          Завершение программы  
= 401.000000  
End of Calculations. (Конец вычислений)
```

Сделайте так, чтобы программа обнаруживала деление на 0, а также выполняла проверку на неизвестные операторы. Используйте для своих вычислений класс `Calculator` из программы 6.8

5. В программе 5.9 выполняется вывод цифр введенного числа в обратном порядке. Однако эта программа не работает должным образом, если введено отрицательное число. Определите, что происходит в этом случае, и затем внесите в программу изменения, чтобы правильно обрабатывались отрицательные числа. Это означает, например, что если введено число -8645, то программа должна вывести 5468-.
6. Напишите программу, которая читает число, введенное с терминала, и выводит каждую цифру этого числа буквами. Например, если пользователь ввел 932, то программа должна вывести следующее.

девять
три
два

(Выведите слово **нуль**, если пользователь ввел 0.) *Примечание.* Это довольно трудное упражнение!

7. Программа 6.10 в ряде случаев работает неэффективно. Например, это касается проверки четных чисел. Поскольку любое четное число больше 2 не может быть простым, программа могла бы пропускать все четные числа и как возможные простые числа, и как возможные делители. Внутренний цикл `for` тоже работает неэффективно, поскольку для `p` все время выполняется деление на все значения `d` от 2 до `p-1`. Чтобы избежать этой неэффективности, можно добавить значение `isPrime` в условия `for`. Тогда вы можете задать выполнение цикла `for` до тех пор, пока не найден делитель и значение `d` меньше `p`. Модифицируйте программу 6.10, чтобы внести эти два изменения, затем выполните эту программу, чтобы проверить ее работу.

Глава 7

Более подробно о классах

В этой главе мы продолжим изучение классов и методов и научимся работать с программными циклами, выражениями и операторами выбора, которые изучили в предыдущей главе. Начнем с разбиения программы на несколько файлов, которое упрощает работу с большими программами.

Разделение файлов объявлений и определений (секции `interface` и `implementation`)

Объявления классов и определения классов для сложных программ принято размещать в отдельных файлах.

Если вы используете Xcode, начните с нового проекта с именем `FractionTest`. Введите следующую программу в файл `FractionTest.m`.

Программа 7.1 - часть main: `FractionTest.m`

```
#import "Fraction.h"

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

    // задание дроби (fraction) 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // вывод дроби

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
```

```
    return 0;
}
```

В этот файл не включено определение класса Fraction, имеется импорт файла Fraction.h.

Обычно объявление класса (то есть секцию @interface) помещают в отдельный файл с именем *класс.h*. Определение класса (то есть секцию @implementation) помещают обычно в файл с тем же именем, но с расширением *.m*. Мы помещаем объявление класса Fraction в файл Fraction.h, а определение этого класса — в файл Fraction.m.

В Xcode выберите New File (Новый файл) в меню File (Файл). В левой панели выберите Cocoa. В верхней правой панели выберите Objective-C class (рис. 7.1).

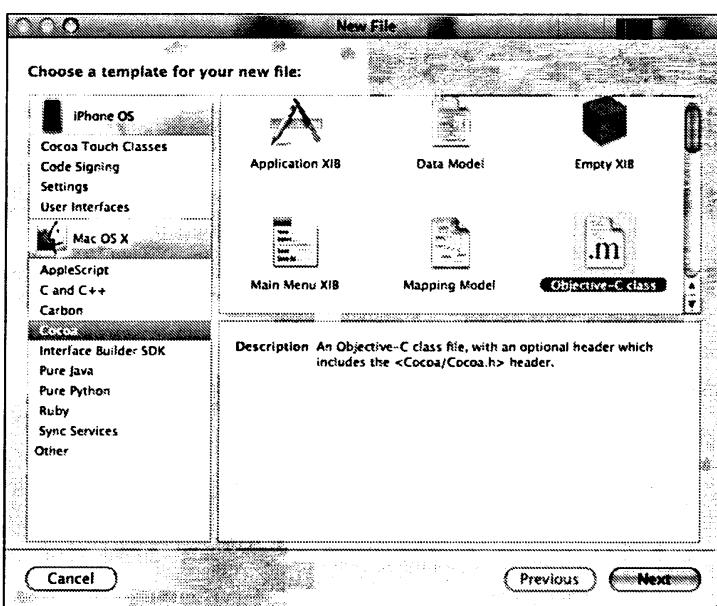


Рис. 7.1. Меню New File в Xcode

Щелкните на кнопке Next (Далее). Введите Fraction.m в поле имени файла (File name). Оставьте установленным флажок Also Create Fraction.h (Создать также Fraction.h). Местоположением (Location) этого файла должна быть папка с файлом FractionTest.m (рис. 7.2).

Теперь щелкните на кнопке Finish (Готово). Xcode добавит в ваш проект два файла: Fraction.h и Fraction.m (рис. 7.3).

Мы не работаем здесь с Cocoa, поэтому заменим в файле Fraction.h строку

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

на строку

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

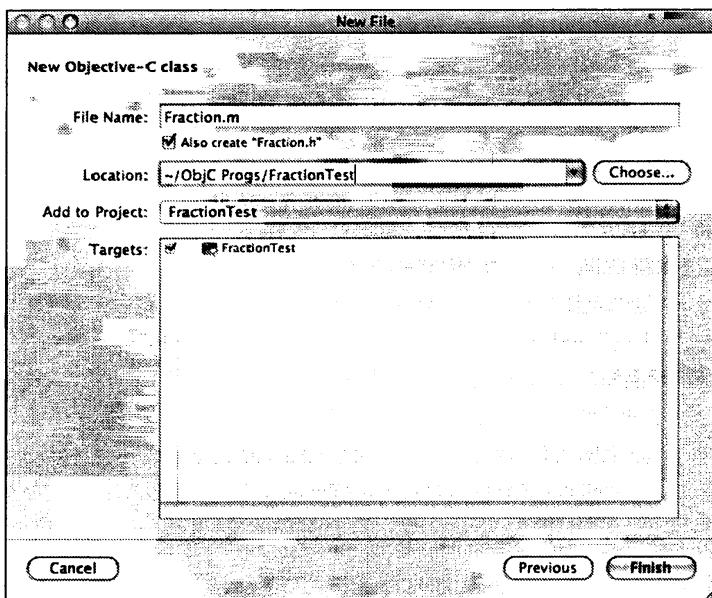


Рис. 7.2. Добавление нового класса к проекту

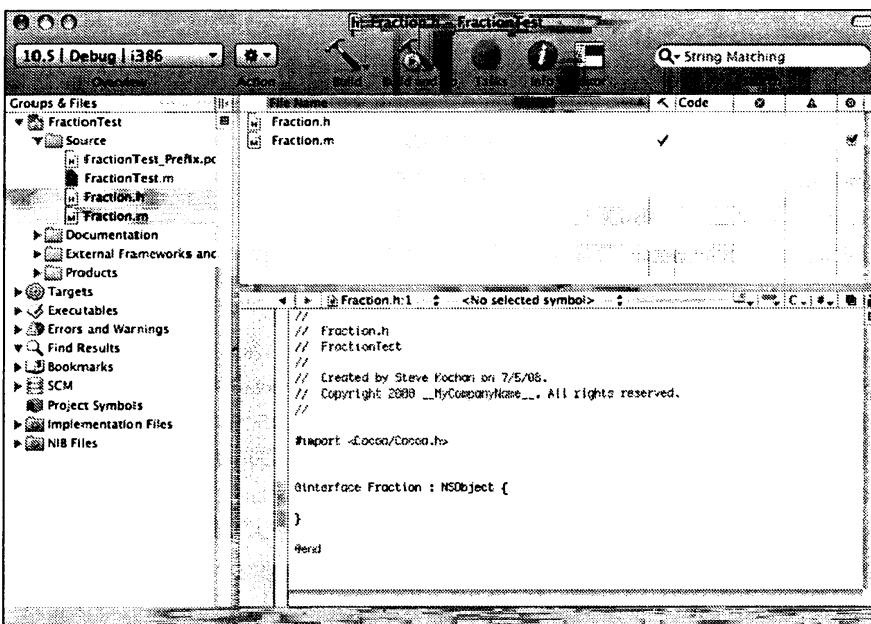


Рис. 7.3. Xcode создает файлы для нового класса

В этом файле (Fraction.h) введем секцию interface для класса Fraction.

Программа 7.1 - файл секции interface - Fraction.h

```

//  

//  Fraction.h  

//  FractionTest  

//  

//  Created by Steve Kochan on 7/5/08.  

//  Copyright 2008 _MyCompanyName_. All rights reserved.  

//  

#import <Foundation/Foundation.h>  

// Класс Fraction  

@interface Fraction : NSObject  

{  

    int numerator;  

    int denominator;  

}  

-(void)    print;  

-(void)    setNumerator: (int) n;  

-(void)    setDenominator: (int) d;  

-(int)     numerator;  

-(int)     denominator;  

-(double)   convertToNum;  

@end

```

Этот файл указывает компилятору (и программистам, см. ниже), как должен выглядеть объект класса `Fraction`: он содержит две целых переменных экземпляра с именами `numerator` (числитель) и `denominator` (знаменатель). Он также содержит шесть методов экземпляра: `print`, `setNumerator:`, `setDenominator:`, `numerator`, `denominator` и `convertToNum`. Первые три метода не возвращают никакого значения, следующие два возвращают значение типа `int`, а последний метод возвращает значение типа `double`. Методам `setNumerator:` и `setDenominator:` передается аргумент целого типа.

Детали определения (*implementation*) класса `Fraction` содержатся в файле `Fraction.m`.

Программа 7.1 файл секции implementation - Fraction.m

```

//  

//  Fraction.m  

//  FractionTest  

//  

//  Created by Steve Kochan on 7/5/08.  

//  Copyright 2008 _MyCompanyName_. All rights reserved.

```

```
//  
  
#import "Fraction.h"  
  
@implementation Fraction  
-(void) print  
{  
    NSLog(@"%@", numerator, denominator);  
}  
  
-(void) setNumerator: (int) n  
{  
    numerator = n;  
}  
  
-(void) setDenominator: (int) d  
{  
    denominator = d;  
}  
  
-(int) numerator  
{  
    return numerator;  
}  
  
-(int) denominator  
{  
    return denominator;  
}  
  
-(double) convertToNum  
{  
    if (denominator != 0)  
        return (double) numerator / denominator;  
    else  
        return 1.0;  
}  
@end
```

Обратите внимание, что файл секции `interface` импортируется в файл секции `implementation` с помощью следующего оператора:

```
#import "Fraction.h"
```

Это позволяет компилятору получать информацию о классах и методах, объявленных для класса `Fraction`, и обеспечивать согласованность между этими

файлами. Переменные экземпляра внутри класса обычно не переобъявляют внутри секции `implementation` (хотя это можно делать), чтобы компилятор брал эту информацию из секции `interface`, содержащейся в `Fraction.h`.

Импортируемый файл заключается в кавычки, а не в угловые скобки (< и >), как `<Foundation/Foundation.h>`. Кавычки используются для локальных файлов (создаваемых вами вместо системных файлов), и они показывают компилятору, где искать указанный файл. Когда используются кавычки, компилятор обычно ищет сначала указанный файл в текущей папке, а затем – в списке других мест. Вы можете указывать для компилятора конкретные места поиска.

Пример тестовой программы в файле `FractionTest.m`:

Программа 7.1 - часть main: FractionTest.m

```
#import "Fraction.h"

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

    // задание дроби (fraction) 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // вывод дроби

    NSLog (@"The value of myFraction is:");
    [myFraction print];
    [myFraction release];

    [pool drain];
    return 0;
}
```

Программа `FractionTest.m` включает файл секции `interface Fraction.h`, а *не* файл секции `implementation Fraction.m`. Таким образом, она разделена на три отдельных файла. Вы можете компилировать и запускать программу точно так же, как раньше. Выберите `Build and Go` в меню `Build` или щелкните на значке `Build and Go` в основном окне Xcode.

Если вы компилируете программы из командной строки, укажите компилятору Objective-C оба файла с расширением «.m». При использовании `gcc` команда строка будет выглядеть следующим образом:

```
gcc -framework Foundation Fraction.m FractionTest.m -o FractionTest
```

В результате будет создан исполняемый файл с именем FractionTest. Ниже приводятся результаты вывода при выполнении этой программы.

Вывод программы 7.1 FractionTest

```
The value of myFraction is: (Значение myFraction)  
1/3
```

Синтезируемые методы доступа

В Objective-C 2.0 методы-установщики (setter) и методы-получатели (getter) (которые называют *методами доступа – accessor methods*) можно генерироваться автоматически. Мы не описывали этого до настоящего момента, поскольку было важно, чтобы вы поняли, как писать эти методы самостоятельно. Однако это настолько удобно, что пора объяснить вам, как использовать эту возможность.

Первый шаг – задание свойств в директиве `@property` в секции `interface`. К этим свойствам относятся переменные экземпляра. Для класса `Fraction` в эту категорию попадают две переменные экземпляра: `numerator` и `denominator`. Ниже приводится новая секция `interface` с добавлением директивы `@property`.

```
@interface Fraction : NSObject  
{  
    int numerator;  
    int denominator;  
}  
  
@property int numerator, denominator;  
  
-(void) print;  
-(double) convertToNum;  
@end
```

Мы больше не включаем определения методов-получателей и методов-установщиков: `numerator`, `denominator`, `setNumerator:` и `setDenominator:`. Нам нужно, чтобы компилятор Objective-C автоматически генерировал, или *синтезировал* (*synthesize*) их для нас. Для этого служит директива `@synthesize` в секции `implementation`, как показано ниже.

```
#import «Fraction.h»  
  
@implementation Fraction  
  
@synthesize numerator, denominator;  
  
-(void) print
```

```

{
    NSLog(@"%@", numerator, denominator);
}
-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 1.0;
}
@end

```

Следующая строка указывает компилятору Objective-C, что нужно создать по паре методов (установщик и получатель) для каждой из указанных переменных экземпляра (`numerator` и `denominator`).

```
@synthesize numerator, denominator;
```

В общем случае, если у вас есть переменная экземпляра с именем `x`, то включение следующей строки в секцию `implementation` указывает компилятору, что нужно синтезировать метод-получатель с именем `x` и метод-установщик с именем `setX:`.

```
@synthesize x;
```

Автоматическое создание методов компилятором имеет смысл, поскольку генерируемые методы доступа эффективны и надежны при работе с несколькими потоками, на нескольких машинах, с несколькими процессорами.

А теперь вернемся к программе 7.1 и внесем указанные изменения в секции `interface` и `implementation`, чтобы методы доступа синтезировались автоматически. Убедитесь, что программа выводит те же результаты.

Доступ к свойствам с помощью оператора «точка»

Для доступа к свойствам имеется более удобный синтаксис. Чтобы получить значение числителя (`numerator`), хранящееся в `myFraction`, можно использовать строку

```
[myFraction numerator]
```

Сообщение `numerator` передается объекту `myFraction`, он возвращает требуемое значение. С помощью оператора «точка» можно написать эквивалентное выражение:

```
myFraction.numerator
```

Его общий формат

instance.property (экземпляр.свойство)

Этот синтаксис присваивает значения:

instance.property = значение

Это эквивалентно выражению *[instance setProperty: значение]*

В программе 7.1 мы задали numerator (числитель) и denominator (знаменатель) дроби 1/3 с помощью двух строк кода:

```
[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

Ниже приводится эквивалентный способ записи этих строк:

```
myFraction.numerator = 1;
myFraction.denominator = 3;
```

Далее мы будем использовать эти средства для синтеза методов и доступа к свойствам.

Передача методам нескольких аргументов

Продолжим работу с классом Fraction и внесем некоторые дополнения. Мы определили шесть методов. Было бы хорошо иметь такой метод, чтобы можно было задавать числитель и знаменатель с помощью одного сообщения. Если метод принимает несколько аргументов, после каждого аргумента ставится двоеточие. Аргументы становятся частью имени метода. Например, метод с именем *addEntryWithName:andEmail:* принимает два аргумента : имя (name) и адрес электронной почты (email). Метод *addEntryWithName:andEmail:andPhone:* принимает три аргумента: имя, адрес электронной почты и номер телефона (phone).

Метод, с помощью которого задаются numerator (числитель) и denominator (знаменатель), можно было бы назвать *setNumerator:andDenominator:* и использовать следующим образом:

```
[myFraction setNumerator: 1 andDenominator: 3];
```

Это вполне подходящий выбор для имени метода, но можно подобрать и более удобное для чтения имя метода, например, *setTo:over:.* Присвоим myFraction значение 1/3 и сравним это сообщение с предыдущим вариантом:

```
[myFraction setTo: 1 over: 3]; (присвоить 1, деленное на 3)
```

Я считаю, что оно читается лучше, но окончательный выбор за вами. Выбор имен методов важен для удобочитаемости программы.

А теперь опробуем работу с новым методом. Сначала добавим объявление *setTo:over:* в файл секции interface, см. программу 7.2.

Программа 7.2 - файл секции interface - Fraction.h

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

// Определение класса Fraction

@interface Fraction : NSObject
{
    int numerator;
    int denominator;
}

@property int numerator, denominator;

-(void)    print;
-(void)    setTo: (int) n over: (int) d;
-(double)  convertToNum;

@end
```

Теперь добавим определение этого метода в файл implementation.

Программа 7.2 - файл секции implementation - Fraction.m

```
#import «Fraction.h»

@implementation Fraction

@synthesize numerator, denominator;

-(void) print
{
    NSLog(@"%@", numerator, denominator);
}

-(double) convertToNum
{
    if (denominator != 0)
        return (double) numerator / denominator;
    else
        return 1.0;
}

-(void) setTo: (int) n over: (int) d
{
    numerator = n;
```

```
    denominator = d;
}
@end
```

Новый метод просто присваивает два передаваемых ему целых аргумента (n и d) переменным экземпляра для дроби (numerator и denominator).

Ниже приводится тестовая программа для этого метода.

Программа 7.2 - тестовый файл - FractionTest.m

```
#import "Fraction.h"
int main (int argc, char *argv[])

{
    Fraction *aFraction = [[Fraction alloc] init];
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    [aFraction setTo: 100 over: 200];
    [aFraction print];

    [aFraction setTo: 1 over: 3];
    [aFraction print];
    [aFraction release];

    [pool drain];
    return 0;
}
```

Вывод программы 7.2

```
100/200
1/3
```

Методы без имен аргументов

При создании имени метода имена аргументов не являются обязательными. Метод можно объявить следующим образом.

```
-(int) set: (int) n: (int) d;
```

В отличие от предыдущих примеров, для второго аргумента не задано никакого имени. Этот метод имеет имя set::, и два двоеточия означают, что методу передаются два аргумента, хотя имя одного из них не задано.

При работе с методом set:: в качестве разделителей используется двоеточие:

```
[aFraction set: 1 : 3];
```

Практика надежного программирования рекомендует указывать имена всех аргументов при написании новых методов, так как их отсутствие затрудняет сложение за ходом программы и делает присваивания фактических параметров менее интуитивным.

Операции над дробями

Продолжим работу с классом Fraction. Сначала напишем метод с именем add:, который позволит складывать одну дробь с другой. Ему будет передаваться дробь как аргумент. Ниже приводится объявление этого метода.

```
- (void) add: (Fraction *) f;
```

Объявление аргумента f:

```
(Fraction *) f
```

Аргумент для метода add: имеет тип класса Fraction. Звездочка обязательна. Следующее объявление является неверным:

```
(Fraction) f
```

В качестве аргумента методу add: будет передаваться одна дробь, и метод будет выполнять сложение этой дроби с получателем сообщения; в следующем выражении для дроби bFraction типа Fraction будет выполнено сложение с дробью aFraction типа Fraction.

```
[aFraction add: bFraction];
```

Напомним, что для сложения дробей a/b и c/d нужно выполнить следующие вычисления.

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Поместим код для этого метода в секцию @implementation.

```
// сложение дроби (Fraction) с дробью-получателем
```

```
- (void) add: (Fraction *) f
{
    // Для сложения двух дробей:
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    numerator = numerator * f.denominator
                + denominator * f.numerator;
    denominator = denominator * f.denominator;
}
```

Вы можете ссылаться на объект Fraction, который является получателем сообщения, по его полям numerator и denominator, но не можете непосредственно ссылаться таким же способом на переменные экземпляра аргумента f. Вместо этого вы должны получать их, применяя к f оператор «точка» (или передавая соответствующее сообщение f).

Предположим, что вы добавили эти объявления и определения для нового метода add: в файлы секций interface и implementation. Ниже приводится пример тестовой программы 7.3 и ее вывода.

Программа 7.3 - тестовый файл - FractionTest.m

```
#import «Fraction.h»

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

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

    // Задание дробей 1/4 и ? и их сложение

    [aFraction setTo: 1 over: 4];
    [bFraction setTo: 1 over: 2];

    // Вывод результатов

    [aFraction print];
    NSLog(@"%@", @"+");
    [bFraction print];
    NSLog(@"%@", @"=");

    [aFraction add: bFraction];
    [aFraction print];
    [aFraction release];
    [bFraction release];

    [pool drain];
    return 0;
}
```

Вывод программы 7.3

```
1/4
+
1/2
```

=
6/8

Тестовая программа выглядит достаточно просто. Выделяется память и инициализируются два объекта класса Fraction: aFraction и bFraction. Им присваиваются значения $1/4$ и $1/2$ соответственно, затем дробь bFraction прибавляется к дроби aFraction и результата сложения выводится. Метод add: прибавляет аргумент к объекту сообщения, изменяя этот объект. Это проверяется, когда происходит вывод значения aFraction в конце процедуры main. Можно было бы вывести значение aFraction до вызова метода add:, чтобы увидеть значение до того, как оно было изменено с помощью метода. Ниже в этой главе мы переопределим метод add:, чтобы он не влиял на значение своего аргумента.

Локальные переменные

Вы, вероятно, заметили, что результат сложения $1/4$ и $1/2$ выведен как $6/8$, а не $3/4$. Наша процедура сложения выполняет только заданные действия, в ней не предусмотрено сокращение дроби. Добавим новый метод, с помощью которого выполняется сокращение дробей.

Сокращение дроби означает, что нужно найти наибольшее число, на которое делятся без остатка числитель и знаменатель этой дроби, и выполнить деление на это число. Это действие называется поиском наибольшего общего делителя (greatest common divisor, gcd) числителя и знаменателя. Вы уже знаете, как это делать, из программы 5.7. Используя этот алгоритм, напишем метод сокращения дроби (reduce).

```
- (void) reduce
{
    int u = numerator;
    int v = denominator;
    int temp;

    while (v != 0) {
        temp = u % v;
        u = v;
        v = temp;
    }

    numerator /= u;
    denominator /= u;
}
```

В этом методе объявляются три целые переменные u, v и temp, которые являются *локальными*. Это означает, что они существуют только во время выполнения метода reduce и доступ к ним может выполняться только внутри метода, в котором они определены. В этом смысле они аналогичны переменным, объявленным в процедуре main; эти переменные тоже были локальными по отноше-

нию к `main`, и доступ к ним мог выполняться только внутри процедуры `main`. Ни один из написанных вами методов не мог иметь непосредственного доступа к этим переменным, определенным в `main`.

Локальные переменные не имеют начального значения по умолчанию, поэтому вы должны присвоить им некоторое значение, прежде чем использовать их. Трем локальным переменным в методе `reduce` присваиваются значения до того, как они используются, поэтому здесь не возникает никакой проблемы. В отличие от переменных экземпляра (которые сохраняют свои значения при вызовах методов), эти локальные переменные не имеют памяти, и после возврата из метода значения этих переменных исчезают. Каждый раз, когда происходит вызов метода, выполняется инициализация каждой локальной переменной, определенной в этом методе: ей присваивается значение, указанное при объявлении переменной.

Аргументы для метода

Имена, которые вы используете для обозначения аргументов метода, тоже являются локальными переменными. При выполнении метода любые аргументы, передаваемые методу, копируются в эти переменные. Поскольку метод работает с копией аргументов, он *не* может изменить исходные значения, переданные методу. Предположим, у нас есть метод с именем `calculate`; определенный следующим образом:

```
- (void) calculate: (double) x
{
    x *= 2;

    ...
}
```

Предположим также, что для его вызова использовано выражение:

```
[myData calculate: ptVal];
```

При вызове метода `calculate` значение, содержащееся в переменной `ptVal`, копируется в локальную переменную `x`. Поэтому изменение `x` внутри `calculate`: не оказывает никакого влияния на значение переменной `ptVal` — только копия ее значения сохраняется внутри `x`.

Для аргументов, которые являются объектами, вы можете изменять переменные экземпляра, хранящиеся в этом объекте. Об этом рассказывается в следующей главе.

Ключевое слово static

Чтобы локальная переменная сохраняла свое значение при нескольких вызовах метода, поместите ключевое слово `static` перед объявлением переменной. В следующей строке целая переменная `hitCount` объявляется как статическая переменная.

```
static int hitCount = 0;
```

В отличие от других локальных переменных, статическая переменная имеет начальное значение 0, поэтому приведенная выше инициализация является излишней. Кроме того, статические переменные инициализируются только один раз, в начале выполнения программы, и сохраняют свои значения при нескольких вызовах метода.

Приведенная ниже последовательность строк может содержаться внутри метода `showPage`, который должен следить за числом своих вызовов (в данном случае – число выведенных страниц).

```
-(void) showPage
{
    static int pageCount = 0;
    ...
    ++pageCount;
    ...
}
```

Локальной статической переменной присваивается значение 0 только один раз (при запуске программы), а затем это значение сохраняется при всех последующих вызовах метода `showPage`.

Если `pageCount` задана как локальная статическая переменная, то в ней подсчитывается число страниц, выведенных всеми объектами, которые вызвали метод `showPage`.

Если `pageCount` задана как переменная экземпляра, то в ней подсчитывается число страниц, выведенных каждым объектом, поскольку каждый объект будет содержать свою собственную копию `pageCount`.

Доступ к статическим или локальным переменным может выполняться только из метода, где они определены, поэтому даже статическая переменная `pageCount` доступна только внутри `showPage`. Чтобы сделать эту переменную доступной другим методам, нужно объявить ее *вне* объявлений любого метода (обычно это делают в начале файла `implementation`), например:

```
#import "Printer.h"
static int pageCount;

@implementation Printer
...
@end
```

После этого любой метод экземпляра или класса сможет выполнять доступ к переменной `pageCount`. Области действия переменных подробно рассматриваются в гл. 10.

Итак, мы можем вставить код метода `reduce` в файл секции `implementation Fraction.m`. Не забудьте объявить метод `reduce` в файле секции `interface Fraction.h`. После этого вы можете проверить метод в программе 7.4.

Программа 7.4 - тестовый файл - FractionTest.m

```
#import «Fraction.h»

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

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

    [aFraction setTo: 1 over: 4]; // set 1st fraction to 1/4
    [bFraction setTo: 1 over: 2]; // set 2nd fraction to 1/2

    [aFraction print];
    NSLog (@"+");
    [bFraction print];
    NSLog (@"=");

    [aFraction add: bFraction];

    // сокращение дроби после сложения и вывод результата

    [aFraction reduce];
    [aFraction print];

    [aFraction release];
    [bFraction release];

    [pool drain];
    return 0;
}
```

Вывод программы 7.4

```
1/4
+
1/2
=
```

```
3/4
```

Этот результат, несомненно, лучше!

Ключевое слово **self**

В программе 7.4 мы решили сокращать дробь вне метода `add:`. Мы могли бы делать это и внутри метода `add:`. Но как мы сможем указать методу `reduce` дробь, для которой нужно выполнить сокращение? Мы знаем, как непосредственно идентифицировать по имени переменные экземпляра внутри метода, но не знаем, как непосредственно идентифицировать получателя сообщения.

Вы можете использовать ключевое слово `self` для ссылки на объект, который является получателем текущего метода. Если внутри метода `add:` добавить

```
[self reduce];
```

то метод `reduce` будет применен к объекту `Fraction`, который был получателем метода `add:`, а это нам как раз и нужно. Далее вы увидите, насколько полезным может оказаться ключевое слово `self`. В методе `add:` оно применяется следующим образом.

```
- (void) add: (Fraction *) f
{
    // Для сложения двух дробей:
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    numerator = (numerator * [f denominator]) +
        (denominator * [f numerator]);
    denominator = denominator * [f denominator];

    [self reduce];
}
```

После сложения будет выполнено сокращение дроби.

Выделение памяти и возврат объектов из методов

Мы уже говорили, что метод `add:` изменяет значение объекта, который получает сообщение. Мы создадим новую версию метода `add:`, который будет вместо этого создавать новый объект типа `fraction` для сохранения результата сложения. В этом случае мы должны возвращать новый объект типа `Fraction` отправителю сообщения. Ниже приводится определение этого нового метода `add:`.

```
- (Fraction *) add: (Fraction *) f
{
    // Для сложения двух дробей:
    // a/b + c/d = ((a*d) + (b*c)) / (b * d)

    // В result будет сохраняться результат сложения
    Fraction *result = [[Fraction alloc] init];
```

```

int resultNum, resultDenom;

resultNum = numerator * f.denominator +
denominator * f.numerator;
resultDenom = denominator * f.denominator;

[result setTo: resultNum over: resultDenom];
[result reduce];

return result;
}

```

Первая строка определения этого метода:

```
- (Fraction *) add: (Fraction *) f;
```

Она указывает, что метод `add:` будет возвращать объект класса `Fraction` и принимать такой объект в качестве аргумента. Этот аргумент будет складываться с получателем сообщения, которым тоже является объект класса `Fraction`.

Метод `add:` выделяет память и инициализирует новый объект типа `Fraction` с именем `result` и затем определяет две локальные переменные с именами `resultNum` и `resultDenom`. Они будут использоваться для сохранения числителей и знаменателей, получаемых в результате сложения.

После выполнения сложения (как и раньше) и присваивания значений числителей и знаменателей локальным переменным можно задать значение `result` с помощью следующего выражения.

```
[result setTo: resultNum over: resultDenom];
```

После сокращения `result` (на наибольший общий делитель) его значение возвращается отправителю сообщения с помощью оператора `return`.

Отметим, что занимаемая объектом `result` память, которая выделена внутри метода `add:`, возвращается этим методом и не освобождается. Вы не можете освободить ее из метода `add:`, поскольку она требуется отправителю сообщения. Поэтому пользователь этого метода обязан знать, что возвращаемый объект является новым экземпляром и должен быть в дальнейшем освобожден. Это можно указать в соответствующей документации, которая предоставляется пользователям данного класса.

В программе 7.5 выполняется проверка нового метода `add:`

Программа 7.5 - тестовый файл - main.m

```

#import «Fraction.h»
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *bFraction = [[Fraction alloc] init];

```

```

Fraction *resultFraction;

[aFraction setTo: 1 over: 4]; // присваивание значения 1/4 первой дроби
[bFraction setTo: 1 over: 2]; // присваивание значения 1/2 второй дроби

[aFraction print];
NSLog (@"+");
[bFraction print];
NSLog (@"=");

resultFraction = [aFraction add: bFraction];
[resultFraction print];

// Непосредственный вывод результата, см. ниже,
// вызовет "утечку" памяти!

[[aFraction add: bFraction] print];
[aFraction release];
[bFraction release];
[resultFraction release];

[pool drain];
return 0;
}

```

Вывод программы 7.5

```

1/4
+
1/2
=
3/4

3/4

```

Сначала определяются два объекта типа Fraction: aFraction и bFraction, которым присваиваются значения 1/4 и 1/2 соответственно. Здесь также определяется объект Fraction с именем resultFraction (для которого не нужно выполнять выделение памяти и инициализацию). В этой переменной будет сохраняться результат последующих операций сложения.

В следующих строках кода сначала выполняется отправка сообщения add: получателю aFraction с передачей bFraction в качестве аргумента метода.

```

resultFraction = [aFraction add: bFraction];
[resultFraction print];

```

Результирующий объект типа Fraction, возвращаемый методом, сохраняется в resultFraction и затем выводится путем передачи сообщения print. Отметим, что

вы должны аккуратно завершить эту программу, освободив (`release`) `resultFraction`, хотя вы не выделяли для нее память (`allocate`) в `main`. Код выполнил метод `add:`, но освободить память должны вы сами. Следующее сообщение кажется очень удобным, но на самом деле здесь возникает проблема.

```
[[aFraction add: bFraction] print];
```

Поскольку здесь берется объект типа `Fraction`, возвращаемый методом `add:`, и ему передается сообщение `print`, у вас нет никакого способа освободить этот. Мы видим здесь пример *утечки памяти* (*memory leakage*). При многократном повторении вложенной передачи сообщения такого рода у вас будет понемногу накапливаться память, которую вы не сможете непосредственным образом освободить.

Чтобы решить эту проблему, можно сделать так, чтобы метод `print` возвращал своего получателя, и вы могли бы его освободить. Но это обходной способ. Проще разбить эти вложенные сообщения на два отдельных сообщения, как сделано выше.

Мы могли бы избежать использования временных переменных `resultNum` и `resultDenom` в методе `add:` с помощью следующего сообщения.

```
[result setTo: numerator * f.denominator + denominator * f.numerator
      over: denominator * f.denominator];
```

Мы не предлагаем писать такой сокращенный код, но вы можете встретить его у других программистов, поэтому нужно уметь читать и понимать подобные сложные выражения.

Рассмотрим еще один (последний в этой главе) пример, связанный с дробями. Это суммирование последовательности.

$$\sum_{i=1}^n \frac{1}{2^i}$$

Оно означает, что нужно сложить значения $1/2^i$, где i изменяется от 1 до n , то есть сложить $1/2 + 1/4 + 1/8 \dots$. Если сделать значение n достаточно большим, сумма будет приближаться к 1. Мы опробуем несколько значений n , чтобы посмотреть, насколько мы приблизимся к 1.

Программа 7.6 запрашивает ввод значения n и выполняет указанные вычисления.

Программа 7.6 FractionTest.m

```
#import "Fraction.h"
int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *aFraction = [[Fraction alloc] init];
    Fraction *sum = [[Fraction alloc] init], *sum2;
    int i, n, pow2;
```

```
[sum setTo: 0 over: 1]; // присваивание значения 0 первой дроби

NSLog (@"Enter your value for n:");
scanf ("%i", &n);

pow2 = 2;
for (i = 1; i <= n; ++i) {
    [aFraction setTo: 1 over: pow2];
    sum2 = [sum add: aFraction];
    [sum release]; // освобождение памяти предыдущей суммы
    sum = sum2;
    pow2 *= 2;
}

NSLog (@"After %i iterations, the sum is %g", n, [sum convertToNum]);
[aFraction release];
[sum release];

[pool drain];
return 0;
}
```

Вывод программы 7.6

Enter your value for n: (Введите значение n)

5

After 5 iterations, the sum is 0.96875 (По 5 итерациям сумма равна ...)

Вывод программы 7.6 (Повторный запуск)

Enter your value for n:

10

After 10 iterations, the sum is 0.999023

Вывод программы 7.6 (Повторный запуск)

Enter your value for n:

15

After 15 iterations, the sum is 0.999969

Переменной sum типа Fraction присваивается значение 0: числитель (numerator), равный 0, и знаменатель (denominator), равный 1. (А что произойдет, если задать числитель и знаменатель, равные 0?). Затем программа запрашивает у пользователя ввод значения n и считывает его с помощью scanf. Затем начинается цикл for для вычисления суммы последовательности. Перед циклом переменной pow2 присваивается значение 2. Эта переменная используется для сохранения значения 2^i . На каждом шаге цикла это значение умножается на 2.

Цикл for начинается с 1 и продолжается до значения n. На каждом шаге цикла aFraction присваивается значение $1/pow2$, то есть $1/2^i$. Это значение затем до-

бавляется к сумме (`sum`) с помощью определенного ранее метода `add`: Значение `result`, возвращаемое из `add`:, присваивается `sum2`, а не `sum`, чтобы избежать проблем утечки памяти. (А что произойдет, если присвоить `result` непосредственно `sum`?) Старое значение `sum` затем освобождается, и для следующего шага цикла `sum` присваивается новое значение суммы — `sum2`. Изучите, каким образом здесь происходит освобождение памяти для дробей. Для цикла `for`, который выполняется сотни или тысячи раз, при неверном освобождении памяти для дробей у вас быстро накопится большой объем неиспользуемой памяти.

По окончании цикла `for` конечный результат выводится в виде десятичного значения с помощью метода `convertToNum`. Остается только освободить два объекта: `aFraction` и конечный объект типа `Fraction`, сохраненный в `sum`. После этого завершается выполнение программы.

Результаты вывода показывают, что происходит при запуске программы. При значении 5 сумма последовательности равна 0.96875. При 15 результат очень близок к 1.

Расширение определений класса и файл секции interface

К настоящему моменту мы разработали небольшую библиотеку методов для работы с дробями. Ниже приводится файл секции `interface`, в котором применяется все, что мы умеем делать с использованием этого класса.

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

// Определение класса Fraction

@interface Fraction : NSObject

{
    int numerator;
    int denominator;
}

@property int numerator, denominator;

-(void)      print;
-(double)    convertToNum;
-(Fraction *) add: (Fraction *) f;
-(void)      reduce;
@end
```

Мы постепенно уточняли и расширяли класс путем добавления новых методов. Этот файл секции `interface` можно передать другому человеку, чтобы он мог писать программы, в которых операции с дробями. Если ему потребуется

добавить новый метод, он сможет сделать это непосредственно, расширив определение класса, или косвенно, определив собственный подкласс и добавив свои собственные новые методы. Этому посвящена следующая глава.

Упражнения

- Добавьте следующие методы к классу `Fraction` для завершения списка арифметических операций над дробями. Выполните сокращение (`reduce`) результата в каждом случае.

```
// Вычитание аргумента из получателя
-(Fraction *) subtract: (Fraction *) f;
// Умножение получателя на аргумент
-(Fraction *) multiply: (Fraction *) f;
// Деление получателя на аргумент
-(Fraction *) divide: (Fraction *) f;
```

- Модифицируйте метод `print` из класса `Fraction`, чтобы он мог принимать необязательный аргумент типа `BOOL`, который указывает, нужно ли выполнить сокращение дроби для ее вывода. Если дробь должна быть сокращена, сделайте так, чтобы сама дробь не изменялась необратимо.
 - Внесите изменения в программу 7.6, чтобы выводить результатирующую сумму (`sum`) как дробь, а не только как вещественное число.
 - Будет ли ваш класс `Fraction` работать с отрицательными дробями? Например, можно ли сложить $-1/4$ и $-1/2$ и получить правильный результат? Напишите тестовую программу, чтобы проверить свой ответ.
 - Внесите изменения в метод `print` класса `Fraction`, чтобы выводить дроби, превышающие 1, как смешанные числа (с целой и дробной частями), например, дробь $5/3$ выводить как $1\frac{2}{3}$.
 - В упражнении 6 главы 4 определяется новый класс с именем `Complex` для работы с комплексными мнимыми числами. Добавьте новый метод с именем `add:`, который можно использовать для сложения двух комплексных чисел. Чтобы сложить два комплексных числа, нужно по отдельности сложить вещественные и мнимые части:
- $$(5.3 + 7i) + (2.7 + 4i) = 8 + 11i$$
- Этот метод `add:` должен сохранять и возвращать результат в виде нового числа типа `Complex` в соответствии с объявлением метода:
- ```
-(Complex *) add: (Complex *) complexNum;
```
- Учтите в тестовой программе все потенциальные проблемы утечки памяти.
- Для класса `Complex`, разработанного в упражнении 6 главы 4, с учетом расширения, сделанного в упражнении 6 этой главы, создайте отдельные файлы секций `interface` и `implementation` – `Complex.h` и `Complex.m`. Создайте отдельный файл тестовой программы.

# Глава 8

# Наследование

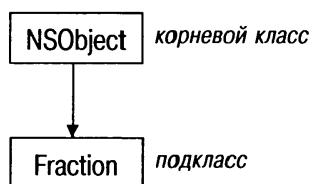
В этой главе рассматривается один из ключевых принципов, который делает объектно-ориентированное программирование столь мощным. Принцип *наследования* позволяет расширять существующие определения классов и настраивать их для приложений.

## Все начинается с корня

Мы уже рассматривали идею родительского класса в главе 3. Родительский класс может быть дочерним классом другого родительского класса. Класс, не имеющий никакого родительского класса, находится вверху иерархии и называется *корневым классом* (*root class*). В Objective-C вы можете определять собственный родительский класс, но обычно используют возможности существующих классов. Все классы, которые мы определяли до настоящего момента, являются дочерними классами (потомками) корневого класса с именем `NSObject`, который указывался в файле секции `interface` следующим образом:

```
@interface Fraction: NSObject
...
@end
```

Класс `Fraction` образуется из класса `NSObject`. Поскольку `NSObject` находится вверху иерархической структуры (то есть над ним уже нет никаких классов), он называется *корневым классом* (рис. 8.1). Класс `Fraction` является *дочерним* (*child class*) или *подклассом* (*subclass*).



**Рис. 8.1.** Корневой класс и подкласс

С точки зрения терминологии, можно говорить о классах, родительских классах и дочерних классах или о классах, подклассах и суперклассах. Если определяется какой-либо новый класс (не корневой), он наследует определенные свойства. Например, все переменные экземпляра и методы из родительского класса неявным образом становятся частью определения нового класса. Это означает, что соответствующий подкласс имеет непосредственный доступ к методам и переменным экземпляра, как будто они были определены непосредственно в определении этого подкласса.

Проиллюстрируем концепцию наследования с помощью простого (хотя и несколько надуманного) примера. Рассмотрим объявление объекта ClassA, содержащего один метод с именем initVar.

```
@interface ClassA: NSObject
{
 int x;
}
-(void) initVar;
@end
```

Метод initVar просто присваивает значение 100 переменной экземпляра класса ClassA.

```
@implementation ClassA
-(void) initVar
{
 x = 100;
}
@end
```

Теперь определим класс с именем ClassB.

```
@interface ClassB: ClassA
-(void) printVar;
@end
```

В первой строке этого объявления

```
@interface ClassB: ClassA
```

указывается, что ClassB объясняется не как подкласс NSObject, а как подкласс класса ClassA. И хотя родительским классом (суперклассом) для класса ClassA является NSObject, родительским классом для ClassB является ClassA (рис. 8.2).

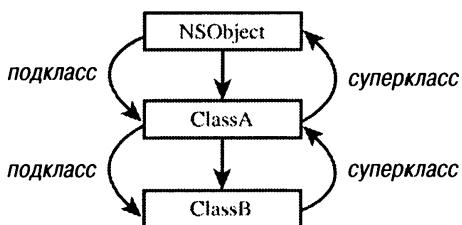
На этом рисунке корневой класс не имеет суперкласса, а ClassB, который находится внизу иерархии, не имеет подкласса. Тем самым ClassA является подклассом NSObject, ClassB является подклассом класса ClassA, а также класса NSObject (он является его «подподклассом», или *внуком*). Кроме того, NSObject является суперклассом для ClassA, который является суперклассом для ClassB. NSObject является также суперклассом для ClassB.

Ниже приводится полное определение для ClassB, где определяется один метод с именем printVar.

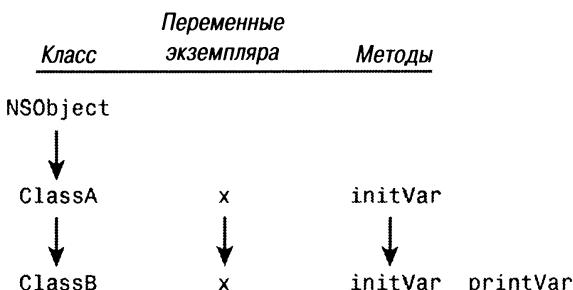
```
@interface ClassB: ClassA
-(void) printVar;
@end
```

```
@implementation ClassB
-(void) printVar
{
 NSLog (@"x = %i", x);
}
@end
```

Метод printVar выводит значение переменной экземпляра x, и при этом мы не определяем никаких переменных экземпляра в ClassB. Поскольку ClassB является подклассом для ClassA, он наследует все переменные экземпляра из ClassA (в данном случае – только одну переменную). Это показано на рис. 8.3.



**Рис. 8.2.** Подклассы и суперклассы



**Рис. 8.3.** Наследование переменных экземпляра и методов

(Конечно, на рис. 8.3 не показаны другие методы или переменные экземпляра, наследуемые из класса NSObject.)

Теперь рассмотрим, как это все сочетается в программе. Для краткости поместим все объявления и определения в один файл.

**Программа 8.1**

```
// Простой пример наследования

#import <Foundation/Foundation.h>

// Объявление и определение ClassA

@interface ClassA: NSObject
{
 int x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
 x = 100;
}
@end

// Объявление и определение ClassB

@interface ClassB : ClassA
-(void) printVar;
@end

@implementation ClassB
-(void) printVar
{
 NSLog(@"x = %i", x);
}
@end

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

 ClassB *b = [[ClassB alloc] init];

 [b initVar]; // будет использовать унаследованный метод
 [b printVar]; // раскрывает значение x;
```

```
[b release];
```

```
[pool drain];
```

```
return 0;
```

```
}
```

### Вывод программы 8.1

```
x = 100
```

Мы начинаем с определения `b` как объекта класса `ClassB`. После выделения памяти и инициализации `b` следует передача сообщения для применения к `b` метода `initVar`. Но в определении класса `ClassB` мы не видим, что определен какой-либо метод. Метод `initVar` был определен в классе `ClassA`, а поскольку `ClassA` является родительским классом для `ClassB`, он может использовать все методы `ClassA`. Метод `initVar` является унаследованным методом по отношению к `ClassB`.

---

**Примечание.** Методы `alloc` и `init`, которые мы все время используем, никогда не определялись, поскольку мы применяем их как унаследованные методы.

---

После отправки объекту `b` сообщения `initVar` вызывается метод `printVar` для вывода значения переменной экземпляра `x`. Результат вывода `x = 100` подтверждает, что `printVar` смог получить доступ к этой переменной экземпляра, поскольку он тоже был унаследован.

Концепция наследования действует вниз по всей цепочке. Так, если определить новый класс с именем `ClassC`, родительским классом для которого является `ClassB`:

```
@interface ClassC: ClassB;
...
@end
```

то `ClassC` унаследует все методы и переменные экземпляра класса `ClassB`, который, в свою очередь, наследует методы и переменные экземпляра класса `ClassA`, который, в свою очередь, наследует методы и переменные экземпляра `NSObject`.

Каждый экземпляр класса получает свои собственные переменные экземпляра, даже если они наследуются. Это означает, что объект класса `ClassC` и объект класса `ClassB` будут иметь свои собственные отдельные переменные экземпляра.

### Поиск подходящего метода

Прежде чем передавать сообщение объекту, надо выбрать подходящий метод для применения к этому объекту. Здесь действуют совершенно простые правила. Сначала нужно проверить класс, которому принадлежит данный объект. Если метод с нужным именем определен в этом классе, применяется этот метод. Если это не так, проверяется родительский класс, и если в нем определен данный метод, то применяется этот метод. В противном случае поиск продолжается.

Проверка родительских классов выполняется до тех пор, пока не найдется класс, который содержит указанный метод, или метод не будет найден даже в корневом классе. В первом случае вы можете действовать дальше; во втором случае возникает проблема и генерируется сообщение, аналогичное следующему:

```
warning: 'ClassB' may not respond to '-inity'
(предупреждение: 'ClassB', возможно, не отвечает '-inity')
```

Это означает, что вы ошибочно пытаетесь передать сообщение с именем `unity` переменной типа `ClassB`. Компилятор указывает, что переменные этого типа «не знают», как реагировать на такой метод. Это было определено после проверки методов класса `ClassB` и методов его родительских классов вплоть до корневого класса (которым в данном случае является `NSObject`).

В некоторых случаях, когда метод не найден, никакое сообщение не генерируется. Это означает, что используется *пересылка (forwarding)*, описание которой приводится в главе 9.

## Расширение посредством наследования: добавление новых методов

Наследование часто используется для расширения класса. Предположим, требуется разработать несколько классов для работы с двумерными графическими объектами, такими как прямоугольник, окружность и треугольник. В данном случае нас интересуют только прямоугольники (`rectangle`). Вернемся к упражнению 7 главы 4 и начнем с секции `@interface`.

```
@interface Rectangle: NSObject
```

```
{
 int width;
 int height;
}
```

```
@property int width, height;
-(int) area;
-(int) perimeter;
@end
```

У вас будут синтезируемые методы для задания ширины (`width, w`) и высоты (`height, h`) прямоугольника, а также возврата этих значений, и ваши собственные методы для вычисления его площади (`area`) и периметра (`perimeter`). Добавим метод, который позволит задавать ширину и высоту прямоугольника в одном сообщении:

```
-(void) setWidth: (int) w andHeight: (int) h;
```

Предположим, что это объявление нового класса введено в файл с именем Rectangle.h. Файл секции implementation с именем Rectangle.m может выглядеть следующим образом.

```
#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;

-(void) setWidth: (int) w andHeight: (int) h

{
 width = w;
 height = h;
}

-(int) area
{
 return width * height;
}

-(int) perimeter
{
 return (width + height) * 2;
}
@end
```

Каждое определение метода достаточно очевидно. В программе 8.2 показана процедура main для тестирования.

## Программа 8.2

```
#import "Rectangle.h"
#import <stdio.h>

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

 Rectangle *myRect = [[Rectangle alloc] init];

 [myRect setWidth: 5 andHeight: 8];

 NSLog (@"Rectangle: w = %i, h = %i",
 myRect.width, myRect.height);
```

```

NSLog(@"Area = %i, Perimeter = %i",
 [myRect area], [myRect perimeter]);
[myRect release];

[pool drain];
return 0;
}

```

### Вывод программы 8.2

Rectangle (прямоугольник): w = 5, h = 8  
 Area (площадь) = 40, Perimeter (периметр) = 26

Сначала выполняются выделение памяти и инициализация объекта `myRect`; затем задается его ширина (5) и высота (8). Это проверяется в первой строке вывода. Затем вычисляются площадь и периметр прямоугольника путем вызова с помощью сообщения, и возвращаемые значения передаются для вывода процедуре `NSLog`.

Для работы с квадратами (`square`) можно было бы определить новый класс с именем `Square` и определить в нем методы, аналогичные методам класса `Rectangle`. Можно также учесть, что квадрат является частным случаем прямоугольника, у которого равны ширина и высота.

Поэтому проще создать новый класс с именем `Square` и сделать его подклассом класса `Rectangle`. Это позволит использовать все методы и переменные класса `Rectangle` помимо ваших собственных. После этого достаточно добавить только методы задания определенного значения для стороны квадрата и считывания этого значения. В программе 8.3 показаны файлы секций `interface` и `implementation` для нового класса `Square`.

### Программа 8.3. Файл Square.h для секции interface

```

#import "Rectangle.h"

@interface Square: Rectangle

-(void) setSide: (int) s;
-(int) side;
@end

```

### Программа 8.3. Файл Square.m для секции implementation

```

#import "Square.h"

@implementation Square: Rectangle

-(void) setSide: (int) s
{
 [self setWidth: s andHeight: s];
}

```

```
- (int) side
{
 return width;
}
@end
```

Мы определили класс `Square` как подкласс класса `Rectangle`, который объявлен в заголовке файла `Rectangle.h`. Здесь не требуется добавлять какие-либо переменные экземпляра, но добавлены новые методы `setSide:` и `side`.

Для квадрата достаточно задать только одну сторону, которая представляется как два числа. Все это скрыто от пользователя класса `Square`. Пользователю не нужно думать об этих деталях, поскольку здесь действует инкапсуляция, о которой мы говорили раньше.

В методе `setSide:` используется метод, наследуемый из класса `Rectangle`, задающий ширину и высоту прямоугольника. Таким образом, `setSide:` вызывает метод `setWidth:andHeight:` из класса `Rectangle`, передавая параметр `s` как значение для ширины (`width`) и высоты (`height`). Больше ничего не требуется. Тот, кто будет работать с объектом класса `Square`, может задавать размеры квадрата с помощью метода `setSide:` и использовать методы из класса `Rectangle` для вычисления площади и периметра квадрата. Ниже показана тестовая программа 8.3 и ее вывод для нашего нового класса `Square`.

### Программа 8.3 - тестовая программа

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

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

 Square *mySquare = [[Square alloc] init];

 [mySquare setSide: 5];

 NSLog (@"Square s = %i", [mySquare side]);
 NSLog (@"Area = %i, Perimeter = %i",
 [mySquare area], [mySquare perimeter]);
 [mySquare release];

 [pool drain];
 return 0;
}
```

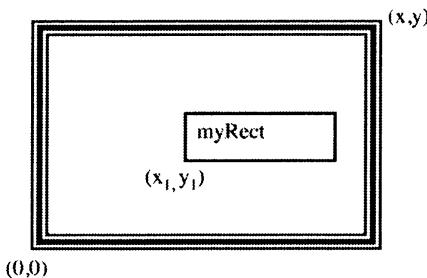
### Вывод программы 8.3

```
Square (квадрат) s = 5
Area (площадь) = 25, Perimeter (периметр) = 20
```

Способ, с помощью которого был определен класс `Square`, является базовым методом работы с классами в Objective-C: расширение того, что уже кем-то сделано, чтобы реализовать то, что вам нужно. Помимо этого, существует механизм *категорий* (*category*), который позволяет добавлять новые методы к существующему определению класса по модульному принципу, то есть без необходимости постоянно добавлять новые определения в файлы секций `interface` и `implementation`. Это особенно удобно в тех случаях, когда вы не имеете доступа к исходному коду. Описание категорий см. в главе 11.

## Класс для точки и выделение памяти

Класс `Rectangle` используется только для хранения размеров прямоугольника. В реальных графических приложениях требуется следить за всевозможной дополнительной информацией, например, цветом заполнения прямоугольника, цветом линий, точкой начала координат прямоугольника (`origin`) внутри окна и т.д. Для этого можно легко расширить существующий класс. Сейчас мы реализуем идею начала координат прямоугольника. Примем за начало координат декартовы координаты ( $x, y$ ) левого нижнего угла прямоугольника. Если вы разрабатываете чертежное приложение, эта точка может представлять местоположение прямоугольника внутри окна (рис. 8.4).



**Рис. 8.4.** Прямоугольник, нарисованный в окне

На рис. 8.4 точка начала координат прямоугольника представлена как  $(x_1, y_1)$ .

Вы можете расширить класс `Rectangle`, чтобы сохранять координаты  $x, y$  точки начала прямоугольника в виде двух отдельных значений или определить класс с именем `XYPoint` (возможно, вы помните об этой задаче из упражнения 7 главы 3).

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

@interface XYPoint: NSObject

{
 int x;
 int y;
}
@property int x, y;
```

```
- (void) setX: (int) xVal andY: (int) yVal;
@end
```

Теперь вернемся к классу Rectangle. Нам нужно сохранять координаты начала прямоугольника, поэтому требуется добавить к определению этого класса еще одну переменную экземпляра с именем origin.

```
@interface Rectangle: NSObject
{
 int width;
 int height;
 XYPoitn *origin;
}
...
```

Вполне разумно добавить метод, с помощью которого задаются ичитываются координаты начала прямоугольника. Мы не будем здесь синтезировать методы доступа (accessor methods) для координат начала, а напишем их сами.

## Директива @class

На данный момент вы можете работать с прямоугольниками (и квадратами), задавая их ширину, высоту и координаты начала. Рассмотрим в полном виде файл Rectangle.h секции interface.

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

@class XYPoitn;
@interface Rectangle: NSObject
{
 int width;
 int height;
 XYPoitn *origin;
}

@property int width, height;

-(XYPoitn *) origin;
-(void) setOrigin: (XYPoitn *) pt;
-(void) setWidth: (int) w andHeight: (int) h
-(int) area;
-(int) perimeter;
@end
```

В файле Rectangle.h использована новая директива:

```
@class XYPoitn;
```

Она требуется нам, поскольку компилятору нужно знать, что представляет собой `XYPoitn`, когда он встречается в одной из переменных экземпляра, определенных для `Rectangle`. Имя этого класса используется также в объявлениях типов аргумента и возвращаемого значения для наших методов `setOrigin:` и `origin` соответственно. У вас есть и другой вариант выбора — импортировать файл заголовка, например, в следующем виде:

```
#import "XYPoitn.h"
```

Директива `@class` эффективнее, поскольку компилятору не нужно обрабатывать весь файл `XYPoitn.h` (хотя это небольшой файл); компилятору достаточно знать, что `XYPoitn` является именем класса. Если нужна ссылка на один из методов класса `XYPoitn`, то директивы `@class` недостаточна, поскольку компилятору потребуется дополнительная информация: аргументы, передаваемые методу, их типы, тип возвращаемого значения метода.

Заполним формы для нового класса `XYPoitn` и новых методов класса `Rectangle`, чтобы протестировать их в программе. В программе 8.4 имеется файл секции `implementation` для класса `XYPoitn`.

Сначала в ней показаны новые методы для класса `Rectangle`.

#### **Программа 8.4.** Rectangle.m - добавленные методы

```
#import "XYPoitn.h"

-(void) setOrigin: (XYPoitn *) pt
{
 origin = pt;
}

-(XYPoitn *) origin
{
 return origin;
}
@end
```

Затем показаны полные определения классов `XYPoitn` и `Rectangle` и тестовая программа для их проверки.

#### **Программа 8.4.** XYPoitn.h - файл секции interface

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

@interface XYPoitn: NSObject

{
 int x;
 int y;
}
```

```
@property int x, y;

-(void) setX: (int) xVal andY: (int) yVal;
@end
```

**Программа 8.4.** XYPoint.m - файл секции implementation

```
#import "XYPoint.h"

@implementation XYPoint

@synthesize x, y;
-(void) setX: (int) xVal andY: (int) yVal
{
 x = xVal;
 y = yVal;
}
@end
```

**Программа 8.4.** Rectangle.h - файл секции interface

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

@class XYPoint;
@interface Rectangle: NSObject
{
 int width;
 int height;
 XYPoint *origin;
}

@property int width, height;

-(XYPoint *) origin;
-(void) setOrigin: (XYPoint *) pt;
-(void) setWidth: (int) w andHeight: (int) h;
-(int) area;
-(int) perimeter;
@end
```

**Программа 8.4.** Rectangle.m - файл секции implementation

```
#import "Rectangle.h"

@implementation Rectangle

@synthesize width, height;
```

```

-(void) setWidth: (int) w andHeight: (int) h

{
 width = w;
 height = h;
}

-(void) setOrigin: (XYPoint *) pt
{
 origin = pt;
}

-(int) area
{
 return width * height;
}

-(int) perimeter
{
 return (width + height) * 2;
}

-(Point *) origin
{
 return origin;
}
@end

```

**Программа 8.4.** тестовая программа

```

#import "Rectangle.h"
#import "XYPoint.h"

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

 Rectangle *myRect = [[Rectangle alloc] init];
 XYPoint *myPoint = [[XYPoint alloc] init];

 [myPoint setX: 100 andY: 200];

 [myRect setWidth: 5 andHeight: 8];
 myRect.origin = myPoint;
}

```

```
 NSLog(@"Rectangle w = %i, h = %i",
 myRect.width, myRect.height);

 NSLog(@"Origin at (%i, %i)",
 myRect.origin.x, myRect.origin.y);

 NSLog(@»Area = %i, Perimeter = %i»,
 [myRect area], [myRect perimeter]);
 [myRect release];
 [myPoint release];

 [pool drain];
 return 0;
}
```

#### Вывод программы 8.4

```
 Rectangle (Прямоугольник) w = 5, h = 8
 Origin at (Координаты начала) (100, 200)
 Area (Площадь) = 40, Perimeter (Периметр) = 26
```

Внутри процедуры `main` выделена память и инициализирован объект класса `Rectangle` с именем `myRect` и объект класса `XYPoint` с именем `myPoint`. С помощью метода `setX:andY:` объекту `myPoint` присваивается значение `(100, 200)`. После задания ширины и высоты этого прямоугольника (5 и 8 соответственно) вызывается метод `setOrigin`, чтобы задать для координат начала прямоугольника точку, указанную в `myPoint`. Затем с помощью трех вызовов процедуры `NSLog` выполняется считывание и вывод этих значений. В выражении

`myRect.origin.x`

считывается объект класса `XYPoint`, возвращенный методом доступа `origin`, и применяется оператор «точка» для получения координаты «`x`» начала прямоугольника. В следующем выражении считывается координата «`y`» начала прямоугольника:

`myRect.origin.y`

### Классы, владеющие своими объектами

Можете ли вы объяснить результаты вывода программы 8.5?

#### Программа 8.5

```
#import "Rectangle.h"
#import "XYPoint.h"

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

```

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

Rectangle *myRect = [[Rectangle alloc] init];
XYPoint *myPoint = [[XYPoint alloc] init];

[myPoint setX: 100 andY: 200];

[myRect setWidth: 5 andHeight: 8];
myRect.origin = myPoint;

NSLog (@"Origin at (%i, %i)",
 myRect.origin.x, myRect.origin.y);

[myPoint setX: 50 andY: 50];
NSLog (@"Origin at (%i, %i)",
 myRect.origin.x, myRect.origin.y);
[myRect release];
[myPoint release];

[pool drain];
return 0;
}

```

### Вывод программы 8.5

Origin at (Координаты начала) (100, 200)  
 Origin at (50, 50)

В этой программе значение объекта `myPoint` было изменено с (100, 200) на (50, 50), то есть были изменены координаты начала прямоугольника. Но почему это произошло? Здесь не было явным образом задано новое значение начала прямоугольника, почему оно изменилось? Вернемся к определению метода `setOrigin:`, чтобы понять причину:

```

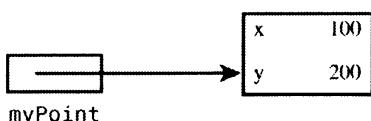
-(void) setOrigin: (XYPoint *) pt
{
 origin = pt;
}

```

При вызове метода `setOrigin:` с помощью выражения

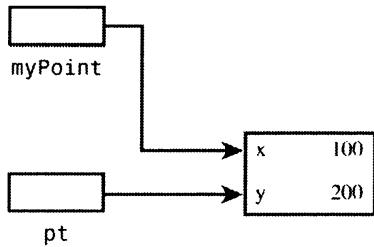
```
myRect.origin = myPoint;
```

значение `myPoint` передается этому методу как аргумент. Это значение указывает место в памяти, где хранится данный объект `XYPoint` (рис. 8.5).



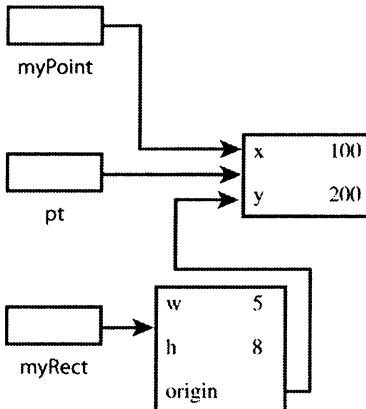
**Рис. 8.5.** Объект `myPoint` класса `XYPoint` в памяти

Это значение, сохраненное в `myPoint` и являющееся указателем места в памяти, копируется в локальную переменную `pt`, определенную внутри метода. После этого `pt` и `myPoint` являются ссылкой на одни и те же данные, хранящиеся в памяти (рис. 8.6).



**Рис. 8.6.** Передача методу информации о начале прямоугольника

Когда переменной `origin` присваивается `pt` внутри этого метода, указатель, хранящийся внутри `pt`, копируется в переменную экземпляра `origin` (рис. 8.7).



**Рис. 8.7.** Задание начала (`origin`) прямоугольника

Поскольку `myPoint` и переменная `origin`, хранящаяся в `myRect`, ссылаются на одну и ту же область в памяти (как и локальная переменная `pt`), при последующем изменении значения `myPoint` на (50, 50) изменяется и значение начала прямоугольника.

Чтобы избежать этой проблемы, нужно модифицировать метод `setOrigin`: так, чтобы он выделял (`alloc`) свою собственную точку и присваивал началу прямоугольника (`origin`) эту точку.

```

-(void) setOrigin: (XYPoint *) pt
{
 origin = [[XYPoint alloc] init];
 [origin setX: pt.x andY: pt.y];
}

```

Метод сначала выделяет память и инициализирует новый объект класса `XYPoint`. В выражении для сообщения

```
[origin setX: pt.x andY: pt.y];
```

новому объекту класса `XYPoint` присваивается значение координат `x,y` аргумента, передаваемого методу.

Это изменение в методе `setOrigin:` означает, что теперь каждый экземпляр `Rectangle` владеет своим собственным экземпляром `XYPoint`. Теперь он не только осуществляет выделение памяти для `XYPoint`, но и освобождает эту память. Если класс содержит другие объекты, бывает нужно, чтобы он владел некоторыми или всеми объектами. Для прямоугольника класс `Rectangle` должен владеть объектом начала (`origin`) прямоугольника, поскольку это один из основных атрибутов.

Но была ли освобождена память, которая использовалась для `origin`? Освобождение памяти, занятой для прямоугольника (`myRect`), не освобождает память, которая была выделена для начала прямоугольника (`origin`). Чтобы освободить эту память, нужно вставить в `main` строку

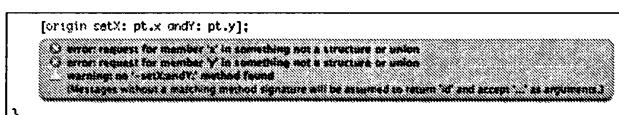
```
[[myRect origin] release];
```

В результате будет освобожден объект `XYPoint`, возвращаемый методом `origin`. Вы должны сделать это до того, как освободите память для самого объекта `Rectangle`, поскольку ни одна из переменных, содержащихся в этом объекте, не действительна после того, как освобождена память объекта. Необходима следующая последовательность строк кода.

```
[[myRect origin] release]; // Освобождение памяти для origin
[myRect release]; // Освобождение памяти для прямоугольника
```

Вы вынуждены помнить, что нужно освобождать память непосредственно для `origin`, хотя не вы выделяли эту память; это сделал класс `Rectangle`. В следующем разделе, «Замещающие методы», вы узнаете, как сделать, чтобы `Rectangle` освобождал память.

После перекомпиляции и перезапуска программы 8.5 с модифицированным методом появляются сообщения об ошибках (рис. 8.8).



**Рис. 8.8.** Сообщения компилятора об ошибках

Проблема возникает из-за того, что мы использовали в модифицированном методе некоторые методы из класса `XYPoint`, и теперь компилятору требуется больше информации об этом классе, чем дает директива `@class`. Нужно вернуться назад и заменить эту директиву импортом:

```
#import "XYPoint.h"
```

### Вывод программы 8.5в

```
Origin at (100, 200)
Origin at (100, 200)
```

Это уже лучше. Теперь изменение значения `myPoint` на `(50, 50)` внутри `main` не окажет никакого влияния на координаты начала прямоугольника, поскольку копия этой точки была создана внутри метода `setOrigin`: объекта `Rectangle`.

Мы не синтезировали здесь методы `origin`, поскольку синтезированный метод-установщик `setOrigin`: будет действовать точно так же, как метод, написанный нами первоначально. По умолчанию синтезированный метод-установщик просто копирует указатель объекта, а не сам объект.

Вы можете синтезировать другой тип метода-установщик, который создает копию объекта, но для этого вам нужно научиться писать копирующий метод. Мы вернемся к этой теме в главе 17.

## Замещающие методы

Выше уже говорилось, что мы не можем удалить или обойти методы при наследовании, но можем изменить определение наследуемого метода путем замещения.

Возвращаясь к двум классам, `ClassA` и `ClassB`, предположим, что нужно написать собственный метод `initVar` для `ClassB`. Мы уже знаем, что `ClassB` будет наследовать метод `initVar`, определенный в классе `ClassA`, но можно ли создать новый метод с тем же именем для замены наследуемого метода? Да, можно, для этого нужно просто определить новый метод с тем же именем. Метод, определенный с таким же именем, как в родительском классе, заменяет, или замещает (`override`), унаследованное определение. Новый метод должен иметь такой же тип возвращаемого значения и принимать такое же число аргументов такого же типа, как метод, который вы замещаете.

В программе 8.6 показан простой пример, отражающий эту концепцию.

### Программа 8.6

```
// Замещающие методы

#import <Foundation/Foundation.h>

// Объявление и определение класса ClassA

@interface ClassA: NSObject

{
 int x;
}

-(void) initVar;
@end
```

```
@implementation ClassA
-(void) initVar
{
 x = 100;
}
@end

// Объявление и определение класса ClassB

@interface ClassB: ClassA
-(void) initVar;
-(void) printVar;
@end

@implementation ClassB
-(void) initVar // добавляемый метод
{
 x = 200;
}

-(void) printVar
{
 NSLog(@"x = %i", x);
}
@end

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

 ClassB *b = [[ClassB alloc] init];

 [b initVar]; // использование замещающего метода в B

 [b printVar]; // раскрытие значения x;
 [b release];

 [pool drain];
 return 0;
}
```

### Вывод программы 8.6

```
x = 200
Сообщение
[b initVar];
```

вызывает использование метода `initVar`, определенного в `ClassB`, а не одноименного метода из `ClassA`, как в предыдущем примере (рис. 8.9).



**Рис. 8.9.** Замещение метода `initVar`

## Какой из методов выбирается?

Мы уже описывали, каким образом система выполняет поиск в иерархии, чтобы найти метод для применения к объекту. Если у вас есть методы с одинаковым именем в различных классах, то нужный метод выбирается в соответствии с классом получателя сообщения. В программе 8.7 используются такие же определения для классов `ClassA` и `ClassB`, как выше.

### Программа 8.7

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

// Здесь нужно вставить определения для классов ClassA и ClassB

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

 ClassA *a = [[ClassA alloc] init];
 ClassB *b = [[ClassB alloc] init];

 [a initVar]; // использование метода из ClassA
 [a printVar]; // раскрытие значения x;

 [b initVar]; // использование замещающего метода из ClassB
 [b printVar]; // раскрытие значения x;
 [a release];
 [b release];

 [pool drain];
 return 0;
}
```

Для этой программы вы получите следующее предупреждающее сообщение:

warning: 'ClassA' may not respond to '-printVar'  
(предупреждение: 'ClassA', возможно, не отвечает '-printVar')

Что произошло? Рассмотрим объявление класса ClassA:

```
// Объявление и определение класса ClassA
```

```
@interface ClassA: NSObject
{
 int x;
}

-(void) initVar;
@end
```

Обратите внимание, что не объявлен никакой метод printVar. Этот метод объявлен и определен в ClassB. И хотя объекты ClassB и их потомки могут использовать этот метод путем наследования, объекты класса ClassA не могут это сделать, поскольку данный метод определен ниже в цепочке иерархии.

---

**Примечание.** Вы могли бы принудительно использовать этот метод, но мы не будем описывать способы обхода, поскольку это не соответствует практике надежного программирования.

---

Вернемся к примеру и добавим метод printVar в класс ClassA, чтобы вывести значение его переменных экземпляра.

```
// Объявление и определение класса ClassA
```

```
@interface ClassA: NSObject
{
 int x;
}

-(void) initVar;
-(void) printVar;
@end

@implementation ClassA
-(void) initVar
{
 x = 100;
}

-(void) printVar
{
 NSLog(@"%@", @"x = %i", x);
```

```
}
```

```
@end
```

Объявление и определение класса ClassB остается без изменений. Запустим компиляцию и выполнение программы.

### Вывод программы 8.7

```
x = 100
```

```
x = 200
```

а и b определены как объекты классов ClassA и ClassB соответственно. После выделения памяти и инициализации передается сообщение для объекта a, у которого запрашивается применение метода initVar. Этот метод определен в определении класса ClassA, поэтому выбирается именно он. Он присваивает значение 100 переменной экземпляра x и выполняет возврат. Затем вызывается метод printVar, только что добавленный в класс ClassA, чтобы вывести значение x.

Выделение памяти и инициализация для объекта b класса ClassB выполняется так же как и для объекта класса ClassA, его переменной экземпляра присваивается значение 200 и выводится ее значение.

Постарайтесь разобраться, как для переменных a и b происходит выбор метода, исходя из класса, которому они принадлежат. Это одна из базовых концепций объектно-ориентированного программирования в Objective-C.

В качестве упражнения попробуйте удалить метод printVar из класса ClassB. Получится ли это? Почему?

## Замещение метода dealloc и ключевое слово super

Теперь, когда вы знаете, как замещать методы, вернемся к программе 8.5B, чтобы изучить более подходящий способ освобождения памяти, выделенной для origin. Метод setOrigin: выделяет память для своего собственного объекта origin класса XYPoint, и вы обязаны освободить эту память. В программе 8.6 освобождение памяти выполнял оператор:

```
[[myRect origin] release];
```

Вам не нужно заботиться об освобождении всех отдельных членов класса; вы можете заместить метод dealloc (наследуемый из NSObject) и освободить там память origin.

---

**Примечание.** Мы будем замещать метод dealloc, а не метод release. Метод release иногда освобождает память, используемую объектом, а иногда нет. Он освобождает память, используемую объектом, только если никто другой не обращается к этому объекту, и делает это, вызывая метод объекта dealloc, который фактически освобождает память.

---

При замещении метода dealloc вы должны проследить, чтобы была освобождена память, занимаемая не только вашими переменными экземпляра, но и всеми унаследованными переменными.

Для этого существует специальное ключевое слово `super`, которое обозначает родительский класс получателя сообщения. Для выполнения замещаемого метода нужно передать `super` сообщение. Выражение с сообщением

```
[super release];
```

при использовании внутри метода вызывает метод `release`, который определен (или унаследован) в родительском классе. Этот метод вызывается в получателе сообщения, то есть в себе самом (`self`).

Таким образом, замещение метода `dealloc` для класса `Rectangle` выполняется следующим образом. Сначала освобождается память, занятая `origin`, а затем вызывается метод `dealloc` из родительского класса. Тем самым освобождается память, занятая самим объектом `Rectangle`. Ниже приводится этот метод.

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

Определенный здесь метод `dealloc` не возвращает никакого значения. Внутри метода сначала `dealloc` выполняется проверка, что `origin` имеет ненулевое значение. Начало прямоугольника (`origin`), возможно, не было задано. В этом случае он имеет по умолчанию нулевое значение. Затем вызывается метод `dealloc` из родительского класса, который был бы унаследован классом `Rectangle`, если бы не был замещен.

Метод `dealloc` можно написать проще:

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

поскольку вы можете без проблем передать сообщение `nil`-объекту. Кроме того, к `origin` применяется `release`, а не `dealloc`. В любом случае, если никто другой не использует `origin`, `release` вызовет метод `dealloc` для `origin`, чтобы освободить его пространство.

Используя этот метод, вы должны освобождать только те объекты-прямоугольники, для которых выделили память, не заботясь об объектах `XYPoint`, которые они содержат. Двух сообщений, показанных в программе 8.5, теперь достаточно, чтобы освободить память для всех объектов, в том числе объекта `XYPoint`, создаваемого с помощью `setOrigin`:

```
[myRect release];
[myPoint release];
```

Правда, остается одна проблема. Если задать для начала прямоугольника (*origin*) одного объекта *Rectangle* другие значения во время выполнения программы, то вы должны освободить память, занятую прежним началом прямоугольника, прежде чем выделить и назначить новую. Рассмотрим следующую последовательность строк:

```
myRect.origin = startPoint;
myRect.origin = endPoint;
...
[startPoint release];
[endPoint release];
[myRect release];
```

Копия объекта *startPoint* класса *XYPoint*, сохраненная в элементе *origin* *myRect*, не будет освобождена, поскольку она перезаписывается вторым значением для *origin* (*endPoint*). Эта копия *origin* будет освобождена правильно, когда будет освобождаться сам объект прямоугольника, если применяется новый метод освобождения памяти.

Но вы должны сделать так, чтобы память, выделенная для предыдущего начала (*origin*), была освобождена до того, как будет задано новое начало. Это можно сделать в методе *setOrigin*:

```
-(void) setOrigin: (XYPoint *) pt
{
 if (origin)
 [origin release];
 origin = [[XYPoint alloc] init];
 [origin setX: pt.x andY: pt.y];
}
```

Если вы синтезируете свои методы доступа, то можете сделать так, чтобы компилятор автоматически разрешил эту проблему.

## Расширение через наследование: добавление новых переменных экземпляра

Вы можете добавлять не только новые методы, расширяя определение класса, но и новые переменные экземпляра. В обоих случаях получается накопительный эффект. При наследовании вы не можете удалять методы или переменные экземпляра. Переменные вы можете только добавлять, а методы – добавлять или замещать.

Вернемся к простым классам ClassA и ClassB и внесем некоторые изменения. Добавим в ClassB новую переменную экземпляра y.

```
@interface ClassB: ClassA
{
 int y;
}

-(void) printVar;
@end
```

Может показаться, что ClassB будет иметь только одну переменную экземпляра (с именем y), но, исходя из предыдущего объявления, на самом деле он имеет две. Он имеет свою собственную переменную экземпляра y и наследует переменную x из класса ClassA.

---

**Примечание.** Этот класс имеет также переменные экземпляра, которые наследуются из класса NSObject, но мы пока будем игнорировать этот факт.

---

Ниже приводится простой пример этой концепции (программа 8.8).

### Программа 8.8

```
// Расширение переменных экземпляра

#import <Foundation/Foundation.h>

// Объявление и определение ClassA

@interface ClassA: NSObject
{
 int x;
}

-(void) initVar;
@end

@implementation ClassA
-(void) initVar
{
 x = 100;
}
@end

// Объявление и определение ClassB

@interface ClassB: ClassA
```

```
{
 int y;
}
-(void) initVar;
-(void) printVar;
@end

@implementation ClassB
-(void) initVar
{
 x = 200;
 y = 300;
}

-(void) printVar
{
 NSLog (@"x = %i", x);
 NSLog (@"y = %i", y);
}
@end

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

 ClassB *b = [[ClassB alloc] init];

 [b initVar]; // uses overriding method in ClassB
 [b printVar]; // reveal values of x and y;

 [b release];

 [pool drain];
 return 0;
}
```

### Вывод программы 8.8

```
x = 200
y = 300
```

Объект b класса ClassB инициализируется путем вызова метода `initVar`, определенного в классе ClassB. Этот метод замещает метод `initVar` из ClassA, присваивает значение 200 переменной x (унаследованное из ClassA) и значение 300 – переменной y (определенной в ClassB). Метод `printVar` выводит значения этих переменных экземпляра.

Есть много других тонкостей, связанных с выбором метода в ответ на сообщение, особенно в тех случаях, когда получателем является один из нескольких классов. Это *динамическое связывание* (*dynamic binding*), которое рассматривается в следующей главе.

## Абстрактные классы

Мы закончим эту главу, добавив еще немного терминологии, связанной с понятием наследования.

Иногда классы создаются просто для того, чтобы упростить другим создание подкласса. Такие классы называют *абстрактными* (*abstract*) или *абстрактными суперклассами*. В этом классе определяются методы и переменные экземпляра, но при этом не предполагается, что кто-либо будет создавать экземпляры из этого класса. Например, р нет никакого смысла определить объект непосредственно из корневого объекта `NSObject`. Foundation framework (см. часть II) содержит несколько абстрактных классов. Например, класс Foundation `NSNumber` является абстрактным классом, который был создан для работы с числами как с объектами. Для хранения целых чисел и чисел с плавающей точкой обычно требуются разные размеры памяти. Для каждого числового типа существуют отдельные подклассы `NSNumber`. Поскольку эти подклассы, в отличие от их абстрактных суперклассов, реально существуют, их называют *конкретными* (*concrete*) подклассами. Каждый конкретный подкласс является дочерним классом класса `NSNumber` и называется *клusterом* (*cluster*). Когда вы отправляете сообщение классу `NSNumber` для создания нового объекта типа `integer`, используется соответствующий подкласс, чтобы выделить память для этого объекта и задать его значение. Эти подклассы на самом деле являются частными. Вы не можете выполнить непосредственный доступ к этим подклассам сами; доступ осуществляется косвенно через абстрактный суперкласс. Абстрактный суперкласс представляет общий интерфейс для работы со всеми типами числовых объектов и позволяет вам не знать, какой тип числа вы сохранили в вашем числовом объекте и как задавать и считывать его значение.

## Упражнения

1. Добавьте в программе 8.1 новый класс с именем `ClassC`, который является подклассом класса `ClassB`. Создайте метод `initVar`, который присваивает значение 300 своей переменной экземпляра `x`. Напишите тестовую процедуру, которая объявляет объекты классов `ClassA`, `ClassB` и `ClassC` и вызывает их методы `initVar`.
2. При работе с объектами высокого разрешения вам может потребоваться координатная система, которая позволяет задавать точки как значения с плавающей точкой вместо целых значений. Внесите изменения в классы `XYPoint` и `Rectangle`, чтобы можно было работать с числами с плавающей точкой. Для ширины, высоты, площади и периметра тоже должны использоваться числа с плавающей точкой.

3. Внесите изменения в программу 8.1, чтобы добавить новый класс с именем ClassB2, который, как и ClassB, является подклассом класса ClassA.

Что вы можете сказать о связи между ClassB и ClassB2? Укажите иерархические связи между классом NSObject, классом ClassA, ClassB и ClassB2.

Что является суперклассом для ClassB?

Что является суперклассом для ClassB2?

Сколько подклассов может иметь класс, и сколько он может иметь суперклассов?

4. Напишите метод Rectangle с именем translate:, который принимает в качестве своего аргумента вектор с именем XYPoint ( $x, y$ ). Сделайте так, чтобы он смещал начало прямоугольника (origin) на указанный вектор.
5. Определите новый класс с именем GraphicObject и сделайте его подклассом NSObject. Определите переменные экземпляра в этом новом классе следующим образом.

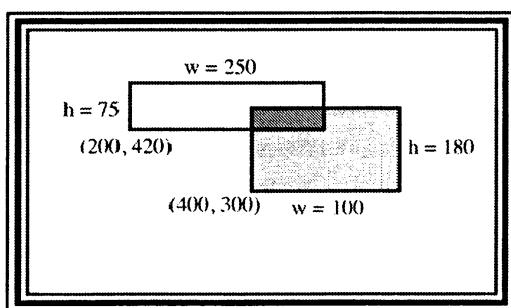
```
int fillColor; // 32-битный цвет
BOOL filled; // Заполняется ли объект?
int lineColor; // 32-битный цвет линии
```

Напишите методы, которые задают и считывают значения определенных выше переменных.

Сделайте класс Rectangle подклассом GraphicObject.

Определите новые классы Circle (Круг) и Triangle (Треугольник), которые тоже являются подклассами GraphicObject. Напишите методы, чтобы задавать и считываивать параметры для этих объектов, вычислять длину окружности и площадь круга, периметр и площадь треугольника.

6. Напишите метод для Rectangle с именем intersect:, который принимает прямоугольник как аргумент и возвращает прямоугольник, представляющий площадь перекрытия этих прямоугольников. Например, для двух прямоугольников, показанных на рис. 8.10, метод должен возвращать прямоугольник, начало (origin) которого находится в точке (400, 420), ширина (width) равна 50 и высота (height) равна 60.



**Рис. 8.10.** Пересечение прямоугольников

Если прямоугольники не пересекаются, нужно возвратить прямоугольник с шириной и высотой 0 и началом (0,0).

7. Напишите метод для класса `Rectangle` с именем `draw`, который рисует прямоугольник из знаков «минус» и «вертикальная черта». Следующая последовательность кодов

```
Rectangle *myRect = [[Rectangle alloc] init];
[myRect setWidth: 10 andHeight: 3];
[myRect draw];
[myRect release];
```



## Глава 9

# Полиморфизм, динамический контроль типов и динамическое связывание

В этой главе описываются возможности языка Objective-C, которые делают его столь мощным языком программирования и отличают его от некоторых объектно-ориентированных языков, таких как C++. Это три ключевых концепции: полиморфизм, динамический контроль типов и динамическое связывание. *Полиморфизм (polymorphism)* позволяет разрабатывать программы таким образом, чтобы объекты из различных классов могли определять методы, которые имеют одно и то же имя. *Динамический контроль типов (dynamic typing)* позволяет откладывать определение класса, которому принадлежит объект, до выполнения программы. *Динамическое связывание (dynamic binding)* позволяет откладывать определение конкретного метода для вызова в объекте до начала выполнения программы.

## Полиморфизм: одно имя, различные классы

В программе 9.1 показан файл секции interface для класса Complex, который используется для представления в программе комплексных чисел.

**Программа 9.1.** Файл секции interface Complex.h

```
// файл секции interface для класса Complex

#import <Foundation/Foundation.h>

@interface Complex: NSObject
{
 double real;
 double imaginary;
}
@property double real, imaginary;
-(void) print;
```

```

-(void) setReal: (double) a andImaginary: (double) b;
-(Complex *) add: (Complex *) f;

@end

```

В упражнении 6 главы 4 мы создали секцию `implementation` для этого класса. Мы добавили дополнительный метод `setReal:andImaginary:`, чтобы задавать вещественную и мнимую части числа с помощью одного сообщения, а также синтезировали методы доступа. Это показано в следующей программе.

### **Программа 9.1.** Файл секции `implementation` `Complex.m`

```

// Файл секции implementation для класса Complex

#import "Complex.h"
@implementation Complex

@synthesize real, imaginary;

-(void) print
{
 NSLog(@"%@", real, imaginary);
}

-(void) setReal: (double) a andImaginary: (double) b

{
 real = a;
 imaginary = b;
}

-(Complex *) add: (Complex *) f
{
 Complex *result = [[Complex alloc] init];

 [result setReal: real + [f real]
 andImaginary: imaginary + [f imaginary]];

 return result;
}
@end

```

### **Программа 9.1.** Тестовая программа `main.m`

```

// Совместно используемые имена методов: полиморфизм

#import "Fraction.h"

```

```
#import "Complex.h"

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

 Fraction *f1 = [[Fraction alloc] init];
 Fraction *f2 = [[Fraction alloc] init];
 Fraction *fracResult;
 Complex *c1 = [[Complex alloc] init];
 Complex *c2 = [[Complex alloc] init];
 Complex *compResult;

 [f1 setTo: 1 over: 10];
 [f2 setTo: 2 over: 15];

 [c1 setReal: 18.0 andImaginary: 2.5];
 [c2 setReal: -5.0 andImaginary: 3.2];

 // добавление и вывод 2 комплексных чисел

 [c1 print]; NSLog(@"%@", [c2 print]);
 NSLog(@"-----");
 compResult = [c1 add: c2];
 [compResult print];
 NSLog(@"\n");

 [c1 release];
 [c2 release];
 [compResult release];

 // добавление и вывод 2 дробей

 [f1 print]; NSLog(@"%@", [f2 print]);
 NSLog(@"----");
 fracResult = [f1 add: f2];
 [fracResult print];

 [f1 release];
 [f2 release];
 [fracResult release];

 [pool drain];
 return 0;
}
```

### Вывод программы 9.1

```
18 + 2.5i
+
-5 + 3.2i

13 + 5.7i
```

```
1/10
+
2/15

7/30
```

Отметим, что оба класса, `Fraction` и `Complex`, содержат методы `add:` и `print`. Но откуда система знает, какие методы нужно вызывать при выполнении следующих выражений с сообщениями?

```
compResult = [c1 add: c2];
[compResult print];
```

Системе выполнения (runtime) Objective-C известно, что `c1`, получатель первого сообщения, является объектом класса `Complex`. Поэтому выбирается метод `add:`, определенный для класса `Complex`. Система выполнения Objective-C определяет также, что `compResult` является объектом класса `Complex`, поэтому она выбирает метод `print`, определенный в классе `Complex`, чтобы вывести результат сложения. То же самое относится к следующим выражениям с сообщениями.

```
fracResult = [f1 add: f2];
[fracResult print];
```

---

**Примечание.** Система всегда содержит информацию о классе, которому принадлежит объект. Это позволяет ей принимать нужные решения во время выполнения, не во время компиляции. Подробнее об этом рассказывается в главе 13.

---

Выбор методов из класса `Fraction` выполняется при оценке выражения с сообщением в зависимости от класса `f1` и `fracResult`.

Возможность использования одного имени из различных классов называется полиморфизмом. Полиморфизм позволяет вам разрабатывать набор классов с одинаковым именем метода. В определение каждого класса включается код, необходимый для ответа на вызов данного определенного метода, и это делает его независимым от определений в других классах. Это позволяет также добавлять новые классы, которые могут отвечать на вызов методов с тем же именем.

**Примечание.** Именно классы Fraction и Complex (а не тестовая программа) должны предусматривать освобождение памяти, занимаемой результатами их методов add:. На самом деле эти объекты должны освобождаться автоматически (autorelease). Подробнее об этом рассказывается в главе 18.

## Динамическое связывание и тип id

В главе 4 уже говорилось, что тип данных `id` является обобщенным типом объекта. Это означает, что `id` используется для хранения объектов, которые принадлежат любому классу. Он используется для хранения в переменной различных типов объектов. Рассмотрим программу 9.2 и ее вывод.

### Программа 9.2

```
// Пример динамического контроля типов и динамического связывания

#import "Fraction.h"
#import "Complex.h"

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

 id dataValue;
 Fraction *f1 = [[Fraction alloc] init];
 Complex *c1 = [[Complex alloc] init];

 [f1 setTo: 2 over: 5];
 [c1 setReal: 10.0 andImaginary: 2.5];

 // в первый раз dataValue присваивается дробь (fraction)
 dataValue = f1;
 [dataValue print];

 // теперь dataValue присваивается комплексное число (complex)
 dataValue = c1;
 [dataValue print];

 [c1 release];
 [f1 release];

 [pool drain];
```

```

 return 0;
}

```

### Вывод программы 9.2

2/5  
10 + 2.5i

Переменная `dataValue` объявляется как объект типа `id`, поэтому `dataValue` можно использовать для хранения в программе объекта любого типа. Отметим, что в строке объявления не используется «звездочка»:

```
id dataValue;
```

Объекту `f1` типа `Fraction` присваивается дробь `2/5`, переменной `c1` типа `Complex` присваивается значение `(10 + 2.5i)`. Оператор

```
dataValue = f1;
```

сохраняет `f1` в `dataValue`. Вы можете вызвать с `dataValue` любой из методов, допустимых в объекте типа `Fraction`, хотя `dataValue` имеет тип `id`, а не `Fraction`. Но как система определяет, какой метод нужно вызывать, если в `dataValue` можно сохранять объект любого типа? Если система встречает выражение с сообщением

```
[dataValue print];
```

откуда она знает, какой метод `print` нужно вызвать? Ведь методы `print` определены и в классе `Fraction`, и в классе `Complex`.

Как уже говорилось, система Objective-C всегда следит за классом, которому принадлежит объект. Это также определяется концепциями динамического контроля типов и динамического связывания: система принимает решение о классе объекта и о методе для его динамического вызова во время выполнения, а не во время компиляции.

Поэтому во время выполнения программы, прежде чем передать `dataValue` сообщение `print`, система проверяет класс объекта, хранящегося в `dataValue`. В случае программы 9.2 эта переменная содержит объект типа `Fraction`, поэтому используется метод `print`, определенный в классе `Fraction`.

Во втором случае а `c1` типа `Complex` присваивается `dataValue`. Затем выполняется следующее выражение с сообщением:

```
[dataValue print];
```

Поскольку теперь `dataValue` содержит объект, принадлежащий классу `Complex`, для выполнения выбирается метод `print` из этого класса.

Это простой пример, но вы можете распространить эту концепцию на более сложные приложения. В сочетании с концепцией полиморфизма динамическое связывание и динамический контроль типов позволяют легко писать коды, которые передают одинаковое сообщение объектам из различных классов.

Например, метод `draw` можно использовать для рисования графических объектов на экране. У вас могут быть различные методы `draw`, определенные для каждого из ваших графических объектов, таких как тексты, окружности, пря-

моугольники, окна и т.д. Если графический объект, который нужно нарисовать, сохраняется, например, в переменной типа `id` с именем `currentObject`, то его можно нарисовать на экране, просто передав ему сообщение `draw`:

```
[currentObject draw];
```

Вы можете сначала проверить, отвечает ли на метод `draw` объект, хранящийся в `currentObject`. Ниже вы увидите, как это делать.

## Проверка на этапе компиляции и проверка на этапе выполнения

Поскольку тип объекта, хранящегося в переменной `id`, во время компиляции может быть неизвестен, некоторые проверки откладываются до выполнения программы (runtime). Рассмотрим следующую последовательность кодов.

```
Fraction *f1 = [[Fraction alloc] init];
[f1 setReal: 10.0 andImaginary: 2.5];
```

Поскольку метод `setReal:andImaginary:` применяется к комплексным числам, а не к дробям, при компиляции программы, содержащей эти строки, появится предупреждающее сообщение.

```
prog3.m: In function 'main':(В функции 'main')
prog3.m:13: warning: 'Fraction' does not respond to 'setReal:andImaginary:'
(предупреждение: объект типа 'Fraction' не отвечает на 'setReal:andImaginary:')
```

Компилятору Objective-C известно, что `f1` является объектом класса `Fraction`, поскольку он был объявлен именно так. При появлении выражения с сообщением

```
[f1 setReal: 10.0 andImaginary: 2.5];
```

ему стало известно, что класс `Fraction` не содержит метода `setReal:andImaginary:` (и не наследует его). Поэтому компилятор выдает предупреждающее сообщение.

Теперь рассмотрим следующую последовательность кодов.

```
id dataValue = [[Fraction alloc] init];
...
[dataValue setReal: 10.0 andImaginary: 2.5];
```

Компилятор не выводит предупреждающего сообщения, поскольку во время обработки исходного файла ему неизвестно, какой тип объекта сохраняется в `dataValue`.

Сообщение об ошибке не появится, пока вы не запустите программу, содержащую эти строки. Сообщение об ошибке может выглядеть следующим образом.

```
objc: Fraction: does not recognize selector -setReal:andImaginary:
(не распознается селектор -setReal:andImaginary)
```

```
dynamic3: received signal: Abort trap
(получен сигнал: аварийное прерывание)
When attempting to execute the expression
(При попытке выполнить выражение)
[dataValue setReal: 10.0 andImaginary: 2.5];
```

Система runtime сначала проверяет тип объекта, хранящегося внутри `dataValue`. Поскольку `dataValue` содержит дробь (объект `Fraction`), система runtime проверяет, определен ли метод `setReal:andImaginary:` для класса `Fraction`. Поскольку это не так, выдается сообщение, и выполнение программы прекращается.

## Тип данных `id` и статический контроль типов

Если тип данных `id` можно использовать для хранения любого объекта, так почему бы нам не объявлять все объекты с типом `id`? Этого не следует делать по нескольким причинам.

Во-первых, определяя переменную как объект из определенного класса, мы используем так называемый *статический контроль типов* (*static typing*). Слово *статический* означает, что переменная всегда используется для хранения объектов из определенного класса, поэтому класс объекта, хранящегося в этом типе, заранее определен, то есть является статическим. При использовании статического контроля типов компилятор обеспечивает согласованное использование этой переменной во всей программе. Компилятор может проверить, определен ли (или унаследован) метод, применяемый к объекту, и если нет, то выводит предупреждающее сообщение. Таким образом, если вы объявляете в своей программе переменную класса `Rectangle` с именем `myRect`, то компилятор проверяет, все ли методы, которые вы вызываете для `myRect`, определены в классе `Rectangle` или наследуются из его суперкласса.

---

**Примечание.** Определенные приемы позволяют вызывать методы, который указываются самой переменной, в таком случае компилятор не может выполнить проверку.

Но если проверка будет выполнена во время выполнения, почему вас должен интересовать статический контроль типов? Дело в том, что лучше выявить ошибки на этапе компиляции программы, чем на этапе выполнения. Если программу будет выполнять другой человек, вы не сможете увидеть ошибку. Это означает, что если программа введена в эксплуатацию, ничего не подозревающий пользователь может столкнуться с ошибкой, при которой указывается, что определенный объект не распознает некоторый метод.

Еще одним доводом к применению статического контроля типов является то, что он делает ваши программы более удобными для чтения. Рассмотрим объявление

```
id f1;
```

и сравним его с

```
Fraction *f1;
```

Конечно, второе объявление лучше, поскольку в нем указывается предполагаемое использование переменной `f1`. Сочетание статического контроля типов и осмысленных имен переменных позволяет делать программы самодокументирующими.

## Типы аргументов и возвращаемых значений при динамическом контроле типов

Если для вызова метода используется динамический контроль типов, соблюдайте следующее правило. Если методы с одинаковым именем реализованы в нескольких классах, каждый метод должен быть согласован с типом каждого аргумента и типом возвращаемого значения, чтобы компилятор мог генерировать правильный код для выражений с сообщениями.

Компилятор выполняет проверку на согласованность в объявлениях каждого класса, которые он встречает. Если один или несколько методов не согласуются с типом аргумента или возвращаемого значения, компилятор выводит предупреждающее сообщение. Например, оба класса, `Fraction` и `Complex`, содержат метод `add:`, но класс `Fraction` принимает в качестве аргумента и возвращает объект типа `Fraction`, а класс `Complex` – объект типа `Complex`. Если `frac1` и `myFract` – объекты типа `Fraction`, а `comp1` и `myComplex` – объекты типа `Complex`, то определения

```
result = [myFract add: frac1];
```

и

```
result = [myComplex add: comp1];
```

не вызовут проблемы. В обоих случаях получатель сообщения доступен для статического контроля типов и компилятор может проверить согласованность при использовании метода, поскольку он определен в классе получателя.

Если `dataValue1` и `dataValue2` – переменные типа `id`, то выражение

```
result = [dataValue1 add: dataValue2];
```

заставляет компилятор генерировать код для передачи данного аргумента методу `add:` и обработки его возвращаемого значения, делая некоторые предположения.

Во время выполнения система `runtime Objective-C` проверит конкретный класс объекта, хранящегося в `dataValue1`, и выберет метод из подходящего класса для выполнения. Однако в более общем случае компилятор может генерировать неверный код для передачи методу аргументов или обработки его возвращаемого значения. Это может произойти, например, если один метод принял в качестве аргумента какой-либо объект, а другой – значение с плавающей точкой, или один метод вернул объект, а другой – целое значение. Если методы отличаются только типом объекта (например, метод `add:` класса `Fraction` принимает в качестве аргумента и возвращает объект типа `Fraction`, а метод `add:` класса

```

action = @selector (draw);
...
[graphicObject performSelector: action];

```

Метод, указанный переменной `action` типа `SEL`, передается графическому объекту, который хранится в `graphicObject`. Предполагается, что действие (`action`) может изменяться во время выполнения программы в зависимости от ввода пользователя, несмотря на указанное действие `draw`. Чтобы убедиться, что объект может ответить на данное действие, нужно использовать, например, следующую последовательность.

```

if ([graphicObject respondsToSelector: action] == YES)
 [graphicObject perform: action]
else
 // здесь должен быть код для обработки ошибок

```

**Примечание.** Вы можете отлавливать ошибку, переопределив метод `doesNotRecognize:`. Этот метод вызывается, когда классу передано нераспознаваемое сообщение и в качестве аргумента передан нераспознаваемый селектор.

Можно применять и другие стратегии. Например, с помощью метода `forward::` можно перенаправлять сообщение для обработки. Всегда можно передавать метод и перехватывать возникшую исключительную ситуацию. Мы рассмотрим этот способ чуть позже.

В программе 9.3 задаются некоторые вопросы о классах `Square` и `Rectangle`, определенных в главе 8. Постарайтесь предсказать результаты этой программы.

### Программа 9.3

```

#import "Square.h"

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

 Square *mySquare = [[Square alloc] init];

 // Проверка с помощью isMemberOf:

 if ([mySquare isMemberOfClass: [Square class]] == YES)
 NSLog (@"mySquare is a member of Square class");

 if ([mySquare isMemberOfClass: [Rectangle class]] == YES)
 NSLog (@"mySquare is a member of Rectangle class");

 if ([mySquare isMemberOfClass: [NSObject class]] == YES)

```

```
 NSLog(@"mySquare is a member of NSObject class");

// Проверка с помощью isKindOfClass:
if ([mySquare isKindOfClass: [Square class]] == YES)
 NSLog(@"mySquare is a kind of Square");

if ([mySquare isKindOfClass: [Rectangle class]] == YES)
 NSLog(@"mySquare is a kind of Rectangle");

if ([mySquare isKindOfClass: [NSObject class]] == YES)
 NSLog(@"mySquare is a kind of NSObject");

// Проверка с помощью respondsToSelector:
if ([mySquare respondsToSelector: @selector (setSide:)] == YES)
 NSLog(@"mySquare responds to setSide: method");

if ([mySquare respondsToSelector: @selector (setWidth:andHeight:)] == YES)
 NSLog(@"mySquare responds to setWidth:andHeight: method");

if ([Square respondsToSelector: @selector (alloc)] == YES)
 NSLog(@"Square class responds to alloc method");

// Проверка с помощью instancesRespondToSelector:
if ([Rectangle instancesRespondToSelector: @selector (setSide:)] == YES)
 NSLog(@"Instances of Rectangle respond to setSide: method");

if ([Square instancesRespondToSelector: @selector (setSide:)] == YES)
 NSLog(@"Instances of Square respond to setSide: method");

if ([Square isSubclassOfClass: [Rectangle class]] == YES)
 NSLog(@"Square is a subclass of a rectangle");

[mySquare release];

[pool drain];
return 0;
}
```

Эта программа должна быть создана с файлами секции implementation для классов Square, Rectangle и XYPoint, которые были рассмотрены в главе 8.

### Вывод программы 9.3

```
mySquare is a member of Square class (mySquare является членом класса Square)
mySquare is a kind of Square (mySquare происходит из Square)
mySquare is a kind of Rectangle
mySquare is a kind of NSObject
mySquare responds to setSide: method (mySquare отвечает на метод setSide:)
mySquare responds to setWidth:andHeight: method
Square class responds to alloc method (Square отвечает на метод alloc)
Instances of Square respond to setSide: method (Экземпляры mySquare отвечают на
метод setSide:)
Square is a subclass of a rectangle (Square - это подкласс класса Rectangle)
```

Вывод программы 9.3 вполне понятен. `isMemberOfClass:` проверяет непосредственное членство в классе, а `isKindOfClass:` проверяет членство в иерархии наследования. `mySquare` является членом класса `Square` и «происходит» из `Square`, `Rectangle` и `NSObject`, поскольку входит в иерархию этого класса (очевидно, что все объекты должны возвращать значение `YES` для проверки `isKindOfClass:` по классу `NSObject`, если только вы не определили новый корневой объект).

В проверке

```
if ([Square respondsToSelector: @selector (alloc)] == YES)
```

определяется, отвечает ли класс `Square` на метод класса `alloc`. Это так, поскольку он унаследован из корневого объекта `NSObject`. Вы всегда можете использовать непосредственно имя класса в качестве получателя в выражении для сообщения и не обязаны включать

`[Square class]`

в предыдущее выражение (хотя могли бы это сделать).

Но это единственное место, где можно без этого обойтись. В других местах для получения объекта-класса нужно обязательно применять метод `class`.

## Обработка исключительных ситуаций с помощью `@try`

Практика надежного программирования обязывает предусматривать проблемы, которые могут возникнуть в программе. Это можно делать, проверяя состояния, которые могут вызвать аварийное завершение программы, и включая обработку этих ситуаций, например, выводя сообщение и корректно завершая программу. Например, выше в этой главе было показано, как проверить, отвечает ли объект на определенное сообщение. Эта проверка позволяет избежать отправки нераспознаваемого сообщения. Обычно при попытке отправки нераспознаваемого сообщения программа сразу прекращает свою работу, выдавая так называемую *исключительную ситуацию*, или *исключение (exception)*.

Рассмотрим программу 9.4. В определении класса Fraction у нас не было метода с именем noSuchMethod («нет такого метода»). При компиляции этой программы вы получите из-за этого предупреждающие сообщения.

#### Программа 9.4

```
#import "Fraction.h"
int main (int argc, char *argv [])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Fraction *f = [[Fraction alloc] init];
 [f noSuchMethod];
 NSLog (@"Execution continues!");
 [f release];
 [pool drain];
 return 0;
}
```

Несмотря на эти предупреждающие сообщения, вы можете попытаться продолжить работу и выполнить программу. В этом случае программа будет аварийно завершена с выводом следующих сообщений об ошибках.

#### Вывод программы 9.4

```
-[Fraction noSuchMethod]: unrecognized selector sent to instance 0x103280
(... нераспознанный селектор передан экземпляру)
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
(Прекращение работы приложения из-за необработанного исключения)
reason: '*** -[Fraction noSuchMethod]: unrecognized selector sent
to instance 0x103280'
(причина: ... нераспознанный селектор передан экземпляру)
Stack: (
2482717003,
2498756859,
2482746186,
2482739532,
2482739730

)
Trace/BPT trap
```

Чтобы избежать аварийного завершения программы, можно поместить один или несколько операторов в блоке операторов, имеющим следующий формат.

```
@try {
 оператор
 оператор
```

```

 ...
}

@catch (NSEException *exception) {
 оператор
 оператор
 ...
}

```

Выполнение программы в блоке @try происходит как обычно. Однако если один из операторов в этом блоке выдает исключение, работа программы не прекращается, а управление передается в блок @catch, где продолжается ее выполнение. Внутри этого блока можно обрабатывать исключение. Последовательность действий в этом случае может быть вывод сообщения об ошибке, очистка и завершение работы программы.

В программе 9.5 показана обработка исключения. Затем приводится вывод программы.

#### Программа 9.5. Обработка исключения

```

#import "Fraction.h"

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

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

 Fraction *f = [[Fraction alloc] init];

 @try {
 [f noSuchMethod];
 }
 @catch (NSEException *exception) {
 NSLog(@"Caught %@%@", [exception name], [exception reason]);
 }
 NSLog(@"Execution continues!");
 [f release];
 [pool drain];
 return 0;
}

```

#### Вывод программы 9.5

```

*** -[Fraction noSuchMethod]: unrecognized selector sent to instance
0x103280
Caught NSInvalidArgumentException: *** -[Fraction noSuchMethod]:
unrecognized selector sent to instance 0x103280

```

Execution continues! (Выполнение продолжается!)

Если возникает исключительная ситуация, выполняется блок @catch. Объект `NSError`, который содержит информацию об исключении, передается в этот блок как аргумент. Метод `name` считывает имя исключения, а метод `reason` указывает причину (которую система runtime раньше выводила автоматически).

После выполнения последнего оператора в блоке @catch (здесь только один оператор) программа продолжает выполнение, начиная с оператора, который непосредственно следует за этим блоком. В данном случае мы выполняем вызов `NSLog`, чтобы подтвердить, что выполнение не было прекращено.

Это очень простой пример, показывающий, как перехватывать исключения в программе. Можно также использовать блок @finally, чтобы включить код, выполняемый независимо от возникновения исключительной ситуации в каком-либо операторе блока @try.

Директива @throw позволяет создавать ваше собственное исключение. Ее можно использовать для создания конкретного исключения или внутри блока @catch для создания той же исключительной ситуации, которая вызвала переход в этот блок:

```
@throw;
```

Это может потребоваться после вашей собственной обработки исключения (например, после выполнения операций очистки). После этого вы можете передать системе остальную часть работы. И, наконец, у вас может быть несколько блоков @catch, которые следуют в определенном порядке для перехвата и обработки исключений различного типа.

## Упражнения

- Что произойдет, если вставить выражение с сообщением

```
[compResult reduce];
```

в программу 9.1 после того, как выполнено сложение (но до выполнения `release` для `compResult`)? Попробуйте и посмотрите, что получится.

- Можно ли переменной `dataValue` типа `id` (определенной в программе 9.2) присвоить объект класса `Rectangle` в соответствии с его определением в главе 8? Иначе говоря, является ли допустимым оператор

```
dataValue = [[Rectangle alloc] init];
```

Почему?

- Добавьте метод `print` к классу `XYPoint`,енному в главе 8. Он должен выводить точку в формате (x,y). Затем внесите изменения в программу 9.2, чтобы включить объект типа `XYPoint`. Эта модифицированная программа должна создавать объект типа `XYPoint`, задавать его значение, присваивать его переменной `dataValue` типа `id` и затем выводить его значение.
- Вспомните, что говорилось в этой главе о типах аргументов и возвращаемых значений, и модифицируйте методы `add:` в классах `Fraction` и `Complex`, что-

бы принимать и возвращать объекты типа `id`. Затем напишите программу, которая включает следующую последовательность кода.

```
result = [dataValue1 add: dataValue2];
[result print];
```

Здесь `result`, `dataValue1` и `dataValue2` – это объекты типа `id`. Не забудьте задать образом значения `dataValue1` и `dataValue2` в программе и освободить (`release`) все объекты, прежде чем завершить программу.

---

**Примечание.** Вам придется изменить имна этих методов. Системный класс `NSObjectController` тоже содержит метод `add:`. Как говорилось выше в разделе «Типы аргументов и возвращаемых значений при динамическом контроле типов», в случае существования нескольких методов с одним именем в разных классах, если на этапе компиляции тип получателя неизвестен, компилятор выполняет проверку согласованности типов аргументов и возвращаемого значения с методами, имеющими одинаковые имена.

---

## 5. Используя определения классов `Fraction` и `Complex`, заданные в этой книге, и определения

```
Fraction *fraction = [[Fraction alloc] init];
Complex *complex = [[Complex alloc] init];
id number = [[Complex alloc] init];
```

определите возвращаемое значение для следующих выражений с сообщениями. Затем введите их в программу, чтобы проверить результаты.

```
[fraction isKindOfClass: [Complex class]];
[complex isKindOfClass: [NSObject class]];
[complex isKindOfClass: [NSObject class]];
[fraction isKindOfClass: [Fraction class]];
[fraction respondsToSelector: @selector (print)];
[complex respondsToSelector: @selector (print)];
[Fraction instancesRespondToSelector: @selector (print)];
[number respondsToSelector: @selector (print)];
[number isKindOfClass: [Complex class]];
[number respondsToSelector: @selector (release)];

[[number class] respondsToSelector: @selector (alloc)];
```

# Глава 10

## Более подробно о переменных и типах данных

В этой главе мы поговорим об области действия переменных, методах инициализации для объектов и типах данных. В главе 7 мы кратко обсуждали область действия переменных экземпляра, статические и локальные переменные. Теперь мы более подробно поговорим о статических переменных и введем понятие глобальных и внешних переменных. Для компилятора Objective-C можно задавать директивы, позволяющие контролировать область действия переменных экземпляра. В этой главе мы рассмотрим их.

*Перечислимый (enumerated)* тип данных позволяет определять имя для типа данных, которое будет использоваться только для хранения заданного списка значений. В языке Objective-C оператор `typedef` позволяет вам назначать собственное имя встроенному или производному типу данных. В этой главе мы описываем действия компилятора по преобразованию типов данных при оценке выражений.

## Инициализация классов

Мы уже встречали такой набор действий, когда выделяется память для нового экземпляра объекта, а затем выполняется его инициализация:

```
Fraction *myFract = [[Fraction alloc] init];
```

После вызова этих методов обычно выполняется присваивание некоторых значений новому объекту:

```
[myFract setTo: 1 over: 3];
```

Процесс инициализации объекта, после которого ему присваиваются начальные значения, часто объединяют в один метод. Например, можно определить метод `initWith::`, который инициализирует объект типа `fraction` (дробь) и присваивает два (неименованных) заданных аргумента его числителю (`numerator`) и знаменателю (`denominator`).

Класс, который содержит много методов и переменных экземпляра, обычно имеет несколько методов инициализации. Например, класс `NSArray` из Foundation framework содержит шесть методов инициализации.

```
initWithArray:
initWithArray:copyItems:
initWithContentsOfFile:
initWithContentsOfURL:
initWithObjects:
initWithObjects:count:
```

Массиву (`NSArray`) можно выделить память и затем инициализировать его, например, с помощью следующей последовательности:

```
myArray = [[NSArray alloc] initWithArray: myOtherArray];
```

Принято, что все инициализаторы в классе обычно начинаются с `init`. Инициализаторы `NSArray` следуют этому правилу. При написании инициализаторов вы можете придерживаться одной из двух стратегий.

Если ваш класс содержит более одного инициализатора, один из них должен быть вашим *назначенным* (*designated*) инициализатором, и все остальные методы инициализации должны его использовать. Обычно это более сложный метод инициализации (и принимает больше всего параметров). При создании назначенного инициализатора основной код инициализации объединяется в одном методе. При создании подкласса можно затем замещать назначенный инициализатор, чтобы обеспечить правильную инициализацию новых экземпляров.

Необходимо следить за тем, чтобы правильно инициализировались любые наследуемые переменные экземпляра. Наиболее простой способ – вызывать сначала назначенный метод инициализации из родительского класса, который обычно называется `init`, а после этого инициализировать свои собственные переменные экземпляра.

Исходя из этого, метод инициализации `initWith::` для класса `Fraction` может выглядеть следующим образом.

```
-(Fraction *) initWith: (int) n: (int) d
{
 self = [super init];

 if (self)
 [self setTo: n over: d];

 return self;
}
```

Этот метод вызывает сначала родительский инициализатор, которым является метод `init` из `NSObject` (напомним, что это родительский класс для `Fraction`). Вы должны присвоить результат `self`, поскольку инициализатор имеет право изменять или перемещать объект в памяти.

После инициализации `super` (и ее успешного завершения, что указывается ненулевым возвращаемым значением) используется метод `setTo:over:`, чтобы задать числитель (`numerator`) и знаменатель (`denominator`) дроби (`Fraction`). Как и для других методов инициализации, предполагается, что вы возвращаете инициализированный объект.

В программе 10.1 выполняется проверка нового метода инициализации `initWith::`.

### Программа 10.1

```
#import "Fraction.h"
int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 Fraction *a, *b;

 a = [[Fraction alloc] initWith: 1: 3];
 b = [[Fraction alloc] initWith: 3: 7];

 [a print];
 [b print];

 [a release];
 [b release];

 [pool drain];
 return 0;
}
```

### Выход программы 10.1

1/3

3/7

Когда программа начинает выполнение, она передает метод вызова инициализации всем нашим классам. Если имеется класс и связанный с ним подкласс, родительский класс получает это сообщение первым. Это сообщение передается каждому классу только один раз, и оно гарантированно отправляется до того, как любые другие сообщения будут переданы классу. Ваша цель в этот момент — выполнение инициализации любого класса. Например, вам может потребоваться инициализация некоторых статических переменных, связанных с данным классом.

## Снова об области действия

На область действия переменных в программе можно влиять разными способами: с переменными экземпляра или с обычными переменными, определен-

ными вне или внутри функций. Ниже мы будем использовать термин *модуль* (*module*) при ссылке на любое число определений методов или функций, содержащихся в одном исходном файле.

## Директивы для управления областью действия переменных экземпляра

Вы уже знаете, что переменные экземпляра имеют область действия, которая ограничивается методами экземпляра, определенными для данного класса. Поэтому любой метод экземпляра может выполнять доступ к своим переменным экземпляра по имени без дополнительных действий.

Вы также знаете, что переменные экземпляра наследуются подклассом. Доступ к переменным экземпляра тоже можно выполнять по имени из любого метода, определенного в этом подклассе. И в этом случае специальные действия тоже не требуются.

Перед переменными экземпляра при объявлении в секции *interface* можно помещать четыре директивы, чтобы более точно управлять их областью действия.

- **@protected.** Методы, определенные в данном классе и любых подклассах, могут выполнять непосредственный доступ к последующим переменным экземпляра. Это вариант по умолчанию.
- **@private.** Методы, определенные в данном классе (но не в подклассах), могут выполнять непосредственный доступ к последующим переменным экземпляра.
- **@public.** Методы, определенные в данном классе и любых классах или модулях, могут выполнять непосредственный доступ к последующим переменным экземпляра.
- **@package.** Для 64-битных образов доступ к переменной экземпляра может выполняться в любом месте образа, который реализует данный класс.

Если вам нужно определить класс с именем *Printer*, содержащий две частные переменные экземпляра с именами *pageCount* и *tonerLevel*, которые доступны только методам из класса *Printer*, то вы можете использовать следующую секцию *interface*.

```
@interface Printer: NSObject
{
 @private
 int pageCount;
 int tonerLevel;
 @protected
 // другие переменные экземпляра
}
...
@end
```

Доступ к этим двум переменным экземпляра нельзя выполнить из любого подкласса класса Printer, поскольку они сделаны частными (*private*).

Эти специальные директивы действуют как переключатели: все переменные, которые появляются после одной из этих директив (пока не появится правая фигурная скобка, которая является концом объявлений этих переменных), имеют указанную область действия, если не использована другая директива. В приведенном примере директива `@protected` гарантирует, что следующие после нее переменные экземпляра будут доступны для методов подклассов и класса Printer.

Директива `@public` делает переменные экземпляра доступными для других методов или функций с помощью оператора-указателя (`->`), который описывается в главе 13. Такой доступ к переменным экземпляра не допускается практикой надежного программирования, поскольку он нарушает концепцию инкапсуляции данных (то есть скрытие классом своих переменных экземпляра).

## Внешние переменные

Если написать

```
int gMoveNumber = 0;
```

в начале программы — вне любого метода, определения класса или функции, — то ее значение можно использовать из любого места данного модуля. В этом случае `gMoveNumber` определяется как *глобальная* переменная. Обычно принято использовать букву `g` как первую букву глобальной переменной, чтобы обозначить для читателя программы ее область действия.

На самом деле такое определение переменной `gMoveNumber` делает ее значение доступным из других файлов. Приведенный оператор определяет переменную `gMoveNumber` не только как глобальную переменную, но и как *внешнюю глобальную* переменную.

*Внешней (external)* переменной является переменная, чье значение может изменяться другими методами или функциями. Внутри модуля, из которого требуется доступ к этой переменной, она объявляется обычным образом, перед ее объявлением ставится ключевое слово `extern`. Это указывает системе, что требуется доступ к глобально определенной переменной из другого файла. Ниже показан пример объявления переменной `gMoveNumber` как внешней переменной.

```
extern int gMoveNumber;
```

Модуль, в котором появилось это объявление, может выполнять доступ к переменной `gMoveNumber` и изменять ее значение. Другие модули тоже могут выполнять доступ к значению `gMoveNumber`, используя в своем файле аналогичное объявление `extern`.

При работе с внешними переменными соблюдайте следующее важное правило. Такая переменная должна быть определена среди ваших исходных файлов. Она должна быть объявлена вне любого метода или функции, и перед ней не должно быть ключевого слова `extern`, например,

Дополнительно такой переменной может быть присвоено начальное значение, как показано выше.

Второй способ определения внешней переменной – это объявление переменной вне любой функции с ключевым словом `extern` перед этим объявлением и явным присваиванием начального значения этой переменной, например,

```
extern int gMoveNumber = 0;
```

Однако при таком способе компилятор предупредит вас, что вы объявили переменную как `extern` и одновременно присвоили ей значение. Использование слова `extern` делает его объявлением, а не определением для переменной, а объявление не вызывает выделения памяти для переменной – это происходит в результате определения. В приведенном примере это правило нарушается, поскольку объявление интерпретируется как определение (поскольку переменной присваивается начальное значение).

При работе с внешними переменными можно объявлять переменную как `extern` во многих местах, но определить ее можно только один раз.

Рассмотрим программу как пример использования внешних переменных. Мы определили класс с именем `Foo` и ввели следующий код в файл `main.m`:

```
#import "Foo.h"

int gGlobalVar = 5;

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Foo *myFoo = [[Foo alloc] init];
 NSLog(@"%@", gGlobalVar);
 [myFoo setgGlobalVar: 100]

 NSLog(@"%@", gGlobalVar);
 [myFoo release];
 [pool drain];
 return 0;
}
```

Определение глобальной переменной `gGlobalVar` в этой программе делает ее значение доступным для любого метода (или функции), где используется соответствующее объявление `extern`. Предположим, что используется метод класса `Foo` с именем `setgGlobalVar`:

```
-(void) setgGlobalVar: (int) val
```

```
{
 extern int gGlobalVar;
 gGlobalVar = val;
```

```
}
```

Программа выведет следующие результаты.

```
5
100
```

Таким образом, метод `setGlobalVar`: может выполнять доступ к внешней переменной `gGlobalVar` и изменять ее значение.

В тех случаях, когда доступ к значению `gGlobalVar` требуется многим методам, проще включить объявление `extern` только один раз в начале файла. Но если доступ к этой переменной требуется одному методу или небольшому числу методов, то имеет смысл включать объявления `extern` в каждый из таких методов; это сделает программу более организованной и ограничит использование конкретной переменной теми функциями, в которых она действительно требуется. Отметим, что если переменная определена внутри файла, содержащего код, который выполняет доступ к этой переменной, то отдельные объявления `extern` не требуются.

## Статические переменные

Только что показанный пример нарушает принцип инкапсуляции данных и практику надежного объектно-ориентированного программирования. Но вам может потребоваться работа с переменными, значения которых должны быть доступны для вызова из различных методов. Может показаться, что нет смысла делать переменную `gGlobalVar` переменной экземпляра в классе `Foo`, надежнее «скрыть» ее в классе `Foo` путем ограничения доступа к ней методами-установщиками (`setter`) и методами-получателями (`getter`), определенными для этого класса.

Теперь вы знаете, что любая переменная, определенная вне метода, является не только глобальной, но и внешней переменной. Но существует много ситуаций, когда нужно определить переменную, которая является глобальной, но не является внешней. Иначе говоря, вам нужно определить глобальную переменную, которая будет локальной для определенного модуля (файла). Имеет смысл определять такую переменную, если доступ к ней требуется только тем методам, которые содержатся в конкретном определении класса. Это можно сделать, определив переменную как *статическую* (*static*) внутри файла, содержащего секцию `implementation` для конкретного класса.

Если следующий оператор помещен вне любого метода (или функции), то значение `gGlobalVar` будет доступно из любой последующей точки файла, содержащей это определение, но будет недоступно из методов или функций, содержащихся в других файлах.

```
static int gGlobalVar = 0;
```

Напомним, что методы класса не имеют доступа к переменным экземпляра (подумайте, почему это относится к данному случаю). Но вам может потребоваться, чтобы какой-либо метод класса имел доступ к переменным и мог задавать их значения. В качестве простого примера можно указать метод класса, выделяющий память для объектов (`alloc`), который должен следить за числом

объектов. Это можно сделать, создав статическую переменную внутри файла секции `implementation` для этого класса. Метод, выделяющий память для объектов, может выполнять непосредственный доступ к этой переменной, поскольку она не будет переменной экземпляра. Пользователям данного класса не обязательно знать об этой переменной. Поскольку она определена как статическая переменная в файле секции `implementation`, ее область действия будет ограничена этим файлом, поэтому пользователи не будут иметь непосредственного доступа к этой переменной и концепция инкапсуляции данных не будет нарушена. Вы можете написать метод для считывания значения этой переменной, если требуется доступ извне этого класса.

В программе 10.2 определение класса `Fraction` расширяется за счет добавления двух новых методов. Метод класса `allocF` выделяется память для нового объекта типа `Fraction` и следит за числом дробей (объектов `Fraction`), которые он выделил, а метод `count` возвращает значение этого счетчика. Метод `count` тоже является методом класса. Его можно было бы реализовать как метод экземпляра, но лучше запросить класс, сколько экземпляров он выделил, вместо передачи сообщения определенному экземпляру этого класса.

Ниже приводятся объявления для двух новых методов класса, добавленные в файл `Fraction.h`.

```
+ (Fraction *) allocF;
+(int) count;
```

Отметим, что здесь не замещается наследуемый метод `alloc`; вместо этого определяется наш собственный метод выделения памяти. В этом методе будет использоваться наследуемый метод `alloc`. Следующий код нужно поместить в файл секции `implementation` `Fraction.m`.

```
static int gCounter;

@implementation Fraction

+(Fraction *) allocF
{
 extern int gCounter;
 ++gCounter;

 return [Fraction alloc];
}

+(int) count
{
 extern int gCounter;

 return gCounter;
}
```

```
// здесь находятся другие методы из класса Fraction
...
@end
```

**Примечание.** В практике надежного программирования не принято замещать метод alloc, поскольку он работает с физическим местоположением в памяти. Не следует вмешиваться в работу системы на этом уровне.

Объявление static для переменной gCounter делает ее доступной для любого метода, определенного в секции implementation, но при этом она недоступна вне этого файла. Метод allocF просто наращивает значение переменной gCounter и затем использует метод alloc для создания новой дроби (Fraction), возвращая результат. Метод count просто возвращает значение счетчика (gCounter), не давая пользователю непосредственный доступ к этой переменной.

Напомним, что объявления extern не требуются в этих методах, поскольку переменная gCounter определена внутри этого файла. Это просто помогает читателю метода понять, что выполняется доступ к переменной, определенной вне метода. Префикс g в имени переменной предназначен для той же цели, поэтому большинство программистов обычно не включают объявления extern.

В программе 10.2 выполняется тестирование этих методов.

## Программа 10.2

```
#import "Fraction.h"

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

 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Fraction *a, *b, *c;

 NSLog (@"Fractions allocated: %i", [Fraction count]);

 a = [[Fraction allocF] init];
 b = [[Fraction allocF] init];
 c = [[Fraction allocF] init];

 NSLog (@"Fractions allocated: %i", [Fraction count]);
 [a release];
 [b release];
 [c release];

 [pool drain];
 return 0;
}
```

## Вывод программы 10.2

```
Fractions allocated: 0 (Выделено объектов Fraction)
```

```
Fractions allocated: 3
```

Когда начинается выполнение этой программы, значение `gCounter` автоматически задается равным 0 (напомним, что вы можете замещать наследуемый метод инициализации класса, если хотите выполнить специальную инициализацию класса в целом, например, присвоить статическим переменным некоторые ненулевые значения). После выделения (и последующей инициализации) трех объектов типа `Fraction` с помощью метода `allocF` метод `count` считывает переменную `counter`, значение которой действительно стало равным 3. Вы можете добавить метод-установщик (`setter`) для этого класса, если хотите выполнять сброс счетчика или задавать для него определенное значение, но в данном случае это не требуется.

## Описатели хранения для класса

Вы уже познакомились с описателями хранения для класса, которые можно помещать перед именами переменных. Здесь мы рассмотрим другие описатели, которые предоставляют компилятору информацию о предполагаемом использовании переменной в программе.

### auto

Это ключевое слово используется для объявления автоматической локальной переменной (в противоположность статической). Это описатель по умолчанию для объявления переменной внутри функции или метода, но никто не использует его в явном виде.

```
auto int index;
```

Это объявление переменной `index` как автоматической локальной переменной; это означает, что для нее выполняется автоматическое выделение памяти при входе в блок (которым может быть заключенная в фигурные скобки последовательность операторов, метод или функция) и автоматическое освобождение при выходе из этого блока. Поскольку это происходит по умолчанию внутри блока, оператор

```
int index;
```

эквивалентен оператору

```
auto int index;
```

В отличие от статических переменных, которые имеют по умолчанию начальные значения 0, автоматические переменные остаются неопределенными, пока вы не присвоите им значения в явном виде.

## const

Компилятор позволяет связывать атрибут `const` с переменными, значения которых не будут изменяться. Иначе говоря, он сообщает компилятору, что указанные переменные имеют постоянное значение при выполнении программы. Если попытаться присвоить значение переменной, объявленной с атрибутом `const`, после инициализации, или попытаться выполнить ее наращивание или уменьшение, компилятор выдаст предупреждающее сообщение. Ниже в переменной `pi` используется атрибут `const`:

```
const double pi = 3.141592654;
```

Он указывает компилятору, что программа не будет изменять эту переменную. Естественно, значение такой переменной должно быть инициализировано при ее определении.

Определение переменной с атрибутом `const` указывает читателю, что программа не будет изменять значение этой переменной.

## volatile

Это описатель, противоположный `const`. Он в явном виде указывает компилятору, что соответствующая переменная *будет изменять свое значение*. Он включен в язык Objective-C, чтобы компилятор не оптимизировал операторы, которые кажутся избыточными, и выполнял повторную проверку переменной, когда ее значение, казалось бы, не изменяется. Типичный пример – порт ввода-вывода, (I/O) (подробнее см. главу 13).

Предположим, что у вас имеется адрес выходного порта, хранящегося в программе в переменной с именем `outPort`. Записать в этот порт два символа, например `O` и `N`, можно с помощью следующего кода.

```
*outPort = 'O';
*outPort = 'N';
```

В первой строке указывается, что символ `O` нужно сохранить по адресу памяти, указанному переменной `outPort`. Во второй строке указывается, что символ `N` нужно сохранить по тому же адресу. Оптимизирующий компилятор может заметить, что это две последовательные записи по одному адресу, и поскольку `outPort` между ними не изменяется, может просто удалить первый оператор из программы. Чтобы этого не произошло, нужно объявить переменную `outPort` с атрибутом `volatile`, например:

```
volatile char *outPort;
```

## Перечислимые типы данных

Язык Objective-C позволяет задавать диапазон значений, которые могут быть присвоены переменной. Определение перечислимого типа данных инициируется с помощью ключевого слова `enum`. Сразу после этого ключевого слова сле-

дует имя перечислимого типа данных и затем список идентификаторов (заключенных в фигурные скобки), которые определяют допустимые значения для этого типа. В следующем операторе определяется тип данных `flag`:

```
enum flag { false, true };
```

Теоретически этому типу данных внутри программы могут быть присвоены только значения `true` и `false`. К сожалению, компилятор Objective-C не выдает предупреждающего сообщения, если это правило нарушается.

Чтобы объявить переменную типа `enum flag`, нужно снова использовать ключевое слово `enum`, после которого следует имя перечислимого типа и список переменных. Например, в следующем операторе определяются две переменные типа `flag`: `endOfData` и `matchFound`.

```
enum flag endOfData, matchFound;
```

Единственные значения, которые могут быть присвоены этим переменным — `true` и `false`. Следующие операторы являются допустимыми.

```
endOfData = true;
if (matchFound == false)
 ...
```

Если вам нужно, чтобы определенное целое значение было связано с идентификатором перечисления, это целое значение можно присвоить идентификатору, когда определяется тип данных. Идентификаторам перечисления, которые появляются затем в списке, будут присвоены последовательные целые значения, начиная с указанного целого значения, увеличенного на 1.

В следующем определении перечислимый тип данных `direction` определяется со значениями `up`, `down`, `left` и `right`.

```
enum direction { up, down, left = 10, right }; (вверх, вниз, влево, вправо)
```

Компилятор присвоит значение 0 идентификатору `up`, поскольку он представлен первым в списке, присвоит 1 идентификатору `down`, поскольку он является следующим в списке, присвоит 10 идентификатору `left`, поскольку здесь явно задано это значение, и присвоит 11 идентификатору `right`, поскольку это увеличенное на 1 значение предыдущего идентификатора в списке.

Идентификаторы перечисления могут иметь одинаковое значение. Например, если в строке

```
enum boolean { no = 0, false = 0, yes = 1, true = 1 };
```

булевой переменной типа `enum` присваивается значение `no` или `false`, то ей присваивается значение 0; если присваивается значение `yes` или `true`, то присваивается значение 1.

Ниже приводится еще один пример определения перечислимого типа данных. В нем определяется тип `enum month` с допустимыми значениями, которые могут быть присвоены переменной этого типа — названиями месяцев.

```
enum month { january = 1, february, march, april, may, june, july,
august, september, october, november, december };
```

Компилятор Objective-C интерпретирует идентификаторы перечисления как целые константы. Переменной `thisMonth` будет присвоено значение 2 (а не а не название месяца `february`), если программа содержит следующие две строки:

```
enum month thisMonth;
...
thisMonth = february;
```

В программе 10.3 используются перечислимые типы данных. В ней читается номер месяца, а затем выполняется оператор `switch`, чтобы определить, какой месяц был введен. Напомним, что перечислимые значения интерпретируются как целые константы, поэтому они являются допустимыми `case`-значениями. Переменной `days` присваивается число дней в указанном месяце, и ее значение выводится после выхода из оператора `switch`. Для февраля (February) введена специальная проверка.

### Программа 10.3

```
// вывод числа дней месяца
int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 enum month { january = 1, february, march, april, may, june,
 july, august, september, october, november,
 december };
 enum month amonth;
 int days;

 NSLog (@"Enter month number: ");
 scanf ("%i", &amonth);

 switch (amonth) {
 case january:
 case march:
 case may:
 case july:
 case august:
 case october:
 case december:
 days = 31;
 break;
 case april:
 case june:
 case september:
 days = 30;
 break;
 case february:
 days = 28;
 if (pool->month % 4 == 0)
 days = 29;
 break;
 }
 printf ("Days in month %i: %i\n", amonth, days);
}
[pool release];
```

```

case november:
 days = 30;
 break;
case february:
 days = 28;
 break;
default:
 NSLog(@"bad month number");
 days = 0;
 break;
}

if (days != 0)
 NSLog(@"Number of days is %i", days);

if (amonth == february)
 NSLog(@"...or 29 if it's a leap year");

[pool drain];
return 0;
}

```

### Вывод программы 10.3

```

Enter month number: (Введите номер месяца)
5
Number of days is 31 (Число дней = 31)

```

### Вывод программы 10.3 (повторный запуск)

```

Enter month number:
2
Number of days is 28 ...or 29 if it's a leap year
(Число дней = 28 ... или 29, если это високосный год)

```

Вы можете явно присвоить целое значение переменной перечислимого типа данных; это делается с помощью оператора приведения типа (type cast). Например, если `monthValue` является переменной целого типа, имеющей значение 6, то следующее выражение является допустимым.

```
lastMonth = (enum month) (monthValue - 1);
```

Если вы не используете оператор приведения типа, компилятор, к сожалению, не реагирует на это.

При использовании программ с перечислимыми типами данных старайтесь не полагаться на то, что перечислимые значения интерпретируются как целые числа. Вместо этого их следует рассматривать как отдельные типы данных. Перечислимый тип данных позволяет связывать символическое имя с целым чис-

лом. Если вам потребуется изменить значение этого числа, вам придется изменить его только в том месте, где определено перечисление. Если вы делаете предположения, основанные на фактическом значении перечислимого типа данных, то теряете преимущество перечисления.

При определении перечислимого типа данных имя типа данных может не указываться, и переменные могут быть объявлены как конкретный перечислимый тип данных, если этот тип определен.

#### Оператор

```
enum { east, west, south, north } direction;
```

определяет неименованный перечислимый тип данных со значениями `east`, `west`, `south` и `north` (восток, запад, юг и север) и объявляет переменную (`direction`) этого типа.

Определение перечислимого типа данных в блоке ограничивает область действия этого определения данным блоком. С другой стороны, определение перечислимого типа данных в начале программы (вне какого-либо блока) делает определение глобальным для данного файла.

Определяя перечислимый тип данных, вы должны проследить, чтобы идентификаторы перечисления были уникальными по отношению к именам других переменных и идентификаторам перечисления, определенным в той же области действия.

## Оператор `typedef`

Objective-C позволяет назначать для типа данных альтернативное имя. Для этого используется оператор `typedef`. В следующей строке определяется имя `Counter` (счетчик) как эквивалент типа данных Objective-C `int`.

```
typedef int Counter;
```

Затем можно объявить переменные с типом `Counter`, как в следующей строке.

```
Counter j, n;
```

Компилятор Objective-C будет интерпретировать это как объявление обычных целых переменных `j` и `n`. Определение `j` и `n` посредством `typedef` указывает назначение переменных в программе. Объявление в традиционной форме с типом `int` оставило бы их назначение непонятным. Ниже с помощью `typedef` определяется тип с именем `NumberObject` для объектов типа `Number`.

```
typedef Number *NumberObject;
```

Переменные, которые объявляются затем с типом `NumberObject`, как в строке

```
NumberObject myValue1, myValue2, myResult;
```

будут интерпретироваться так, как если бы они были объявлены обычным образом:

```
Number *myValue1, *myValue2, *myResult;
```

Чтобы определить имя нового типа с помощью `typedef`, нужно выполнить следующую процедуру.

1. Написать такое же объявление, как при объявлении переменной нужного типа.
2. Там, где должно быть имя объявляемой переменной, поместить имя нового типа.
3. Перед всем этим поставить слово `typedef`.

Для примера определим тип с именем `Direction` (Направление) как перечислимый тип данных со значениями `east`, `west`, `north` и `south` (восток, запад, юг и север), напишем определение этого перечислимого типа данных и подставим имя `Direction` там, где обычно ставится имя переменной. Перед всем этим нужно поместить ключевое слово `typedef`.

```
typedef enum { east, west, south, north } Direction;
```

После этого можно объявлять переменные с типом `Direction`:

```
Direction step1, step2;
```

Foundation framework содержит следующее определение `typedef` для `NSComparisonResult` в одном из заголовочных файлов.

```
enum _NSComparisonResult {
 NSOrderedAscending = -1, NSOrderedSame, NSOrderedDescending
};
```

```
typedef NSInteger NSComparisonResult;
```

Некоторые методы в Foundation framework, которые выполняют сравнение (`comparison`), возвращают значение этого типа. Например, метод сравнения строк Foundation с именем `compare:` возвращает значение типа `NSComparisonResult` после сравнения двух строк, которые являются объектами `NSString`. Этот метод объявляется следующим образом.

```
-(NSComparisonResult) compare: (NSString *) string;
```

Чтобы проверить равенство двух объектов `NSString` с именами `userName` и `savedName`, можно включить в программу следующую строку.

```
if ([userName compare: savedName] == NSOrderedSame) {
 // Имена совпадают
 ...
}
```

На самом деле здесь проверяется, равен ли нулю результат метода `compare:`.

## Преобразования типов данных

В главе 4 говорилось о том, что иногда при оценке выражений система неявным образом выполняет преобразования. Там рассматривался случай с типами

данных `float` и `int`. Операция, включающая типы `float` и `int`, выполнялась как операция с плавающей точкой, и элемент данных целого типа автоматически преобразовывался в элемент с плавающей точкой.

Там же было показано, как использовать оператор приведения типа, чтобы явно определить преобразование. Пусть `total` и `n` являются целыми переменными. В строке

```
average = (float) total / n;
```

значение переменной `total` преобразуется в тип `float` до выполнения операции, что гарантирует выполнение деления как операции с плавающей точкой.

## Правила преобразования

Компилятор Objective-C соблюдает строгие правила при оценке выражений, состоящих из различных типов данных.

Ниже описывается порядок, в котором выполняются преобразования при оценке двух operandов в выражении.

1. Если один из operandов имеет тип `long double`, второй operand преобразуется в `long double`, результат имеет такой же тип.
2. Если один из operandов имеет тип `double`, второй operand преобразуется в `double`, результат имеет такой же тип.
3. Если один из operandов имеет тип `float`, второй operand преобразуется в тип `float`, результат имеет такой же тип.
4. Если один из operandов имеет тип `_Bool`, `char`, `short int` или `bit field`<sup>1</sup> или перечислимый тип данных, то он преобразуется в тип `int`.
5. Если один из operandов имеет тип `long long int`, второй operand преобразуется в `long long int`, результат имеет такой же тип.
6. Если один из operandов имеет тип `long int`, второй operand преобразуется в `long int`, результат имеет такой же тип.
7. Если мы дошли до этого шага, то оба operandы имеют тип `int`, результат имеет такой же тип.

Это упрощенная версия шагов преобразования operandов в выражении. Правила усложняются, если включены operandы без знака (`unsigned`). Полный список правил см. в Приложении B.

Из этой последовательности шагов понятно, что если достигнут шаг, где говорится «результат имеет такой же тип», процесс преобразования закончен.

В качестве примера рассмотрим, в каком порядке выполняется оценка следующего выражения. В нем `f` имеет тип `float`, `i` – тип `int`, `l` – `long int`, `s` – переменная типа `short int`.

```
f * i + l / s
```

<sup>1</sup> В главе 13 кратко описывается тип `bit field`.

Рассмотрим сначала умножение `f` на `i`, то есть умножение переменной типа `float` на переменную типа `int`. Из шага 3 известно, что поскольку `f` имеет тип `float`, второй operand (`i`) будет тоже преобразован в тип `float`, и такой же тип будет иметь результат операции умножения.

Затем `l` делится на `s`, то есть выполняется деление переменной типа `long int` на `short int`. Из шага 4 известно, что `short int` преобразуется в `int`. Переходим к шагу 6. Поскольку один из operandов (`l`) имеет тип `long int`, второй operand преобразуется в `long int`, и результат будет иметь такой же тип. Таким образом, это деление дает значение типа `long int`, причем дробная часть отбрасывается.

И, наконец, шаг 3 указывает, что поскольку один из operandов выражения имеет тип `float` (как результат умножения `f * i`), второй operand будет преобразован в тип `float`, и результат будет иметь такой же тип. Таким образом, после деления `l` на `s` результат операции будет преобразован в тип `float` и затем прибавлен к произведению `f` на `i`. Поэтому конечный результат этого выражения будет иметь тип `float`.

Вы всегда можете использовать оператор приведения типа для явных преобразований, управляя оценкой конкретного выражения.

Например, если вы не хотите, чтобы при делении `l` на `s` в предыдущем выражении отбрасывалась дробная часть, то можете выполнить приведение одного из этих operandов к типу `float`, чтобы выполнить деление с плавающей точкой:

`f * i + (float) l / s`

В этом выражении `l` будет преобразована в тип `float` до операции деления, поскольку оператор приведения типа имеет более высокий приоритет, чем оператор деления. Поскольку один из operandов деления будет теперь иметь тип `float`, другие operandы будут автоматически преобразованы в тип `float`, и такой же тип будет иметь результат.

## Расширение для знака

Если переменная типа `signed int` или `signed short` преобразуется в тип `int` или тип большего размера, то в результате преобразования слева тоже появляется описатель `signed`. Поэтому переменная типа `short int`, которая имеет значение `-5`, будет иметь значение `-5` после преобразования в `long int`. Но если целая переменная с описателем `unsigned` преобразуется в тип `int` или тип большего размера, то расширение для знака не происходит (как и можно было ожидать).

На некоторых машинах (например, с процессорами Intel, которые используются в современных семействах компьютеров Macintosh, или с процессорами ARM, которые используются в iPhone и iPad) символы интерпретируются как значения со знаком. Это означает, что если символ (`character`) преобразуется в целый тип, то происходит расширение для знака. Если используются символы из стандартного набора символов ASCII, это не создает проблем. Но если значение символа не является частью этого стандартного набора символов, для его знака может быть выполнено расширение, если символ преобразуется в целый тип. Например, на компьютерах Mac символная константа '`\377`' преобра-

зуется в значение -1, потому что это значение является отрицательным, если рассматривать его как 8-битное значение со знаком (signed).

Objective-C позволяет объявлять символьные переменные без знака (`unsigned`), что позволяет избежать этой потенциальной проблемы — переменная `unsigned char` не получает расширения для знака при преобразовании в целый тип; ее значение всегда больше или равно нулю. Для обычного 8-битного символа символьная переменная со знаком имеет диапазон значений от -128 до +127 включительно. Символьная переменная без знака имеет диапазон значений от 0 до 255 включительно.

Если ваши символьные переменные должны получать расширение для знака, вы можете объявить такие переменные с типом `signed char`. В главе 15 вы узнаете о многобайтных символах Unicode. Это предпочтительный способ работы со строками.

## Упражнения

- Используя класс `Rectangle` из главы 8, добавьте метод-инициализатор в соответствии со следующим объявлением.  
`-(Rectangle *) initWithWidth: (int) w: andHeight: (int) h;`
- С учетом того, что мы назвали метод, разработанный в упражнении 1, назначенным (designated) инициализатором для класса `Rectangle`, и основываясь на определениях классов `Square` и `Rectangle` из главы 8, добавьте метод-инициализатор в класс `Square` в соответствии со следующим объявлением.  
`-(Square *) initWithSide: (int) side;`
- Добавьте счетчик (counter) к методу `add:` класса `Fraction`, чтобы вычислять количество вызовов этого метода. Каким образом вы можете считывать значение этого счетчика?
- Используя `typedef` и перечислимые типы данных, определите тип с именем `Day` (День) с возможными значениями `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday` и `Saturday` (Воскресенье, Понедельник, Вторник, Среда, Четверг, Пятница и Суббота).
- Используя `typedef`, определите тип с именем `FractionObj`, который позволяет писать следующие операторы.

```
FractionObj f1 = [[Fraction alloc] init],
f2 = [[Fraction alloc] init];
```

- Используя определения

```
float f = 1.00;
short int i = 100;
long int l = 500L;
double d = 15.00;
```

и семь шагов, описанных выше для преобразования operandов в выражениях, определите тип и значение следующих выражений.

f + i  
l / d  
i / l + f  
l \* i  
f / 2  
i / (d + f)  
l / (i \* 2.0)  
l + i / (double) l

7. Напишите программу, которая определяет, выполняется ли на вашей машине расширение для знака у переменных `signed char`.

# Глава 11

# Категории и протоколы

В этой главе описывается, как добавлять методы для класса в модульном стиле с помощью категорий и как создавать стандартизованный список методов для реализации другими людьми.

## Категории

При работе с определенным классом к нему приходится добавлять новые методы. Например, для класса `Fraction` могут потребоваться методы, реализующие вычитание, умножение и деление двух дробей.

Предположим, что ваша группа в составе большого проекта определяет новый класс, который содержит много различных методов. Вашей задачей является написание для этого класса методов, которые работают с файловой системой. Другие члены проекта должны написать методы, которые реализуют создание и инициализацию экземпляров этого класса, выполняют операции над объектами в этом классе и рисуют представления объектов этого класса на экране.

Вы изучили, как использовать класс массивов из библиотеки Foundation framework с именем `NSArray` и поняли, что в этом классе необходимо реализовать один или нескольких методов. Конечно, вы могли бы написать новый подкласс класса `NSArray` и реализовать эти новые методы, но есть более простой способ.

На практике для разрешения подобных ситуаций используются *категории* (*category*). Категория представляет простой способ модульного определения класса в виде групп или категорий связанных методов. Она также позволяет достаточно просто расширить существующее определение класса без доступа к существующему исходному коду этого класса или создания подкласса. Это мощная и достаточно простая для изучения концепция.

Вернемся к первому примеру и покажем, как добавить новую категорию в класс `Fraction` для работы с четырьмя основными арифметическими операциями. Приведем исходную секцию interface для `Fraction`.

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

#import <stdio.h>

// Определение класса Fraction

@interface Fraction : NSObject
{
```

```

int numerator;
int denominator;
}

@property int numerator, denominator;
-(void) setTo: (int) n over: (int) d;
-(Fraction *) add: (Fraction *) f;
-(void) reduce;
-(double) convertToNum;
-(void) print;
@end

```

Теперь удалим метод `add:` из этой секции `interface` и добавим его в новую категорию вместе с тремя другими арифметическими операциями, которые нужно реализовать. Ниже показана секция `interface` для новой категории `MathOps`.

```

#import «Fraction.h»
@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end

```

Здесь представлено некоторое определение секции `interface`, но на самом деле это расширение существующей секции. Мы должны включить исходную секцию `interface`, чтобы указать компилятору на класс `Fraction` (правда, вы можете включить эту новую категорию непосредственно в исходный файл `Fraction.h`).

После строки `#import` мы вилим следующую строку:

```
@interface Fraction (MathOps)
```

Она указывает компилятору, что для класса `Fraction` определяется новая категория с именем `MathOps`. Имя категории после имени класса заключено в круглые скобки. Отметим, что здесь не указывается родительский класс для `Fraction`; компилятору уже известно о нем из `Fraction.h`. Кроме того, мы ничего не сообщаем ему о переменных экземпляра, хотя делали это во всех предыдущих секциях `interface`. На самом деле мы получим от компилятора сообщение о синтаксической ошибке, если попытаемся включить родительский класс или переменные экземпляра.

Эта секция `interface` указывает компилятору, что мы добавляем в класс `Fraction` расширение под категорией с именем `MathOps`. Категория `MathOps` содержит четыре метода экземпляра: `add:`, `mul:`, `sub:` и `div:`. Каждый метод получает в качестве аргумента дробь (`fraction`) и возвращает тоже дробь.

Определения всех этих четырех методов можно поместить в одну секцию `implementation`. Мы можем определить в одной секции `implementation` все методы из секции `interface` файла `Fraction.h` плюс методы из категории `MathOps`. Мы также можем определить методы из этой категории в отдельной секции `implementation`. Тогда в секции `implementation` для этих методов следует также

идентифицировать категорию, к которой принадлежат эти методы. Как и в секции `interface`, имя категории после имени класса нужно заключить: в круглые скобки

```
@implementation Fraction (MathOps)

// код для методов этой категории

...
```

```
@end
```

В программе 11.1 для новой категории `MathOps` секции `interface` и `implementation` объединены в одном файле вместе с тестовой процедурой.

### Программа 11.1. Категория `MathOps` и тестовая программа

```
#import "Fraction.h"

@interface Fraction (MathOps)
-(Fraction *) add: (Fraction *) f;
-(Fraction *) mul: (Fraction *) f;
-(Fraction *) sub: (Fraction *) f;
-(Fraction *) div: (Fraction *) f;
@end

@implementation Fraction (MathOps)
-(Fraction *) add: (Fraction *) f
{
 // Для сложения двух дробей:
 // a/b + c/d = ((a*d) + (b*c)) / (b * d)

 Fraction *result = [[Fraction alloc] init];
 int resultNum, resultDenom;

 resultNum = (numerator * f.denominator) +
 (denominator * f.numerator);
 resultDenom = denominator * f.denominator;

 [result setTo: resultNum over: resultDenom];
 [result reduce];

 return result;
}

// Для вычитания двух дробей:
// a/b - c/d = ((a*d) - (b*c)) / (b * d)

Fraction *result = [[Fraction alloc] init];
int resultNum, resultDenom;
```

```
resultNum = (numerator * f.denominator) -
 (denominator * f.numerator);
resultDenom = denominator * f.denominator;

[result setTo: resultNum over: resultDenom];
[result reduce];

return result;
}

-(Fraction *) mul: (Fraction *) f
{
 Fraction *result = [[Fraction alloc] init];

 [result setTo: numerator * f.numerator
 over: denominator * f.denominator];
 [result reduce];

 return result;
}

-(Fraction *) div: (Fraction *) f
{
 Fraction *result = [[Fraction alloc] init];

 [result setTo: numerator * f.denominator
 over: denominator * f.numerator];
 [result reduce];

 return result;
}
@end

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

 Fraction *a = [[Fraction alloc] init];
 Fraction *b = [[Fraction alloc] init];
 Fraction *result;

 [a setTo: 1 over: 3];
 [b setTo: 2 over: 5];

 [a print]; NSLog(@"%@", "+"); [b print]; NSLog(@"%@", "----");
 result = [a add: b];
```

```
[result print];
NSLog (@"\n");
[result release];

[a print]; NSLog (@"-"); [b print]; NSLog (@"----");
result = [a sub: b];
[result print];
NSLog (@"\n");
[result release];

[a print]; NSLog (@" *"); [b print]; NSLog (@"----");
result = [a mul: b];
[result print];
NSLog (@"\n");
[result release];

[a print]; NSLog (@" /"); [b print]; NSLog (@"----");
result = [a div: b];
[result print];
NSLog (@"\n");
[result release];
[a release];
[b release];

[pool drain];
return 0;
}
```

### Вывод программы 11.1

```
1/3
+
2/5

11/15
```

```
1/3
-
2/5

-1/15
```

```
1/3
*
2/5

2/15
```

1/3

/

2/5

-----

5/6

И снова напомним, что в Objective-C вполне допустимы такие операторы, как

```
[[a div: b] print];
```

В этой строке выполняется непосредственный вывод результата деления *a* на *b*, что позволяет избежать промежуточного присваивания результата какой-либо переменной, как в программе 11.1. Нам нужно это промежуточное присваивание, чтобы получить результатирующую дробь (*Fraction*) и освободить память, которую она занимает. В противном случае при каждом выполнении арифметической операции над дробью будет происходить утечка памяти.

В программе 11.1 секции *interface* и *implementation* для новой категории помещены в один файл вместе с тестовой программой. Как уже говорилось выше, секция *interface* для категории может быть включена в исходный *header*-файл *Fraction.h* (чтобы все методы были объявлены в одном месте), или в свой собственный *header*-файл.

Если поместить категорию в мастер-файл определения класса, все пользователи этого класса будут иметь доступ к методам данной категории. Если у вас нет возможности непосредственного внесения изменений в исходный *header*-файл (см. добавление категории в существующий класс из библиотеки в части II, «Foundation Framework»), то вы вынуждены хранить категорию в отдельном файле.

## Некоторые замечания по категориям

Отметим несколько особенностей категорий. Во-первых, хотя категория имеет доступ к переменным экземпляра исходного класса, в ней нельзя добавить ее собственные переменные экземпляра. При необходимости нужно использовать подклассы.

Кроме того, категория может замещать другой метод своего класса. Обычно это считается недопустимым в практике надежного программирования. После замещения метода вы уже не имеете доступа к исходному методу, поэтому при замене вы должны аккуратно дублировать все функциональные возможности замещаемого метода. Если вам действительно требуется замещение какого-либо метода, используйте подклассы. При замещении метода в подклассе вы можете по-прежнему обращаться к родительскому методу, передавая сообщение *super*. В этом случае вам не нужно знать все особенности метода, который вы замещаете; можно просто вызвать родительский метод и добавить ваши функциональные возможности в метод подкласса.

Соблюдая изложенные здесь правила, можно иметь сколько угодно категорий. Если метод определен более чем в одной категории, язык не указывает, какая из них будет использоваться.

В отличие от обычной секции *interface*, вам не нужно реализовать все ме-

ды, указанные в категории. Это полезно для пошаговой разработки программ, поскольку вы можете объявить все методы в категории и постепенно их реализовывать.

Расширение класса путем добавления новых методов с помощью категории влияет не только на этот класс, но и на все его подклассы. Это может быть потенциально опасным. Например, если вы добавляете новые методы в корневой объект `NSObject`, каждый пользователь будет наследовать эти методы. Новые методы, которые вы добавляете к существующему классу с помощью категории, возможно, будут отвечать вашим намерениям, но могут оказаться несогласованными с исходной организацией или целями класса. Например, превращение квадрата (Square) в окружность (Circle) путем добавления новой категории и некоторых методов искажает определение класса и не согласуется с практикой надежного программирования.

Кроме того, пары объект/категория должны быть уникальными. Только одна категория `NSString (Private)` может существовать в заданном пространстве имен Objective-C. Это может вызывать затруднения, поскольку пространство имен Objective-C совместно используется кодом программы и всеми библиотеками, структурами framework (фреймворками) и дополнительными программными модулями (plug-in). Это особенно важно для программистов Objective-C, которые пишут коды экранных заставок (screensaver), панелей предпочтений и других дополнительных модулей, поскольку их код будет вставляться в код приложения или фреймворк, которыми они не могут управлять.

## Протоколы

*Протокол (protocol)* – это список методов, которые совместно используются классами. Методы, включенные в протокол, не имеют соответствующих реализаций (*implementation*): предполагается, что они будут реализованы кем-то другим. Протокол – это способ определения набора методов, которые каким-либо образом связаны с указанным именем. Эти методы обычно документируются, что позволяет вам реализовать их в ваших определениях классов.

Если вы реализуете все необходимые методы для определенного протокола, то вы *подчиняетесь (conform)* этому протоколу или *принимаете (adopt)* его.

Протокол определяется просто: укажите имя протокола после директивы `@protocol`. После этого нужно объявить методы так же, как в секции `interface`. Все объявления методов вплоть до директивы `@end` становятся частью данного протокола.

При работе с Foundation framework вы увидите, что несколько протоколов уже определены. Один из них, `NSCopying`, объявляет метод, который вам потребуется реализовать, если ваш класс должен поддерживать копирование объектов с помощью метода `copy` (или `copyWithZone:`). (Подробно тема копирования объектов рассматривается в главе 18.)

Ниже показано, как определяется протокол `NSCopying` в стандартном файле Foundation `NSObject.h`.

```
@protocol NSCopying
- (id)copyWithZone: (NSZone *)zone;
@end
```

Если вы приняли протокол `NSCopying` в своем классе, то должны реализовать метод `copyWithZone:`. Вы сообщаете компилятору, что принимаете протокол, заключая имя этого протокола в угловые скобки (`<...>`) в строке `@interface`. Имя протокола указывается после имени класса и его родительского класса, как в следующей строке:

```
@interface AddressBook: NSObject <NSCopying>
```

В этой строке сообщается, что `AddressBook` является объектом с родительским классом `NSObject` и подчиняется протоколу `NSCopying`. Поскольку система уже знает о методах, определенных ранее для этого протокола (в данном случае – из файла `NSObject.h`), эти методы не нужно объявлять в секции `interface`. Однако их нужно определить в вашей секции `implementation`.

В данном примере в секции `implementation` для `AddressBook` компилятор предполагает обнаружить определение метода `copyWithZone:`.

Если ваш класс принимает более одного протокола, просто перечислите их в угловых скобках, разделяя запятыми:

```
@interface AddressBook: NSObject <NSCopying, NSCoding>
```

Здесь компилятору сообщается, что класс `AddressBook` принимает протоколы `NSCopying` и `NSCoding`. В данном случае компилятор предполагает обнаружить определение всех требуемых методов (перечисленных в этих протоколах) в секции `implementation` для `AddressBook`.

Определив свой собственный протокол, вы не обязаны реализовать его сами. Вы уведомляете других программистов, что если они хотят принять этот протокол, то должны реализовать соответствующие методы. Эти методы могут наследоваться из суперкласса. Так, если класс подчиняется протоколу `NSCopying`, это действительно и для его подклассов (хотя и не означает, что эти методы правильно реализованы для данного подкласса).

Протокол позволяет определить методы, которые будут реализовать другие люди, использующие подкласс вашего класса. Например, вы можете определить протокол `Drawing` для своего класса `GraphicObject`; в нем можно определить методы `paint` (окраска), `erase` (стирание) и `outline` (контур).

```
@protocol Drawing
-(void) paint;
-(void) erase;
@optional
-(void) outline;
@end
```

Создав класс `GraphicObject`, вы не обязаны реализовать эти методы, однако вы должны указать методы, которые должен реализовать человек, создающий подкласс класса `GraphicObject`, чтобы соответствовать стандарту для создаваемых объектов рисования.

---

**Примечание.** Любые методы, которые указаны после директивы `@optional` директивы, не являются обязательными. Что человек, принявший протокол `Drawing`, не обязан реализовать метод `outline`, подчиняясь этому протоколу. (Вы можете вернуться к перечислению обязательных протоколов с помощью директивы `@required` в определении протокола.)

---

Таким образом, если вы создаете подкласс `Rectangle` класса `GraphicObject` и объявляете (то есть документируете), что ваш класс `Rectangle` подчиняется протоколу `Drawing`, пользователи данного класса будут знать, что они могут передавать экземплярам этого класса сообщения `paint`, `erase` и (возможно) `outline`.

---

**Примечание.** Это теория. Компилятор позволяет вам указать, что вы подчиняетесь протоколу, и выдает предупреждающие сообщения, только если вы не реализуете эти методы.

---

Отмстим, что в протоколе нет ссылки ни на какие классы; это «бесклассовое» средство. Протоколу `Drawing` может подчиняться любой класс, не только подклассы `GraphicObject`.

Чтобы проверить, подчиняется ли объект какому-либо протоколу, можно использовать метод `conformsToProtocol:`. Например, вы хотите определить, подчиняется ли объект с именем `currentObject` протоколу `Drawing`, чтобы передавать этому объекту сообщения для рисования. Для этого можно написать следующее.

```
id currentObject;
...
if ([currentObject conformsToProtocol: @protocol (Drawing)] == YES)
{
 // Передача сообщений currentObject paint, erase и/или outline
 ...
}
```

Специальная директива `@protocol`, которая используется здесь, принимает имя протокола и создает объект типа `Protocol`, который используется как аргумент методом `conformsToProtocol:`.

Вы можете воспользоваться помощью компилятора, чтобы проверить согласование с вашими переменными, заключив имя протокола в угловые скобки после имени типа:

```
id <Drawing> currentObject;
```

Это указывает компилятору, что `currentObject` будет содержать объекты, подчиняющиеся протоколу `Drawing`. Если присвоить `currentObject` объект статического типа, который не согласуется с протоколом `Drawing` (например, у вас есть несогласующийся класс `Square`), то компилятор выдаст предупреждающее сообщение:

```
warning: class 'Square' does not implement the 'Drawing' protocol
(предупреждение: класс 'Square' не реализует протокол 'Drawing')
```

Здесь проверку выполняет компилятор, поэтому присваивание `currentObject` переменной типа `id` не приведет к выводу этого сообщения. Для объекта, хранящегося в переменной типа `id`, компилятор не сможет определить, подчиняется ли он протоколу `Drawing`.

В списке можно указать более одного протокола, если переменная будет содержать объект, подчиняющийся нескольким протоколам:

```
id <NSCopying, NSCoding> myDocument;
```

Определение протокола можно расширять. В следующем объявлении протокола указывается, что протокол `Drawing3D` принимает также протокол `Drawing`.

```
@protocol Drawing3D <Drawing>
```

Таким образом, класс, который принимает протокол `Drawing3D`, должен реализовать методы, перечисленные для этого протокола, а также методы из протокола `Drawing`.

И, наконец, категория тоже может принимать протокол, например:

```
@interface Fraction (Stuff) <NSCopying, NSCoding>
```

Здесь `Fraction` содержит категорию `Stuff`, которая принимает протоколы `NSCopying` и `NSCoding`.

Как и имена классов, имена протоколов должны быть уникальными.

## Неформальные протоколы

В литературе встречается понятие *неформального, свободного (informal)* протокола. На самом деле это категория, содержащая список группы методов, но не реализующая их. Все (или почти все) наследуется из одного корневого объекта, поэтому неформальные категории часто определяются для корневого класса. Иногда неформальные протоколы называют также *абстрактными (abstract)* протоколами.

В header-файле `<NSScriptWhoseTests.h>` могут встретиться объявления методов, которые выглядят следующим образом:

```
@interface NSObject (NSComparisonMethods)
```

```
- (BOOL)isEqualTo:(id)object;
- (BOOL)isLessThanOrEqualTo:(id)object;
- (BOOL)isLessThan:(id)object;
- (BOOL)isGreaterThanOrEqualTo:(id)object;
- (BOOL)isGreaterThan:(id)object;
- (BOOL)isNotEqualTo:(id)object;
- (BOOL)doesContain:(id)object;
- (BOOL)isLike:(NSString *)object;
- (BOOL)isCaseInsensitiveLike:(NSString *)object;
@end
```

Здесь определяется категория с именем `NSComparisonMethods` для класса `NSObject`. В этом неформальном протоколе содержится список группы методов (в дан-

ном случае – девять), которые могут быть реализованы как часть этого протокола. Неформальный протокол – это фактически просто группа методов под определенным именем. Это полезно с точки зрения документирования и модульной организации методов.

Класс, где объявляется неформальный протокол, не реализует методы в самом этом классе. В подклассе, выбранном для реализации этих методов, требуется переобъявить их в секции `interface`, а также реализовать один или несколько из этих методов. В отличие от формальных протоколов, компилятор не оказывает никакой помощи с неформальными протоколами: здесь нет никакой концепции подчинения или проверки компилятором.

Если объект принимает какой-либо формальный протокол, этот объект должен подчиняться всем требуемым сообщениям в этом протоколе. Это можно сделать как на этапе выполнения (`runtime`), так и во время компиляции. Если объект принимает неформальный протокол, то он не обязан принять все методы данного протокола (в зависимости от самого протокола). Подчинение неформальному протоколу можно сделать обязательным на этапе выполнения (с помощью `respondsToSelector:`), но не во время компиляции.

---

**Примечание.** Описанную выше директиву `@optional` (она была добавлена в Objective-C 2.0) можно использовать вместо неформальных протоколов. Она используется в нескольких классах `UIKit` (`UIKit` – составная часть структур Cocoa Touch framework).

---

## Составные объекты

Вы уже знаете несколько способов, позволяющих расширить определение класса с помощью таких средств, как подклассы и категории. Еще один способ – это определение класса, который содержит один или несколько объектов из других классов. Объект из этого класса называется *составным* (*composite*) объектом, поскольку он составлен из других объектов.

В качестве примера рассмотрим класс `Square` (квадрат), который мы определили в главе 8. Он был определен как подкласс класса `Rectangle` (прямоугольник), поскольку квадрат – это прямоугольник с равными сторонами. Подкласс, который мы определяем, наследует все переменные экземпляра и методы родительского класса. В некоторых случаях это нежелательно. Метод `setWidth:andHeight:` (задание ширины и высоты) класса `Rectangle` наследуется классом `Square`, но реально не относится к квадрату (хотя действует правильно). Кроме того, создавая подкласс, мы должны обеспечить правильность работы наследуемых методов, поскольку пользователи этого подкласса будут иметь к ним доступ.

Вместо подкласса можно определить новый класс, который содержит в качестве одной из своих переменных экземпляра объект из класса, который вы хотите расширить. Затем нужно определить в новом классе только те методы, которые для него подходят. В примере с классом `Square` можно определить `Square` следующим образом.

```
@interface Square: NSObject
{
```

```

 Rectangle *rect;
}
-(int) setSide: (int) s;
-(int) side; (сторона)
-(int) area; (площадь)
-(int) perimeter; (периметр)
@end

```

Здесь определен класс `Square` с четырьмя методами. В отличие от версии с подклассом, которая дает непосредственный доступ к методам класса `Rectangle` (`setWidth:`, `setHeight:`, `setWidth:andHeight:`, `width` и `height`), этих методов нет в определении для `Square`. Это имеет смысл, поскольку не все методы подходят для работы с квадратами.

Если мы определяем класс `Square` таким способом, то он становится ответственным за выделение памяти для прямоугольника (`rectangle`), который содержит. Например, без замещающих методов оператор

```
Square *mySquare = [[Square alloc] init];
```

выделяет память для нового объекта типа `Square`, но не выделяет память для объекта типа `Rectangle`, хранящегося в его переменной экземпляра `rect`.

Чтобы выполнить выделение памяти, требуется замещение метода `init` или добавление нового метода, например, `initWithSide:`. Этот метод может выделять память для переменной `Rectangle rect` и задавать соответствующим образом сторону (`side`). Необходимо также заместить метод `dealloc` (как описано для класса `Rectangle` в главе 8), чтобы освободить память, используемую для `Rectangle rect`, когда освобождается сам объект `Square`.

Определяя свои методы в классе `Square`, мы по-прежнему можем использовать методы класса `Rectangle`. Например, мы можем реализовать метод `area` следующим образом:

```

-(int) area
{
 return [rect area];
}

```

Реализацию остальных методов вы можете написать в качестве упражнения (см. ниже упражнение 5).

## Упражнения

1. Выполните расширение категории `MathOps` из программы 11.1, чтобы дополнительно включить метод `invert`, который возвращает дробь (`Fraction`), обратную получателю.
2. Добавьте в класс `Fraction` категорию с именем `Comparison`. В этой категории добавьте два метода в соответствии со следующими объявлениями.  
`-(BOOL) isEqualTo: (Fraction *) f;`  
`-(int) compare: (Fraction *) f;`

Первый метод должен возвращать значение YES, если две дроби равны, и значение NO в противном случае. Правильно сравнивайте дроби (например, при сравнении дробей 3/4 и 6/8 следует возвращать значение YES). Второй метод должен возвращать значение -1, если получатель меньше дроби, представляющейся аргументом; возвращать 0, если две дроби равны; возвращать 1, если получатель больше дроби, представляющейся аргументом.

3. Выполните расширение класса Fraction, добавив методы, которые подчиняются неформальному протоколу NSComparisonMethods, описанному в этой главе. Напишите реализацию первых шести методов из этого протокола (isEqualTo:, isLessThanOrEqualTo:, isLessThan:, isGreaterThanOrEqualTo:, isGreaterThan:, isNotEqualTo:) и выполните их тестирование.
4. Функции sin(), cos() и tan() включены в стандартную библиотеку Standard Library (как и scanf()). Эти функции объявлены в header-файле <math.h>, который вы должны импортировать в программу с помощью строки

```
#import <math.h>
```

Эти функции можно использовать для вычисления синуса, косинуса и тангенса аргумента типа double, выраженного в радианах. Возвращаемый результат является значением с плавающей точкой двойной точности. Например, для вычисления синуса аргумента d, где d – угол, выраженный в радианах, можно использовать следующую строку:

```
result = sin(d);
```

Добавьте категорию с именем Trig в класс Calculator, определенный в главе 6. Включите в эту категорию методы для вычисления синуса, косинуса и тангенса в соответствии со следующими объявлениями.

```
-(double) sin;
-(double) cos;
-(double) tan;
```

5. Напишите секцию implementation для Square и выполните тестирование программы для проверки ее методов, используя описание составных объектов из этой главы и следующую секцию interface,

```
@interface Square: NSObject
{
 Rectangle *rect;
}
-(Square*) initWithSide: (int) s;
-(void) setSide: (int) s;
-(int) side;
-(int) area;
-(int) perimeter;
-(void) dealloc; // Замещающий метод для освобождения памяти объекта типа Rectangle
@end
```

## Глава 12

# Препроцессор

Препроцессор содержит средства, упрощающие чтение, разработку и встраивание программ в различные системы. Препроцессор позволяет настроить язык Objective-C в соответствии с конкретным программным приложением или вашим стилем программирования.

Препроцессор – это составная часть процесса компиляции Objective-C. Он распознает специальные операторы и обрабатывает их, прежде чем будет выполнен анализ самой программы. Операторы препроцессора идентифицируются знаком «решетка» (#), который должен быть первым символом строки, отличным от пробела. Синтаксис операторов препроцессора несколько отличается от обычных операторов Objective-C. Мы начнем с описания оператора `#define`.

## Оператор `#define`

Оператор `#define` позволяет присваивать символические имена программным константам. Оператор препроцессора

```
#define TRUE 1
```

определяет имя `TRUE` и делает его эквивалентным значению 1. Затем имя `TRUE` можно использовать в любом месте программы, где могла бы использоваться константа 1. Там, где появляется это имя, препроцессор автоматически подставляет вместо него значение 1. Например, следующий оператор Objective-C использует определенное имя `TRUE`.

```
gameOver = TRUE;
```

Этот оператор присваивает значение `TRUE` переменной `gameOver`. Вам не обязательно помнить, какое конкретное значение вы определили для `TRUE`. Приведенный выше оператор будет присваивать 1 переменной `gameOver`. Оператор препроцессора

```
#define FALSE 0
```

определяет имя `FALSE` и делает его эквивалентным значению 0. Например, оператор

```
gameOver = FALSE;
```

присваивает значение `FALSE` переменной `gameOver`, а оператор

```
if (gameOver == FALSE)
...
```

сравнивает значение `gameOver` с определенным значением `FALSE`.

Определенное таким образом имя *не* является переменной, поэтому вы не можете присвоить ему значение, если результатом подстановки значения не является переменная. Если в программе используется определенное имя, препроцессор автоматически подставляет то, что представлено в правой части оператора `#define`. Это аналогично операции поиска и замены в текстовом редакторе; в данном случае препроцессор заменяет все экземпляры определенного имени соответствующим текстом.

Отметим, что оператор `#define` имеет специальный синтаксис: для присваивания `TRUE` значения 1 не используется знак равенства. Кроме того, в конце оператора нет точки с запятой.

Операторы `#define` часто помещают ближе к началу программы, после операторов `#import` или `#include`, хотя они могут присутствовать в любом месте программы. Однако имя должно быть определено до того, как оно будет использовано. Определенные таким образом имена отличаются по своему поведению от переменных: не существует такого понятия, как локальный «`define`». После определения имени его можно использовать *в любом месте* программы. Большинство программистов помещают операторы `define` в header-файлы (`*.h`), чтобы использовать их в нескольких исходных файлах.

В следующем примере напишем два метода для поиска площади и длины окружности объекта типа `Circle`. Поскольку в обоих методах требуется использовать константу, значение которой трудно запомнить, имеет смысл определить значение этой константы в начале программы и затем применять его в каждом методе.

Поэтому мы можем включить в программу следующую строку:

```
#define PI 3.141592654
```

Теперь эту константу можно использовать в обоих методах класса `Circle` (в предположении, что класс `Circle` содержит переменную экземпляра с именем `radius`).

```
-(double) area
{
 return PI * radius * radius;
}

-(double) circumference
{
 return 2.0 * PI * radius;
}
```

Назначение константы для символического имени освобождает вас от необходимости помнить значение константы. Кроме того, если потребуется изменить значение этой константы, то это нужно будет сделать только в одном месте программы: в операторе `#define`. В противном вам придется выполнить поиск по всей программе и явно изменить значение константы.

Во всех приведенных примерах операторов `define` использовались прописные буквы (`TRUE`, `FALSE` и `PI`). Это было сделано, чтобы визуально отличить определенное значение от переменной. Некоторые программисты записывают все определенные имена прописными буквами, чтобы сразу различать, что перед ними: переменную, объект, имя класса или определенное имя. Перед определенным именем принято ставить букву `k`. В этом случае прописными буквами обозначаются не все символы, например, `kMaximumValues` и `kSignificantDigits`.

Применение определенного имени для значения константы упрощает расширение программы. Для массивов вместо конкретного указания размера массива, который нужно выделить в памяти, можно определить нужное значение:

```
#define MAXIMUM_DATA_VALUES 1000
```

Теперь на этом определенном значении можно основывать все ссылки на размер массива (например, чтобы выделить место для этого массива в памяти) и допустимые индексные значения.

Кроме того, если программа использует `MAXIMUM_DATA_VALUES` во всех случаях, где нужно указать размер массива, то чтобы изменить размер массива, достаточно изменить только это определение.

## Более сложные типы определений

Определение имени может включать не только простое константное значение. Это может быть и выражение и почти все, что можно предположить!

Ниже имя `TWO_PI` определяется как произведение 2.0 на 3.141592654:

```
#define TWO_PI 2.0 * 3.141592654
```

После этого определенное имя можно использовать в любом месте программы, где требуется выражение `2.0 * 3.141592654`. Например, оператор `return` в методе `circumference` из предыдущего примера можно заменить оператором

```
return TWO_PI * radius;
```

Если в программе на Objective-C встречается определенное имя, то все, что находится справа от этого определенного имени в операторе `#define`, подставляется вместо этого имени. Так, если препроцессор встречает имя `TWO_PI` в приведенном выше операторе `return`, он подставляет вместо этого имени то, что находится справа от этого имени в операторе `#define`, то есть `2.0 * 3.141592654`.

Теперь вы понимаете, почему оператор `#define` нельзя закончить точкой с запятой. Если вы сделаете это, точка с запятой будет подставлена в том месте, где встретится соответствующее определенное имя. Если определить `PI` с помощью оператора

```
#define PI 3.141592654;
```

и затем написать

```
return 2.0 * PI * r;
```

препроцессор заменит этот экземпляр определенного имени `PI` на `3.141592654`. Тогда после подстановки компилятору нужно будет обработать оператор

```
return 2.0 * 3.141592654; * r;
```

что даст синтаксическую ошибку. Ставить точку с запятой после операторов `define` можно только в тех случаях, когда это действительно нужно.

Определение для препроцессора может не быть допустимым выражением Objective-C, но результирующее выражение в программе должно быть допустимым. Например, вы можете задать определения:

```
#define AND &&
#define OR ||
```

А затем писать такие выражения, как

```
if (x > 0 AND x < 10)
```

```
...
```

и

```
if (y == 0 OR y == value)
```

```
...
```

Можно даже использовать `#define` для оператора проверки равенства:

```
#define EQUALS ==
```

И затем написать выражение

```
if (y EQUALS 0 OR y EQUALS value)
```

```
...
```

Это позволяет избежать ошибки в тех случаях, когда для проверки равенства используют один знак равенства.

Возможности оператора `#define` велики, но практика надежного программирования не приветствует такие способы переопределения синтаксиса базового языка. Кроме того, это затрудняет другим людям понимание вашего кода.

Это еще не все. Поскольку определенное значение может ссылаться на другое определенное значение, следующие две строки `#define` допустимы.

```
#define PI 3.141592654
#define TWO_PI 2.0 * PI
```

Имя `TWO_PI` определяется со ссылкой на предыдущее определенное имя `PI`, что избавляет от необходимости повторно писать значение `3.141592654`.

Обратный порядок этих определений тоже допустим.

```
#define TWO_PI 2.0 * PI
#define PI 3.141592654
```

Определенные значения можно использовать в определениях, если все, что нужно, определено на тот момент, когда соответствующее определенное имя используется в программе.

Разумное использование операторов `#define` позволяет сократить количество комментариев в программе. Рассмотрим оператор

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
...
```

В выражении проверяется, високосный ли год записан в переменной `year`. Рассмотрим оператор `#define` и оператор `if`.

```
#define IS_LEAP_YEAR year % 4 == 0 && year % 100 != 0 \
||year % 400 == 0
...
if (IS_LEAP_YEAR)
...
```

Обычно препроцессор предполагает, что определение целиком содержится в одной строке. Символом переноса строки (т.е. продолжения оператора) для препроцессора является обратный слэш. Этот символ игнорируется при подстановке. Если строк продолжения несколько, то каждая строка должна заканчиваться обратным слэшем.

Последний оператор `if` намного понятнее, чем предыдущий. Конечно, данное определение позволяет проверить на високосный год только переменную `year`, но мы можем написать определение, с помощью которого можно проверить на високосный год любое выражение. Для этого нужно включить в определение один или несколько аргументов.

`IS_LEAP_YEAR` можно определить с аргументом `y` следующим образом.

```
#define IS_LEAP_YEAR(y) y % 4 == 0 && y % 100 != 0 \
|| y % 400 == 0
```

В отличие от определения метода, мы не определяем тип аргумента `y`, поскольку здесь выполняется просто подстановка текста, а не вызов функции. Отметим, что в определении имени с использованием аргументов не допуска-

ются пробелы между определенным именем и левой круглой скобкой перед списком аргументов.

Используя это определение, напишем оператор

```
if (IS_LEAP_YEAR (year))
...
```

Оператор проверяет, является ли значение для `year` високосным годом. Можно написать аналогичное выражение для `nextYear`.

```
if (IS_LEAP_YEAR (nextYear))
...
```

В этом операторе определение для `IS_LEAP_YEAR` непосредственно подставляется в оператор `if` с заменой аргумента `y` на `nextYear`. В результате компилятор будет обрабатывать следующий оператор.

```
if (nextYear % 4 == 0 && nextYear % 100 != 0 || nextYear % 400 == 0)
...
```

Определения час с одним или несколькими аргументами то называют *макросами*. В следующем макросе представлен квадрат его аргумента.

```
#define SQUARE(x) x * x
```

В определении для `SQUARE` необходимо учитывать возможность следующей ошибки. Исходя из нашего описания, оператор

```
y = SQUARE (v);
```

присваивает `y` значение  $v^2$ . Но в операторе

```
y = SQUARE (v + 1);
```

переменной не присваивается значение  $(v + 1)^2$ , как ожидалось. Поскольку в определении этого макроса препроцессор выполняет подстановку текста вместо аргумента, здесь получится следующее выражение:

```
y = v + 1 * v + 1;
```

Чтобы решить проблему, нужно использовать в определении макроса `SQUARE` круглые скобки.

```
#define SQUARE(x) ((x) * (x))
```

Определение может показаться несколько странным, но помните, что `x` будет заменяться подставляемым выражением. Теперь оператор

```
y = SQUARE (v + 1);
```

будет правильно обработан как

```
y = ((v + 1) * (v + 1));
```

Следующий макрос позволяет создавать новые дроби из нашего класса Fraction.

```
#define MakeFract(x,y) ([[Fraction alloc] initWith: x over: y]])
```

Теперь для сложения дробей  $n1/d1$  и  $n2/d2$  можно писать такие выражения, как

```
myFract = MakeFract (1, 3); // Создание дроби 1/3
```

или

```
sum = [MakeFract (n1, d1) add: MakeFract (n2, d2)];
```

Макросы удобны для работы с условными выражениями. В следующей строке определяется макрос с именем MAX, определяющий максимальное из двух значений.

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

С помощью этого макроса можно писать такие операторы, как

```
limit = MAX (x + y, minValue);
```

Переменной limit присваивается максимальное из двух значений:  $x + y$  и minValue. Все определение MAX заключено в круглые скобки, чтобы правильно обрабатывать такие выражения, как

MAX (x, y) \* 100

Каждый аргумент тоже заключен в круглые скобки, чтобы правильно обрабатывать такие выражения, как

MAX (x & y, z)

Оператор & – это побитовый оператор AND, и он имеет меньший приоритет, чем оператор > в данном макросе. Без этих круглых скобок оператор > обрабатывался бы раньше побитового AND, что давало бы неверный результат.

В следующем макросе проверяется, является ли символ строчной буквой.

```
#define IS_LOWER_CASE(x) (((x) >= 'a') && ((x) <= 'z'))
```

С помощью этого макроса можно писать такие выражения, как

```
if (IS_LOWER_CASE (c))
```

...

Этот макрос можно даже использовать в определении другого макроса, чтобы преобразовывать символ из нижнего регистра в верхний (делать строчную букву прописной), не изменяя другие символы.

```
#define TO_UPPER(x) (IS_LOWER_CASE (x) ? (x) -'a' + 'A' : (x))
```

Здесь мы снова работаем со стандартным набором символов ASCII. В части II, когда будут описываться объекты-строки, вы увидите, как выполнять преобразование символов из одного регистра в другой для международных наборов символов (Unicode).

## Оператор #

Если поместить символ # перед параметром в определении макроса, препроцессор создаст строку-константу в стиле C из аргумента макроса при его вызове. Например, определение

```
#define str(x) # x
```

при последующем вызове

```
str (testing)
```

будет раскрыто препроцессором как

```
"testing"
```

Например, вызов printf

```
printf (str (Программировать на Objective-C интересно.\n));
```

эквивалентен

```
printf ("Программировать на Objective-C интересно.\n");
```

Препроцессор заключает в кавычки фактический аргумент макроса. Препроцессор сохраняет в аргументе все кавычки и обратные слэши. Поэтому вызов

```
str ("hello")
```

даст в результате

```
"\"hello\""
```

Более близкий к практике пример оператора # представляет следующее определение макроса.

```
#define printint(var) printf (# var " = %i\n", var)
```

Этот макрос используется для вывода значения целой переменной. Если count – переменная целого типа со значением 100, то оператор

```
printint (count);
```

будет раскрыт как

```
printf ("count" " = %i\n", count);
```

Компилятор выполнит конкатенацию двух смежных литеральных строк, чтобы создать одну строку. Поэтому после конкатенации оператор примет следующий вид.

```
printf ("count = %i\n", count);
```

## Оператор ##

В определении макроса оператор ## сливает два маркера. Он ставится перед именем параметра макроса (или после него). Препроцессор берет фактический аргумент, указанный при вызове макроса, и создает один маркер из этого аргумента и из маркера, который следует за ## или предшествует ##.

Предположим, что у нас имеется список переменных от x1 до x100. Мы можем написать макрос с вызовом printx, который принимает в качестве аргумента значение от 1 до 100 и выводит значение соответствующей переменной.

```
#define printx(n) printf ("%i\n", x ## n)
```

Часть x ## n указывает, что нужно взять маркеры, стоящие перед и после ## (соответственно букву x и аргумент n), и создать из них один маркер. Поэтому вызов

```
printx (20);
```

будет раскрываться следующим образом.

```
printf ("%i\n", x20);
```

В макросе printx можно использовать ранее определенный макрос printint, чтобы выводить имя переменной вместе с ее значением.

```
#define printx(n) printint(x ## n)
```

Вызов

```
printx (10);
```

сначала раскрывается как

```
printint (x10);
```

затем как

```
printf ("x10" "= %i\n", x10);
```

и, наконец, так:

```
printf ("x10 = %i\n", x10);
```

## Оператор #import

Программируя на Objective-C, вы постепенно разработаете для своих программ собственный набор макросов. Чтобы не вводить их в каждую новую программу, вы можете собрать все определения в один файл и включать свои макросы в программу с помощью оператора #import. Эти файлы обычно имеют расширение имени .h и называются заголовочными (*header*) или включаемыми (*include*) файлами.

Предположим, что мы пишем набор программ для метрических преобразований. Нам нужно задать операторы `#define` для констант, которые требуются при выполнении этих преобразований.

```
#define INCHES_PER_CENTIMETER 0.394 (дюйм/см)
#define CENTIMETERS_PER_INCH (1 / INCHES_PER_CENTIMETER) (см/дюйм)

#define QUARTS_PER_LITER 1.057 (кварты/литр)
#define LITERS_PER_QUART (1 / QUARTS_PER_LITER) (литр/кварты)

#define OUNCES_PER_GRAM 0.035 (унция/г)
#define GRAMS_PER_OUNCE (1 / OUNCES_PER_GRAM) (г/унция)
...
```

Мы ввели эти определения в файл с именем `metric.h`. Чтобы использовать определения из файла `metric.h`, в программе достаточно ввести следующую директиву препроцессора:

```
#import "metric.h"
```

Этот оператор должен появиться до ссылки на любые операторы `#define`, содержащиеся в файле `metric.h`. Обычно его помещают в начале исходного файла. Препроцессор ищет указанный файл в системе и копирует содержимое этого файла в то место программы, где находится оператор `#import`. Таким образом, любые операторы из этого файла обрабатываются так, как если бы они были непосредственно введены в программу в этом месте.

Кавычки, в которые заключено имя файла, показывают препроцессору, что файл нужно искать в одном или нескольких файловых каталогах (папках). Обычно поиск начинается с каталога, содержащего исходный файл, но в Xcode можно указать конкретные места для поиска, изменив настройки проекта (`Project Settings`).

Если заключить имя файла в угловые скобки (`< и >`)

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

то препроцессор будет искать `include`-файл только в специальном «системном» каталоге (или каталогах) `header`-файлов, но в текущем каталоге поиск выполниться не будет. И в этом случае при работе в Xcode можно изменить каталоги, выбрав в меню `Project, Edit Project Settings` (Изменить настройки проекта).

---

**Примечание.** При компиляции программ для этого раздела книги файл `Foundation.h` был импортирован из следующего каталога на моем компьютере:

```
/Developers/SDKs/MacOSX10.5.sdk/System/Library/Frameworks/Foundation.framework/Versions/C/Headers.
```

Покажем на примере конкретной программы, как работают include-файлы. Введем шесть приведенных выше операторов #define в файл с именем metric.h. Затем введем и запустим программу 12.1.

### Программа 12.1

```
/* Пример использования оператора #import
Примечание. В этой программе предполагается, что определения
заданы в файле с именем metric.h (галлон = 4 кварты) */

#import <Foundation/Foundation.h> #import «metric.h»
int main (int argc, char *argv[])

{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 float liters, gallons;

 NSLog (@"*** Liters to Gallons ***");
 NSLog (@>Enter the number of liters:");
 scanf ("%f", &liters);

 gallons = liters * QUARTS_PER_LITER / 4.0;
 NSLog (@"%g liters = %g gallons", liters, gallons);

 [pool drain];
 return 0;
}
```

### Вывод программы 12.1

```
*** Liters to Gallons *** (Литры в галлоны)
Enter the number of liters: (Введите количество литров)
55.75
55.75 liters = 14.7319 gallons. (55.75 литров = 14.7319 галлонов)
```

В программе 12.1 используется только одно определенное значение (QUARTS\_PER\_LITER) из include-файла metric.h. Тем не менее, это вполне показательный пример: после ввода в файл metric.h определения доступны в любой программе, где применяется соответствующий оператор #import.

Одним из наиболее важных преимуществ использования файла импорта является возможность централизовать все определения, что гарантирует использование в программах одного и того же значения. Кроме того, ошибки в include-файле достаточно исправить только в одном месте, не изменяя все программы, использующие это значение. Любую программу, в которой используется неверное значение, достаточно перекомпилировать, не редактируя.

Другие системные `include`-файлы содержат объявления для различных функций, хранящихся в базовой системной библиотеке С. Например, файл `limits.h` содержит системно-зависимые значения, которые задают размеры символьных и целых типов данных. Максимальный размер типа `int` определяется внутри этого файла именем `INT_MAX`, максимальный размер `unsigned long int` определяется с помощью `ULONG_MAX`, и т.д.

Файл `float.h` задает информацию о типах данных с плавающей точкой: `FLT_MAX` указывает максимальное число с плавающей точкой, а `FLT_DIG` – число десятичных цифр для точности типа `float`.

Файл `string.h` содержит объявления для библиотечных процедур копирования, сравнения и конкатенации, которые выполняют операции с символьными строками. Если вы работаете исключительно со строчными классами Foundation (см. главу 15), то эти процедуры вам не потребуются.

## Условная компиляция

Препроцессор Objective-C позволяет работать с *условной компиляцией* (*conditional compilation*).

Условная компиляция обычно применяется для создания одной программы, которую можно компилировать для выполнения в различных компьютерных системах. Она часто используется для включения или отключения в программе различных операторов, например, операторов отладки, которые выводят значения переменных или отслеживают последовательность выполнения программы.

### Операторы `#ifdef`, `#endif`, `#else` и `#ifndef`

К сожалению, программа может основываться на системно-зависимых параметрах, которые различны для разных процессоров (например, Power PC или Intel) в разных версиях операционной системы (например, Tiger или Leopard).

Для большой программы, содержащей много таких зависимостей от оборудования или программного обеспечения вам пришлось бы изменять много операторов типа `define`.

Чтобы свести к минимуму эту проблему, вы можете включить в программу значения определений операторов типа `define` для разных машин, используя средства условной компиляции препроцессора. Например, в следующих операторах для `DATADIR` определяется значение `"/uxn1/data"`, если ранее был определен символ `MAC_OS_X`, в противном случае присваивается значение `"\usr\data"`.

```
#ifdef MAC_OS_X
define DATADIR "/uxn1/data"
#else
define DATADIR "\usr\data"
#endif
```

После символа #, начинающего оператор препроцессора, можно помещать один или несколько пробелов.

Операторы `#ifdef`, `#else` и `#endif` действуют в соответствии с их именами. Если символ, указанный в строке `#ifdef`, уже определен (с помощью оператора `#define` или из командной строки при компиляции программы), компилятор обрабатывает последующие строки до `#else`, `#elif` или `#endif`; в противном случае они игнорируются.

Чтобы определить символ `POWER_PC` для препроцессора, достаточно оператора

```
#define POWER_PC 1
```

или

```
#define POWER_PC.
```

Как видно из примера, после определенного имени не требуется текста, чтобы выполнить проверку `#ifdef`. Компилятор также позволяет определить имя для препроцессора, если программа компилируется с использованием специальной опции в команде компилятора. В командной строке

```
gcc -framework Foundation -D POWER_PC program.m -
```

для препроцессора определяется имя `POWER_PC`, чтобы все операторы `#ifdef POWER_PC` внутри программы `program.m` давали значение `TRUE` (отметим, что `-D POWER_PC` следует ввести в командной строке до указания имени программы). Это средство позволяет определять имена без редактирования исходной программы.

В Xcode для добавления новых определенных имен и указания их значений нужно выбрать `Add User-Defined Setting` (Добавить пользовательское значение) в `Project Settings` (Настройки проекта).

После оператора `#ifndef` следуют такие же строки, как после оператора `#ifdef`. В нем последующие строки обрабатываются в том случае, если указанный символ *не* определен.

Как уже говорилось, условная компиляция полезна для отладки программ. Можно включить в программу много вызовов `printf` для вывода промежуточных результатов и отслеживания последовательности выполнения. Для запуска этих операторов можно выполнить их условную компиляцию в программе, если определено некоторое имя, например `DEBUG`. Следующую последовательность операторов можно использовать для вывода значений некоторых переменных, только если программа была откомпилирована с определением имени `DEBUG`.

```
#ifdef DEBUG
 NSLog(@"%@", @"User name = %s, id = %i", userName, userId);
#endif
```

В вашей программе может быть много таких отладочных операторов. При отладке программу можно компилировать с определением имени `DEBUG`, чтобы выполнялась компиляция отладочных операторов. Если программа работает правильно, ее можно перекомпилировать без определения `DEBUG`. Это позволяет сократить размер программы, поскольку все отладочные операторы не будут компилироваться.

## Операторы препроцессора `#if` и `#elif`

Оператор препроцессора `#if` представляет более общий подход к управлению условной компиляцией. Этот оператор позволяет проверить, равно ли нулю константное выражение. Если результат выражения не равен нулю, последующие строки до `#else`, `#elif` или `#endif` обрабатываются; в противном случае они пропускаются.

Рассмотрим следующие строки из файла Foundation NSString.h.

```
#if MAC_OS_X_VERSION_MIN_REQUIRED < MAC_OS_X_VERSION_10_5
#define NSMaximumStringLength (INT_MAX-1)
#endif
```

Здесь сравниваются значения определенных переменных `MAC_OS_X_VERSION_MIN_REQUIRED` и `MAC_OS_X_VERSION_10_5`. Если первое значение меньше второго, то обрабатывается следующий оператор `#define`; в противном случае он пропускается. Это позволяет задать максимальную длину строки, равную максимальному размеру целого типа минус 1, если программа компилируется в системе Mac OS X 10.5 или последующих версиях.

В строках с `#if` применяется также специальный оператор

```
defined (имя)
```

Следующие наборы операторов дают одинаковый результат:

```
#if defined (DEBUG)
...
#endif
```

и

```
#ifdef DEBUG
...
#endif
```

Следующие операторы включены в файл NSObjCRuntime.h, чтобы определить имя `NS_INLINE` (если оно не было определено раньше) в зависимости от конкретного компилятора.

```
#if !defined(NS_INLINE)
#if defined(__GNUC__)
 #define NS_INLINE static __inline_attribute__((always_inline))
#elif defined(__MWERKS__) || defined(__cplusplus)
 #define NS_INLINE static inline
#elif defined(_MSC_VER)
 #define NS_INLINE static __inline
#elif defined(__WIN32__)
 #define NS_INLINE static __inline_
#endif
#endif
```

Ниже приводится еще один типичный случай применения `#if`.

```
#if defined (DEBUG) && DEBUG
...
#endif
```

Операторы между `#if` и `#endif` будут обрабатываться в том случае, если имя `DEBUG` определено и имеет ненулевое значение.

## Оператор `#undef`

Сделать определенное имя неопределенным позволяет оператор `#undef`. Чтобы удалить определение имени, нужно написать строку

```
#undef имя
```

Следующий оператор удаляет определение `POWER_PC`.

```
#undef POWER_PC
```

Последующие операторы `#ifdef POWER_PC` или `#if defined (POWER_PC)` будут давать значение `FALSE`.

На этом заканчивается наше описание возможностей препроцессора. В приложении В приводятся дополнительные операторы препроцессора, которые не были описаны здесь.

## Упражнения

1. Найдите на своей машине файлы `limits.h` и `float.h`. Посмотрите, что находится в этих файлах. Если в эти файлы включены другие файлы, просмотрите их.
2. Определите макрос с именем `MIN`, который дает минимальное из двух значений. Напишите программу, которая проверяет определение этого макроса.
3. Определите макрос с именем `MAX3`, который дает максимальное из трех значений. Напишите программу, которая проверяет это определение.
4. Напишите макрос с именем `IS_UPPER_CASE`, который выдает ненулевое значение, если символ является прописной буквой.
5. Напишите макрос с именем `IS_ALPHABETIC`, который выдает ненулевое значение, если символ является буквой. Сделайте так, чтобы в этом макросе использовались макрос `IS_LOWER_CASE`, определенный в данной главе, и макрос `IS_UPPER_CASE`, определенный в упражнении 4.
6. Напишите макрос с именем `IS_DIGIT`, который выдает ненулевое значение, если символ является цифрой от 0 до 9. Используйте этот макрос в определении другого макроса с именем `IS_SPECIAL`, который выдает ненулевое значение, если символ является специальным символом (то есть не буквой и не цифрой). Обязательно используйте макрос `IS_ALPHABETIC` из упражнения 5.

7. Напишите макрос с именем `ABSOLUTE_VALUE`, который вычисляет абсолютное значение своего аргумента. Этот макрос должен правильно вычислять такие выражения, как

`ABSOLUTE_VALUE (x + delta)`

8. Рассмотрим определение макрона `printint` из этой главы

```
#define printx(n) printf ("%i\n", x ## n)
```

Можно ли использовать следующие строки для вывода значений 100 переменных от `x1` до `x100`? Почему?

```
for (i = 1; i <= 100; ++i)
 printx (i);
```

## Глава 13

# Базовые средства из языка С

В этой главе описываются средства языка Objective-C, взятые из базового языка программирования С. Вам необязательно знать их. Такие средства, как функции, структуры, указатели, объединения и массивы лучше изучать по мере необходимости. Поскольку С является процедурным языком, некоторые из этих средств противоречат основам объектно-ориентированного программирования и могут также не согласовываться с некоторыми стратегиями Foundation framework, например, с методологией выделения памяти или работой с символьными строками, содержащими символы из нескольких байтов.

---

**Примечание.** На уровне Objective-C существуют способы работы с многобайтными символами, но в Foundation имеется намного более удобное решение с помощью собственного класса `NSString`.

---

С другой стороны, в некоторых приложениях может потребоваться низкоуровневый подход. Например, при работе с большими массивами данных могут применяться встроенные структуры данных в виде массивов из Objective-C вместо объектов-массивов из Foundation (см. главу 15). Функции могут оказаться удобным средством для группировки повторяющихся операций и разбиения программы на модули.

Ознакомьтесь с данной главой, чтобы получить общее представление, и вернитесь к ней, когда закончите изучение части II (Foundation Framework), или пропустите ее и перейдите к части II, где описывается Foundation framework. Если вам придется поддерживать чей-то код или вы начнете изучать header-формы Foundation framework, вам могут встретиться некоторые конструкции, описанные в этой главе. Некоторые из типов данных Foundation, такие как `NSRange`, `NSPoint` и `NSRect`, требуют элементарного понимания описываемых здесь структур. В таких случаях вы всегда сможете вернуться к этой главе.

## Массивы

Язык Objective-C позволяет определить набор упорядоченных элементов данных, который называется *массивом* (*array*). В этом разделе описывается определение и управление массивами. В последующих разделах описывается использование массивов совместно с функциями, структурами, символьными строками и указателями.

Предположим, что вам нужно считать набор оценок (*grades*) и затем выполнить с ними некоторые операции, например, расположить их в порядке возрастания, вычислить среднее значение и найти медиану. Вы не можете выполнить эти операции, пока не введете все оценки.

В Objective-C вы можете определить переменную с именем *grades*, которая представляет не одно значение оценки, а весь набор оценок. Для ссылки на элементы этого набора используется число, которое называется *порядковым номером*, или *индексом* (*index* или *subscript*). В математике *i*-й элемент набора *x* обозначается как  $x_i$ ; в Objective-C он обозначается как

`x[i]`

Тем самым, выражение

`grades[5]`

соответствует элементу с номером 5 в массиве с именем *grades*. В Objective-C элементы массива начинаются с номера 0, поэтому

`grades[0]`

на самом деле обозначает первый элемент массива.

Отдельный элемент массива можно использовать в любом месте как обычную переменную. Например, элемент массива можно присвоить другой переменной с помощью оператора

`g = grades[50];`

Здесь значение, содержащееся в `grades[50]`, присваивается переменной *g*. В общем виде, если *i* объявлена как целая переменная, оператор

`g = grades[i];`

присваивает переменной *g* значение, содержащееся в элементе с номером *i* массива *grades*.

Чтобы сохранить значение в элементе массива, нужно указать этот элемент массива слева от знака равенства. В операторе

`grades[100] = 95;`

значение 95 сохраняется в элементе с номером 100 массива *grades*.

Для перебора элементов массива нужно изменять значение переменной, которая используется как индекс массива. Например, в цикле

```
for (i = 0; i < 100; ++i)
 sum += grades[i];
```

выполняется перебор первых 100 элементов массива `grades` (элементы 0-99), и значение каждой оценки (`grade`) добавляется к переменной `sum`. По окончании цикла `for` переменная `sum` будет содержать сумму первых 100 значений массива `grades` (если `sum` присвоено значение 0 до начала цикла).

Как и с другими типами переменных, необходимо объявить массив, прежде чем использовать его. Объявление массива – это объявление типа элементов, содержащихся в массиве (например, `int`, `float` или объект) и максимального числа элементов в массиве.

#### Определение

```
Fraction *fracts [100];
```

объявляет `fracts` как массив, содержащий 100 дробей (`fraction`). Для обозначения элементов этого массива нужно использовать индексы от 0 до 99. Выражение

```
fracts[2] = [fracts[0] add: fracts[1]];
```

представляет вызов метода `add:` класса `Fraction` для сложения первых двух дробей из массива `fracts` и сохранения результата в третьем элементе массива.

В программе 13.1 создается таблица из первых 15 чисел Фибоначчи. Попытайтесь предсказать ее результаты. Какая связь существует между числами этой таблицы?

#### Программа 13.1

```
// Программа генерации первых 15 чисел Фибоначчи
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 int Fibonacci[15], i;

 Fibonacci[0] = 0; /* by definition */

 Fibonacci[1] = 1; /* ditto */

 for (i = 2; i < 15; ++i)
 Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];

 for (i = 0; i < 15; ++i)
 NSLog(@"%@", Fibonacci[i]);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.1

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Первые два числа Фибоначчи, которые мы назовем  $F_0$  и  $F_1$ , определяются соответственно как 0 и 1. Затем каждое следующее число Фибоначчи  $F_i$  определяется как сумма двух предыдущих чисел Фибоначчи  $F_{i-2}$  и  $F_{i-1}$ . Например,  $F_2$  вычисляется как сумма значений  $F_0$  и  $F_1$ . В приведенной программе это соответствует вычислению `Fibonacci[2]` путем сложения значений `Fibonacci[0]` и `Fibonacci[1]`. Вычисление выполняется внутри цикла `for` для значений  $F_0$ - $F_{14}$  (то есть от `Fibonacci[2]` до `Fibonacci[14]`).

## Инициализация элементов массива

Элементам массива можно назначать начальные значения точно так же, как переменным при их объявлении. Для этого нужно просто перечислить начальные значения массива, начиная с первого элемента. Значения в списке разделяются запятыми, и весь список заключается в фигурные скобки.

В операторе

```
int integers[5] = { 0, 1, 2, 3, 4 };
```

элементу `integers[0]` присваивается значение 1, `integers[1]` – значение 1, `integers[2]` – значение 2, и т.д. Массивы символов инициализируются аналогичным образом. Например, в операторе

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

определяется массив символов, и пяти элементам этого массива присваиваются соответственно значения 'a', 'b', 'c', 'd' и 'e'.

Вы не обязаны полностью инициализировать весь массив. Если указано меньшее количество начальных значений, инициализируется соответствующее количество элементов; остальные значения задаются равными нулю. Например, в объявлении

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

инициализируются первые три значения массива `sample_data` (соответственно 100.0, 300.0 и 500.5), а остальным 497 элементам присваивается значение 0.

Заключая номер элемента в прямоугольные скобки, можно инициализировать определенные элементы массива в любом порядке. Например, в строках,

```
int x = 1233;
int a[] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

определяется массив `a` из 10 элементов (по максимальному указанному индексу) и для последнего элемента задается значение `x + 1` (1234). Кроме того, происходит инициализация первых трех элементов (значения 1, 2 и 3 соответственно).

## Массивы символов

В программе 13.2 показано, как использовать массив символов. Однако здесь есть одна особенность, требующая обсуждения. Вы уже видите ее?

### Программа 13.2

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

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 char word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
 int i;

 for (i = 0; i < 6; ++i)
 NSLog(@"%@", word[i]);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.2

```
H
e
l
l
o
!
```

Наиболее примечательной особенностью этой программы является объявление массива символов `word` без указания числа элементов. При определении массива без указания числа элементов размер массива определяется автоматически по числу инициализируемых элементов. Поскольку в программе 13.2 для массива `word` инициализируются шесть значений, язык Objective-C по умолчанию определяет, что данный массив содержит шесть элементов.

Этот принцип выполняется, если мы инициализируем каждый элемент массива в том месте, где определяется этот массив. В противном случае необходимо явным образом задать размер массива.

Если поместить завершающий нуль-символ ('\0') в конце массива символов, то создается *символьная строка (character string)*. Если в программе 13.2 заменить инициализацию массива word на строку

```
char word[] = { 'H', 'e', 'T', 'T', 'o', '!', '\0' };
```

то в дальнейшем можно выводить эту строку с помощью одного вызова NSLog, например,

```
NSLog(@"%@", word);
```

Символы форматирования %s указывают NSLog, что вывод символов должен продолжаться до тех пор, пока не будет достигнут завершающий нуль-символ. А именно этот символ мы поставили в конце массива word.

## Многомерные массивы

Все массивы, рассмотренные выше, являются линейными, то есть имеют одну размерность. Язык Objective-C позволяет определять массивы любой размерности. В этом разделе описываются двумерные массивы.

Типичным примером двумерного массива являются матрицы. Рассмотрим матрицу 4x5.

|    |    |    |    |    |
|----|----|----|----|----|
| 10 | 5  | -3 | 17 | 82 |
| 9  | 0  | 0  | 8  | -7 |
| 32 | 20 | 1  | 0  | 14 |
| 0  | 0  | 8  | 7  | 6  |

В математике для указания элемента матрицы используют два индекса. Если назвать эту матрицу  $M$ , то обозначение  $M_{i,j}$  будет указывать элемент  $i$ -й строки,  $j$ -го столбца, где  $i$  изменяется от 1 до 4 и  $j$  изменяется от 1 до 5. Обозначение  $M_{3,2}$  указывает значение 20, которое находится в третьей строке втором столбце этой матрицы. Аналогичным образом,  $M_{4,5}$  указывает элемент, который находится в четвертой строке пятом столбце (значение 6).

В Objective-C при обозначении элементов двумерной матрицы используются аналогичные обозначения. Но поскольку в Objective-C нумерация начинается с 0, первая строка матрицы имеет номер 0, и первый столбец матрицы тоже имеет номер 0. Поэтому приведенная выше матрица будет иметь следующие обозначения строк и столбцов.

| Строка (i) | Столбец (j) |    |    |    |    |
|------------|-------------|----|----|----|----|
|            | 0           | 1  | 2  | 3  | 4  |
| 0          | 10          | 5  | -3 | 17 | 82 |
| 1          | 9           | 0  | 0  | 8  | -7 |
| 2          | 32          | 20 | 1  | 0  | 14 |
| 3          | 0           | 0  | 8  | 7  | 6  |

Применяемое в математике обозначение  $M_{ij}$  заменяется в Objective-C обозначением

`M[i][j]`

Напомним, что первый индекс указывает номер строки, и второй индекс – номер столбца. Тем самым,

`sum = M[0][2] + M[2][4];`

означает сложение значения из строки 0 столбца 2 (-3) со значением из строки 2 столбца 4 (14) и присваивание результата (11) переменной `sum`.

Описание двумерных массивов выполняется так же, как для одномерных массивов; например,

`int M[4][5];`

означает объявление массива `M` как двумерного массива, содержащего 4 строки и 5 столбцов (всего 20 элементов). Каждый элемент этого массива должен содержать целое значение.

Двумерные массивы можно инициализировать аналогично одномерным. Перечисление элементов выполняется по строкам. Чтобы отделить инициализацию одной строки от другой, используются фигурные скобки. Таким образом, чтобы определить и инициализировать массив `M` с элементами из приведенной выше таблицы, можно использовать следующий оператор.

```
int M[4][5] = {
 { 10, 5, -3, 17, 82 },
 { 9, 0, 0, 8, -7 },
 { 32, 20, 1, 0, 14 },
 { 0, 0, 8, 7, 6 }
};
```

Уделите особое внимание синтаксису этого оператора. Отметим, что после каждой закрывающей фигурной скобки (кроме последней) требуется запятая. Использование внутренних пар фигурных скобок не является обязательным. Если они не указаны, инициализация происходит по строкам, поэтому предыдущий оператор можно было бы написать следующим образом.

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
 20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

Как и в случае одномерных массивов, можно инициализировать не весь массив. Например, в следующем операторе инициализируются только первые три элемента каждой строки матрицы.

```
int M[4][5] = {
 { 10, 5, -3 },
 { 9, 0, 0 },
 { 32, 20, 1 },
 { 0, 0, 8 }
};
```

Остальным значениям присваивается значение 0. Отметим, что в данном случае для правильной инициализации необходимы внутренние пары фигурных скобок. Без таких скобок были бы инициализированы первые две строки и первые два элемента третьей строки. (Проверьте это сами.)

## ФУНКЦИИ

Процедура `NSLog`, которую мы использовали в каждой программе этой книги, является примером функции. Каждая программа содержит функцию с именем `main`. Вернемся к нашей первой программе (программа 2.1), где на терминал выводилась фраза «Программировать на Objective-C интересно».

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

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

 NSLog (@"Программировать интересно.");
 [pool drain];
 return 0;
}
```

Следующая функция с именем `printMessage` выводит тот же текст.

```
void printMessage (void)
{
 NSLog (@"Программировать интересно.");
}
```

Единственным отличием между `printMessage` и функцией `main` из программы 2.1 является первая строка. В первой строке определения функции компилятору сообщаются четыре факта об этой функции.

- Кто может вызывать ее.
- Тип значения, которое она возвращает.
- Ее имя.
- Количество и тип аргументов, которые она принимает.

В первой строке определения функции printMessage компилятору сообщается, что имя функции printMessage и что она не возвращает никакого значения (первое применение ключевого слова void). В отличие от методов, тип возвращаемого значения функции не заключают в круглые скобки. Компилятор выдаст сообщение об ошибке, если вы сделаете это.

После сообщения, что printMessage не возвращает значений, ключевое слово void применяется во второй раз, указывая, что этой функции не передаются никакие аргументы.

Напомним, что main – это специальное имя в системе Objective-C, которое указывает, где должно начаться выполнение данной программы. main необходимо указывать всегда. Мы можем добавить функцию main к приведенному выше коду, чтобы получить законченную программу (см. программу 13.3).

### Программа 13.3

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

void printMessage (void)
{
 NSLog (@"Программировать интересно.");
}

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

 printMessage ();
 [pool drain];
 return 0;
}
```

### Вывод программы 13.3

Программировать интересно.

Программа 13.3 состоит из двух функций: printMessage и main. Поскольку printMessage не принимает никаких аргументов, при ее вызове используется просто пара круглых скобок.

## Аргументы и локальные переменные

В главе 5 мы разрабатывали программы для вычисления треугольных чисел. Здесь мы определим функцию с именем calculateTriangularNumber для генерации треугольного числа и будем вызывать ее. В качестве аргумента функции указывается номер треугольного числа. Функция будет вычислять нужное число и выводить результаты. В программе 13.4 показана функция для выполнения этой задачи и процедура main для ее проверки.

### Программа 13.4

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

// Функция для вычисления n-го треугольного числа

void calculateTriangularNumber (int n)
{
 int i, triangularNumber = 0;

 for (i = 1; i <= n; ++i)
 triangularNumber += i;

 NSLog (@"Triangular number %i is %i", n, triangularNumber);
}

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

 calculateTriangularNumber (10);
 calculateTriangularNumber (20);
 calculateTriangularNumber (50);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.4

```
Triangular number 10 is 55 (10-е треугольное число равно 55)
Triangular number 20 is 210
Triangular number 50 is 1275
```

Рассмотрим первую строку функции calculateTriangularNumber.

```
void calculateTriangularNumber (int n)
```

Она указывает компилятору, что calculateTriangularNumber — это функция, которая не возвращает никакого значения (ключевое слово `void`) и которой передается один аргумент с именем `n` типа `int`. Отметим еще раз, что тип аргумента нельзя помещать в круглые скобки, как мы привыкли делать при написании методов.

Открывающая фигурная скобка указывает начало определения функции. Поскольку нам нужно вычислять  $n$ -е треугольное число, мы должны задать переменную для сохранения значения треугольного числа при его вычислении. Нам нужна также переменная, действующая как индекс цикла. Для этих целей переменные `TriangularNumber` и `i` определяются и объявляются с типом `int`. Их определение и инициализацию мы выполняем так же, как для других переменных внутри процедуры `main` в предыдущих программах.

Локальные переменные действуют в функциях так же, как в методах. Если для переменной внутри функции задано начальное значение, это начальное значение присваивается переменной при каждом вызове данной функции.

Переменные, определенные внутри функции, называются *автоматически локальными* (*automatic local*) переменными, поскольку они автоматически «создаются» каждый раз, когда происходит вызов данной функции, а их значения являются локальными по отношению к данной функции.

*Статические локальные* (*Static local*) переменные описываются с помощью ключевого слова `static`, сохраняют свои значения при вызовах функций и имеют начальные значения по умолчанию, равные 0.

Значение локальной переменной доступно только внутри функции, где определена эта переменная. Ее значение нельзя получить с помощью непосредственного доступа извне этой функции.

Вернемся к нашему примеру. После определения локальных переменных в функции вычисляется треугольное число, и результаты выводятся на терминал. Закрывающая фигурная скобка определяет конец функции.

Внутри процедуры `main` при первом вызове функции `calculateTriangularNumber` ей передается в качестве аргумента значение 10. После этого управление передается непосредственно в функцию, где значение 10 становится значением формального параметра `n` внутри этой функции. Затем функция вычисляет значение 10-го треугольного числа и выводит результат.

При следующем вызове `calculateTriangularNumber` передается аргумент 20. Значение 20 становится значением формального параметра `n` внутри функции. Затем функция вычисляет значение 20-го треугольного числа и выводит результат.

## Возвращение результатов функций

Как и методы, функции могут возвращать значение. Тип значения, возвращаемого с помощью оператора `return`, должен быть согласован с типом возвращаемого значения, объявленного для этой функции. В объявлении функции, которое начинается с

```
float kmh_to_mph (float km_speed)
```

содержится определение функции с именем `kmh_to_mph`, которая принимает один аргумент типа `float` с именем `km_speed` и возвращает значение тоже типа `float`. Аналогичным образом,

```
int gcd (int u, int v)
```

определяет функцию `gcd` (наибольший общий делитель) с целыми аргументами `u` и `v` и возвращает целое значение.

Мы перепишем алгоритм определения наибольшего общего делителя, используемый в программе 5.7, в форме функции. Этой функции передаются два аргумента в виде двух чисел, для которых нужно определить их наибольший общий делитель (`gcd`, greatest common divisor), см. программу 13.5.

**Программа 13.5**

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

// Эта функция ищет наибольший общий делитель двух
// неотрицательных целых значений и возвращает соответствующий результат

int gcd (int u, int v)
{
 int temp;

 while (v != 0)
 {
 temp = u % v;
 u = v;
 v = temp;
 }

 return u;
}

main ()
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 int result;

 result = gcd (150, 35);
 NSLog (@"The gcd of 150 and 35 is %i", result);

 result = gcd (1026, 405);
 NSLog (@"The gcd of 1026 and 405 is %i", result);

 NSLog (@"The gcd of 83 and 240 is %i", gcd (83, 240));
 [pool drain];
 return 0;
}
```

**Вывод программы 13.5**

The gcd of 150 and 35 is 5 (Наибольший общий делитель 150 и 35 равен 5)  
 The gcd of 1026 and 405 is 27  
 The gcd of 83 and 240 is 1

Функция `gcd` принимает два целых аргумента. Данная функция имеет доступ к этим аргументам через имена формальных параметров `u` и `v`. После объявления переменной `temp` типа `int` программа выводит на терминал значения аргу-

ментов и и в вместе с соответствующим сообщением. Затем функция вычисляет и возвращает наибольший общий делитель этих двух целых чисел.

В строке

```
result = gcd (150, 35);
```

происходит вызов функции `gcd` с аргументами 150 и 35 и сохранение значения, которое возвращает функция, в переменной `result`.

Если не указан тип возвращаемого значения функции, компилятор предполагает, что функция возвращает целое значение (если она вообще возвращает значение). Многие программисты не указывают тип возвращаемого значения, если функция должна возвращать целые значения. Однако такой подход противоречит практике надежного программирования. Компилятор предупредит, что для возвращаемого значения задан по умолчанию тип `int`, но это признак того, что вы делаете что-то неверно!

Тип по умолчанию для возвращаемого значения отличается от методов. Если для метода не указан тип возвращаемого значения, компилятор предполагает, что метод возвращает значение типа `id`. Но для метода тоже следует указывать тип возвращаемого значения.

### Объявление типов возвращаемых значений и типов аргументов

По умолчанию компилятор Objective-C предполагает тип `int` для возвращаемого значения. Точнее говоря, если вызывается функция, компилятор предполагает, что эта функция возвращает значение типа `int`, за исключением следующих случаев.

- Эта функция определена в программе до того, как встретился вызов этой функции.
- Значение, возвращаемое функцией, объявлено до того, как встретился вызов этой функции. Объявление типов возвращаемого значения и аргументов для функции называется *объявлением прототипа* (*prototype*).

Объявление функции используется не только для объявления типа возвращаемого значения функции, но также для того, чтобы указать компилятору, сколько аргументов и каких типов передавать функции. Это аналогично объявлению методов в секции `@interface` при объявлении нового класса.

Чтобы объявить `absoluteValue` как функцию, которая возвращает значение типа `float` и принимает один аргумент типа `float`, можно использовать следующее объявление прототипа.

```
float absoluteValue (float);
```

Достаточно указать только тип аргумента в круглых скобках, но не его имя. Вы можете дополнительно указать после типа «фиктивное» имя, например,

```
float absoluteValue (float x);
```

Это имя не обязательно должно совпадать с именем в определении функции (в любом случае, компилятор его игнорирует).

Для надежности при объявлении прототипа скопируйте первую строку из фактического объявления функции. Не забудьте поставить в конце точку с за-

пятой. Если функция принимает переменное число параметров (как в случае с NSLog и scanf), об этом нужно информировать компилятор. Объявление

```
void NSLog (NSString *format, ...);
```

указывает компилятору, что NSLog принимает объект типа NSString в качестве первого аргумента, после которого следует любое число дополнительных аргументов. NSLog объявляется в специальном файле Foundation/Foundation.h<sup>1</sup>, и поэтому в начале каждой программы мы помещаем строку

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

Без этой строки компилятор может предполагать, что NSLog принимает фиксированное число аргументов, что может привести неверному генерируемому коду.

Компилятор автоматически преобразует числовые аргументы в соответствующие типы при вызове функции, если вы поместили определение функции или объявили функцию и ее аргументы до этого вызова.

Ниже приводятся некоторые сведения о функциях.

- По умолчанию компилятор предполагает, что функция возвращает значение типа int.
- Определяя функцию с возвращаемым значением типа int, определите ее таким же образом.
- Определяя функцию, которая не возвращает никакого значения, определите ее с ключевым словом void.
- Компилятор преобразует аргументы для согласования с аргументами, которые ожидает функция, только если вы ранее определили или объявили эту функцию.

Для надежности объявляйте все функции в своей программе, даже если они определены до их вызова. Лучше всего поместить объявления ваших функций в header-файл и затем просто импортировать этот файл в ваши модули.

Функции по умолчанию являются *внешними* (*external*). Область действия по умолчанию для функции устроена так, чтобы ее могли вызывать любые функции или методы, содержащиеся в любых файлах, которые связаны с этой функцией. Вы можете ограничить область действия функции, сделав ее статической. Для этого нужно поместить перед объявлением функции ключевое слово static, как показано ниже.

```
static int gcd (int u, int v)
{
 ...
}
```

Статическую функцию могут вызывать только другие функции или методы, которые находятся в одном файле с определением этой функции.

---

<sup>1</sup> Вообще говоря, он определяется в файле NSObjCRuntime.h, который импортируется из файла Foundation.h

## Функции, методы и массивы

Чтобы передать функции или методу один элемент массива, его нужно указать как аргумент. Например, если функция `squareRoot` вычисляет квадратные корни, и мы хотим получить квадратный корень от `averages[i]` и присвоить результат переменной `sq_root_result`, то можно написать оператор

```
sq_root_result = squareRoot (averages[i]);
```

Передача всего массива функции или методу выполняется иначе. При вызове функции или метода нужно указать только имя этого массива (без индексов). Например, если `grade_scores` был объявлен как массив, содержащий 100 элементов, выражение

```
minimum (grade_scores)
```

будет передавать все 100 элементов, содержащихся в массиве `grade_scores`, функции с именем `minimum`. Конечно, функция `minimum` должна ожидать передачи всего массива как аргумента и иметь соответствующее объявление формального параметра.

Следующая функция ищет минимальное целое значение в массиве с указанным количеством элементов.

```
// Функция для поиска минимума в массиве

int minimum (int values[], int numElements)
{
 int minValue, i;

 minValue = values[0];

 for (i = 1; i < numElements; ++i)
 if (values[i] < minValue)
 minValue = values[i];

 return (minValue);
}
```

В соответствии с определением, функция `minimum` принимает два аргумента. Первый аргумент — это массив, минимум значений которого мы хотим найти, а второй аргумент — это число элементов в данном массиве. Прямоугольные скобки, которые непосредственно следуют после `values` в заголовке функции, информируют компилятор Objective-C, что `values` — это массив с целыми значениями. Компилятору не нужен размер этого массива.

Формальный параметр `numElements` используется как верхний предел в операторе `for`. С помощью оператора `for` выполняется перебор всего массива от элемента `values[1]` до последнего элемента `values[numElements-1]`.

Если функция или метод изменяет значение элемента массива, это изменение вносится в исходный массив, переданный функции или методу. Это изменение остается в силе даже после того, как функция или метод закончит свое выполнение.

Работа с целым массивом происходит иным образом, чем с простой переменной или элементом массива (значения которых функция или метод не может изменить). Мы уже говорили, что при вызове функции или метода значения, передаваемые как аргументы, копируются в соответствующие формальные параметры, но при работе с массивами содержимое всего массива не копируется в массив формального параметра. Вместо этого передается указатель, показывающий, где в памяти компьютера размещен этот массив. Таким образом, любые изменения, внесенные в массив формального параметра, на самом деле вносятся в исходный массив, а не в копию этого массива. Поэтому при выходе из функции или метода эти изменения остаются действительными.

### Многомерные массивы

Элемент многомерного массива можно передавать функции или методу как обычную переменную или элемент одномерного массива. В строке

```
result = squareRoot (matrix[i][j]);
```

происходит вызов функции `squareRoot` с передачей в качестве аргумента значения, содержащегося в `matrix[i][j]`.

В качестве аргумента можно передать весь многомерный массив так же, как одномерный массив: нужно просто указать имя самого массива. Например, если матрица `measuredValues` объявлена как двумерный массив с целыми значениями, то оператор Objective-C

```
scalarMultipliy (measuredValues, constant);
```

можно использовать для вызова функции, которая умножает каждый элемент этой матрицы на значение константы. Из этого, конечно, следует, что сама функция может изменять значения, содержащиеся в массиве `measuredValues`. Все, что говорилось выше об одномерных массивах, применимо и к многомерным массивам. Изменения, вносимые внутри функции в любой элемент массива формального параметра, являются также изменениями в массиве, который был передан функции.

Мы уже говорили, что в случае объявления одномерного массива как формального параметра нам не нужен конкретный размер массива. Мы просто используем пару прямоугольных скобок, чтобы информировать компилятор Objective-C, что параметр является массивом. В случае многомерных массивов это применимо лишь отчасти. Для двумерного массива можно не указывать число строк массива, но объявление должно содержать число столбцов массива. Оба объявления

```
int arrayValues[100][50]
```

и

```
int arrayValues[][50]
```

являются допустимыми для массива формального параметра с именем `arrayValues`, который содержит 100 строк и 50 столбцов. Однако оба объявления

```
int arrayValues[100][]
```

и

```
int arrayValues[][][]
```

не являются допустимыми, поскольку необходимо указать число столбцов массива.

## Структуры

Помимо массивов, язык Objective-C содержит еще одно средство группирования элементов — *структур*.

Предположим, что нам нужно нам нужно сохранить внутри программы дату (например, 18/07/09), чтобы использовать ее для заголовка результатов программы или для вычислений. Обычный способ сохранения даты — это присвоить месяц целой переменной с именем *month*, день — целой переменной *day* и год — целой переменной *year*. Поэтому здесь вполне подойдут операторы

```
int month = 7, day = 18, year = 2009;
```

Но как быть, если в программе требуется сохранять несколько дат? Следовало бы сгруппировать эти наборы из трех переменных.

В языке Objective-C мы можем определить структуру с именем date (дата), которая состоит из трех компонентов, представляющих месяц, день и год. Синтаксис такого определения достаточно очевиден.

```
struct date
```

```
{
 int month;
 int day;
 int year;
};
```

Только что определенная структура *date* содержит три целых компонента с именами *month*, *day* и *year*. Такое определение даты — новый тип переменных, переменные теперь можно объявлять с типом *struct date*, как в следующем определении:

```
struct date today;
```

Мы можем определить переменную такого же типа purchaseDate (дата покупки) с помощью отдельного определения

```
struct date purchaseDate;
```

или просто включить оба определения в одну строку:

```
struct date today, purchaseDate;
```

В отличие от переменных типа *int*, *float* или *char*, при работе со структурными переменными требуется специальный синтаксис. Для доступа к компоненту структуры нужно указать имя структурной переменной, после которого следует точка (она называется *оператором «точка» – dot operator*) затем имя компо-

нента структуры. Например, чтобы задать значение 21 для компонента day переменной today, нужно написать

```
today.day = 21;
```

Отметим, что между именем переменной, точкой и именем компонента не допускаются пробелы.

Вы можете возразить, что мы уже использовали, казалось бы, такой же оператор для вызова свойства объекта. Вспомним, что оператор

```
myRect.width = 12;
```

вызывает метод-установщика (с именем `setWidth`) объекта класса `Rectangle`, передавая ему значение аргумента 12. Здесь нет никакой путаницы: компилятор сам определяет, что находится справа от оператора «точка», — структура или объект, — и выполняет соответствующую обработку.

Вернемся к примеру `struct date`, чтобы задать значение 2010 для компонента `year` в структуре `today`.

```
today.year = 2010;
```

И, наконец, чтобы проверить, что значение `month` равно 12, можно использовать оператор

```
if (today.month == 12)
 next_month = 1;
```

В программе 13.6 реализуется то, что мы обсуждали выше.

### Программа 13.6

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

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

 struct date
 {
 int month;
 int day;
 int year;
 };

 struct date today;

 today.month = 9;
 today.day = 25;
 today.year = 2009;

 NSLog (@"Today's date is %i/%i/.2i.", today.month,
 today.day, today.year % 100);
```

```
[pool drain];
return 0;
}
```

### Вывод программы 13.6

Today's date is 9/25/09. (Текущая дата - 25.9.09)

В первом операторе внутри main определяется структура с именем date, которая состоит из трех целых компонентов: month, day и year. Во втором операторе объявляется переменная today с типом struct date. Таким образом, в первом операторе просто определяется, как выглядит структура даты для компилятора Objective-C, и не требуется никакого резервирования памяти внутри компьютера. Во втором операторе объявляется переменная типа struct date, и здесь происходит резервирование памяти для хранения трех целых компонентов структурной переменной today.

После присваивания значений соответствующий вызов NSLog выводит значения, содержащиеся в этой структуре. Вычисляется остаток от деления today.year на 100, поэтому функция NSLog выводит для года только две цифры. Символы формата %.2i в обращении к NSLog указывают вывод не менее двух символов, в результате чего для года выводится ведущий нуль.

При вычислении выражений компоненты структуры подчиняются таким же правилам, что и обычные переменные в языке Objective-C. Деление целого компонента структуры на другое целое значение выполняется как деление целых, например

```
century = today.year / 100 + 1;
```

Напишем несложную программу, которая принимает на входе текущую дату и выводит завтрашнюю дату (tomorrow). На первый взгляд это кажется совсем простой задачей. Нужно запросить у пользователя ввод текущей даты и затем вычислить завтрашнюю дату с помощью следующего набора операторов.

```
tomorrow.month = today.month;
tomorrow.day = today.day + 1;
tomorrow.year = today.year;
```

Конечно, для большинства дат это подходит, но два случая будут реализованы неверно.

- Текущая дата приходится на конец месяца.
- Текущая дата приходится на конец года (то есть на 31 декабря).

Чтобы определить, приходится ли текущая дата на конец месяца, нужно задать массив целых значений, соответствующих числу дней каждого месяца. Поиск в этом массиве даст число дней месяца (см. программу 13.7).

### Программа 13.7

```
// Программа определения завтрашней даты
#import <Foundation/Foundation.h>
```

```
struct date
{
 int month;
 int day;
 int year;
};

// Функция для вычисления завтрашней даты

struct date dateUpdate (struct date today)
{
 struct date tomorrow;
 int numberOfDays (struct date d);

 if (today.day != numberOfDays (today))
 {
 tomorrow.day = today.day + 1;
 tomorrow.month = today.month;
 tomorrow.year = today.year;
 }
 else if (today.month == 12) // end of year
 {
 tomorrow.day = 1;
 tomorrow.month = 1;
 tomorrow.year = today.year + 1;
 }
 else
 {
 // конец месяца
 tomorrow.day = 1;
 tomorrow.month = today.month + 1;
 tomorrow.year = today.year;
 }

 return (tomorrow);
}

// Функция для поиска числа дней в месяце

int numberOfDays (struct date d)

{
 int answer;
 BOOL isLeapYear (struct date d);
 int daysPerMonth[12] =
 { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

 if (isLeapYear (d) == YES && d.month == 2)
```

```
 answer = 29;
 else
 answer = daysPerMonth[d.month - 1];

 return (answer);
}

// Функция, определяющая, является ли год високосным

BOOL isLeapYear (struct date d)
{
 if ((d.year % 4 == 0 && d.year % 100 != 0) ||
 d.year % 400 == 0)
 return YES;
 else
 return NO;
}

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 struct date dateUpdate (struct date today);
 struct date thisDay, nextDay;

 NSLog(@"Enter today's date (mm dd yyyy): ");
 scanf ("%i%i%i", &thisDay.month, &thisDay.day,
 &thisDay.year);

 nextDay = dateUpdate (thisDay);

 NSLog(@"Tomorrow's date is %i/%i/.%i.",nextDay.month,
 nextDay.day, nextDay.year % 100);

 [pool drain];
 return 0;
}
```

### Выход программы 13.7

```
Enter today's date (mm dd yyyy): (Введите текущую дату (мм дд гггг))
2 28 2012
Tomorrow's date is 2/29/12. (Завтрашняя дата)
```

### Выход программы 13.7 (повторный запуск)

```
Enter today's date (mm dd yyyy):
10 2 2009
Tomorrow's date is 10/3/09.
```

**Вывод программы 13.7 (повторный запуск)**

Enter today's date (mm dd yyyy):

**12 31 2010**

Tomorrow's date is 1/1/10.

Хотя мы не работаем в этой программе с классами, здесь был импортирован файл Foundation.h, поскольку нам нужен тип BOOL и определенные имена YES и NO. Они определены в этом файле.

Отметим, что определение структуры данных представлено в первую очередь и вне какой-либо функции. Определения структур данных действуют аналогично переменным. Если структура определена внутри какой-либо функции, только эта функция «знает» о ее существовании, и тогда это *локальное определение структуры*. Если структура определена вне любой функции, это определение является *глобальным*. Глобальное определение структуры позволяет объявлять любые последующие переменные (внутри или вне функции) как структуру этого типа. Определения структур, которые используются в нескольких файлах, обычно централизуются в одном header-файле и затем импортируются в те файлы, где нужна эта структура.

Внутри процедуры main объявление

```
struct date dateUpdate (struct date today);
```

указывает компилятору, что функция dateUpdate принимает в качестве аргумента структуру типа date и возвращает структуру такого же типа. Это объявление не обязательно, поскольку компилятор уже «видел» конкретное объявление функции в этом файле, но принято в практике надежного программирования. Если вы в дальнейшем разделите определение этой функции и main на отдельные исходные файлы, объявление будет необходимо.

Как и для обычных переменных (но не для массивов), любые изменения, которые вносятся функцией в значения, содержащиеся в аргументе структуры, не оказывают влияния на исходную структуру. Они влияют только на копию этой структуры, которая создается при вызове функции.

После ввода даты и ее сохранения в структурной переменной thisDay типа date происходит вызов функции dateUpdate.

```
nextDay = dateUpdate (thisDay);
```

Здесь происходит вызов функции dateUpdate с передачей этой функции структуры даты thisDay.

Внутри функции dateUpdate объявление прототипа

```
int numberOfDays (struct date d);
```

информирует компилятор Objective-C, что функция numberOfDays возвращает целое значение и принимает один аргумент типа struct date.

Оператор

```
if (today.day != numberOfDays (today))
```

указывает, что структура `today` должна передаваться как аргумент функции `numberOfDays`. Внутри этой функции должно быть соответствующее объявление, информирующее систему, что аргументом должна быть структура, как в следующей строке.

```
int numberOfDays (struct date d)
```

Сначала в функции `numberOfDays` нужно выполнить проверку на високосный год и определить, не является ли месяц февралем. Для первой проверки вызывается другая функция с именем `isLeapYear`.

Функция `isLeapYear` вполне понятна; она проверяет год, содержащийся в структуре типа `date`, которая передается как аргумент, и возвращает значение YES в случае високосного года или NO в противном случае.

Вы должны понять иерархию вызовов функций в программе 13.7. Функция `main` вызывает `dateUpdate`, которая вызывает `numberOfDays`, которая, в свою очередь, вызывает функцию `isLeapYear`.

## Инициализация структур

Инициализация структур аналогична инициализации массивов: значения компонентов просто перечисляются в фигурных скобках с применением запятой в качестве разделителя.

Чтобы инициализировать переменную `today` структуры типа `date`, задав значение Июль, 2 число, 2011 год, можно использовать оператор

```
struct date today = { 7, 2, 2011 };
```

Как и при инициализации массива, можно указывать значения не для всех компонентов структуры. Например, в операторе

```
struct date today = { 7 };
```

задается значение 7 для компонента `today.month`, но не задаются начальные значения для `today.day` и `today.year`. В таких случаях начальные значения по умолчанию не определены.

Для конкретных компонентов можно выполнять инициализацию в любом порядке, используя в списке инициализации форму записи

.компонент = значение

например,

```
struct date today = { .month = 7, .day = 2, .year = 2011 };
```

и

```
struct date today = { .year = 2011 };
```

В последнем примере этой структуры задается только значение года, равное 2011. Остальные два компонента не определены.

## Массивы структур

Работа с массивами структур не представляет особых сложностей. Например, в

```
struct date birthdays[15];
```

определяется массив `birthdays` (дни рождения), содержащий 15 элементов типа `struct date`. Нужный элемент в массиве структур указывается естественным образом. Например, чтобы задать для второго элемента в массиве `birthdays` дату дня рождения 22 февраля 1996 г., можно написать последовательность

```
birthdays[1].month = 2;
birthdays[1].day = 22;
birthdays[1].year = 1996;
```

Оператор

```
n = numberOfDays (birthdays[0]);
```

передает первый элемент этого массива функции `numberOfDays`, чтобы определить, сколько дней содержит указанный в дате месяц.

## Структуры внутри структур

Objective-C предоставляет большие возможности в определении структур. Например, можно определить структуру, содержащую в качестве своих компонентов другие структуры, или определять структуры, содержащие массивы.

Вы уже видели, как логически группировать месяц, день и год (`month`, `day` и `year`) в структуре с именем `date`. Предположим, что у нас есть аналогичная структура с именем `time` для группирования часов, минут и секунд (`hour`, `minutes` и `seconds`). В некоторых приложениях может потребоваться логическая группировка структур `date` и `time` – например, для списка событий с определенной датой и временем.

Из предыдущего описания следует, что нам нужны удобные средства, объединяющие дату и время. Это можно сделать в Objective-C, определив новую структуру (например, `date_and_time`), компоненты которой содержатся в элементах `date` и `time`.

```
struct date_and_time
{
 struct date sdate;
 struct time stime;
};
```

Первый компонент этой структуры имеет тип `struct date` и называется `sdate`. Второй компонент структуры `date_and_time` имеет тип `struct time` и называется `stime`. В этом определении структуры `date_and_time` требуется, чтобы структура `date` и структура `time` были предварительно определены для компилятора.

Теперь можно определять переменные типа `struct date_and_time`:

```
struct date_and_time event;
```

Для указания ссылки на структуру date в переменной event используется тот же синтаксис:

```
event.sdate
```

Мы можем вызвать функцию dateUpdate, указав дату в качестве аргумента, чтобы получить результат в том же месте.

```
event.sdate = dateUpdate (event.sdate);
```

То же самое можно сделать для структуры time, содержащейся в структуре date\_and\_time.

```
event.stime = timeUpdate (event.stime);
```

Для указания определенного компонента внутри одной из этих структур нужно добавить точку и имя этого компонента:

```
event.sdate.month = 10;
```

В этом операторе задается значение 10 (октябрь) для компонента month структуры date переменной event. В операторе

```
++event.stime.seconds;
```

значение компонента seconds структуры time увеличивается на 1.

Инициализировать переменную event можно уже известным способом:

```
struct date_and_time event =
{ { 12, 17, 1989 }, { 3, 30, 0 } };
```

Здесь для переменной event задается дата 17 декабря 1989 г. и время 3:30:00.

Массив структур date\_and\_time можно задать с помощью объявления

```
struct date_and_time events[100];
```

Здесь объявляется, что массив events содержит 100 элементов типа struct date\_and\_time. Для указания 4-го элемента date\_and\_time в массиве нужно написать events[3], а для 25-й даты в массиве при обращении к функции dateUpdate нужно написать

```
events[24].sdate = dateUpdate (events[24].sdate);
```

Чтобы записать 12 часов дня в первый элемент этого массива, применяется следующий набор операторов.

```
events[0].stime.hour = 12;
events[0].stime.minutes = 0;
events[0].stime.seconds = 0;
```

## Дополнительно о структурах

Определять структуру можно достаточно гибко. Во-первых, можно объявить переменную с определенным структурным типом одновременно с объявлением структуры. Для этого достаточно включить имя переменной (переменных) с завершающей точкой с запятой в определение структуры. Например, в следующем операторе определяется структура `date` и объявляются переменные `todayDate` и `purchaseDate` этого типа.

```
struct date
{
 int month;
 int day;
 int year;
} todayDate, purchaseDate;
```

Вы можете присвоить этим переменным начальные значения обычным образом. Ниже определяется структура `date` и переменная `todayDate` с указанием начальных значений.

```
struct date
{
 int month;
 int day;
 int year;
} todayDate = { 9, 25, 2010 };
```

Если все переменные определенного структурного типа определяются вместе с определением структуры, то можно не задавать имя структуры. Ниже определяется массив с именем `dates`, содержащий 100 элементов.

```
struct
{
 int month;
 int day;
 int year;
} dates[100];
```

Каждый элемент является структурой, содержащей три целых компонента: `month`, `day` и `year`. Поскольку здесь не указывается имя структуры, при последующем объявлении переменных того же типа структуру придется явно определить снова.

### Битовые поля

В Objective-C имеются два способа упаковки информации. Первый способ – это представление данных внутри переменной целого типа и последующий доступ к нужным битам с помощью побитовых операций, описанных в главе 4.

Второй способ – это определение структуры упакованной информации с помощью конструкции Objective-C, которая называется *битовым полем* (*bitfield*). Для этого применяется специальный синтаксис в определении структуры, позволяющий определять поле битов и присваивать ему имя.

Для определения битовых полей можно определить структуру, например, с именем packedStruct.

```
struct packedStruct
{
 unsigned int f1:1;
 unsigned int f2:1;
 unsigned int f3:1;
 unsigned int type:4;
 unsigned int index:9;
};
```

В соответствии с этим определением, структура packedStruct состоит из пяти компонентов. Первый компонент, f1, имеет тип unsigned int. Обозначение :1 после имени компонента указывает, что компонент будет содержаться в 1 бите. Флаги f2 и f3 определяются аналогичным образом. Компонент type занимает в соответствии с определением 4 бита, а компонент index имеет длину 9 битов.

Компилятор автоматически упаковывает подряд эти битовые поля. Удобство этого подхода состоит в том, что поля переменной типа packedStruct можно указывать как обычные компоненты структуры. Например, если объявлена переменная packedData

```
struct packedStruct packedData;
```

вы можете легко присвоить значение 7 полю type переменной packedData с помощью простого оператора

```
packedData.type = 7;
```

Можно присвоить этому полю значение переменной n с помощью оператора

```
packedData.type = n;
```

В последнем случае вы можете не думать о том, что значение n может оказаться слишком большим, чтобы уместиться в поле type; в packedData.type будут записаны только младшие 4 бита.

Извлечь значение из битового поля тоже нетрудно. Например, с помощью оператора

```
n = packedData.type;
```

извлекается значение поля type переменной packedData (с автоматическим смещением в младшие биты, если это требуется), которое присваивается переменной n.

Битовые поля можно использовать в обычных выражениях и автоматически преобразовывать их в целые значения. Например, оператор

```
i = packedData.index / 5 + 1;
```

является вполне допустимым, как и оператор

```
if (packedData.f2)
```

```
...
```

Проверка, установлен ли флаг f2. Мы не можем точно сказать, как заполняются битовые поля, — слева направо или справа налево. Если они заполняются справа налево, то f1 будет находиться в позиции младшего бита, f2 — в битовой позиции непосредственно слева от f1, и т.д. Это не представляет проблемы, если вы не работаете с данными, которые созданы другой программой или на другой машине.

Вы можете включать обычные типы данных в структуру, содержащую битовые поля. Например, если нужно определить структуру, содержащую типы int, char и два 1-битовых флага, можно использовать следующее определение.

```
struct table_entry
{
 int count;
 char c;
 unsigned int f1:1;
 unsigned int f2:1;
};
```

Битовые поля упаковываются в *блоки (unit)* в соответствии с порядком их следования в определении структуры, причем размер блока определяется реализацией и, скорее всего, является словом. Компилятор Objective-C не изменяет определения битовых полей, пытаясь уменьшить размер используемого пространства.

Можно задавать битовое поле без имени, чтобы пропускать биты внутри слова. Ниже определяется структура x\_entry, которая содержит 4-битовое поле с именем type и 9-битовое поле с именем count.

```
struct x_entry
{
 unsigned int type:4;
 unsigned int :3;
 unsigned int count:9;
};
```

Поле без имени указывает, что поле type отделяется от поля count 3 битами.

Особым случаем является использование неименованного поля длиной 0. Его можно использовать, чтобы принудительно выровнять следующее поле в структуре с началом границы блока.

## Не забывайте об объектно-ориентированном программировании!

Теперь вы знаете, как определить структуру для хранения даты, и умеете писать процедуры для работы со структурой даты. А как быть с объектно-ориентированным программированием? Ведь вместо этого можно создать класс с именем Date и затем разработать методы для работы с объектами типа Date. Конечно,

такой подход будет лучше. Несомненно, при работе с большим числом дат в программах определение класса и методов для работы с датами является более подходящим. Для таких целей Foundation framework содержит пару классов с именами `NSDate` и `NSCalendarDate`. Реализацию класса `Date` для работы с датами как с объектами, а не структурами мы оставляем вам в качестве упражнения.

## Указатели

*Указатели (Pointer)* позволяют эффективно представлять сложные структуры данных, изменять значения, передаваемые в виде аргументов функциям и методам, а также проще и эффективнее работать с массивами. В конце этой главы мы расскажем, насколько они важны для реализации объектов в языке Objective-C.

Мы ввели понятие объекта в главе 8, когда описывали классы `Point` и `Rectangle`, и упомянули, что можно иметь несколько ссылок на один объект.

Чтобы понять, как действуют указатели, вы должны сначала ознакомиться с понятием *косвенного обращения (indirection)*. Мы часто встречаемся с этим понятием в повседневной жизни. Предположим, мне нужно купить новый картридж с тонером для моего принтера. В компании, где я работаю, все приобретения выполняются через отдел снабжения, поэтому я звоню сотруднику этого отдела и прошу заказать для меня новый картридж. Он звонит в местный магазин запчастей, чтобы заказать картридж. Для получения картриджа я применяю косвенный подход вместо прямого заказа картриджа в магазине.

Понятие косвенного подхода можно применить и к указателям в Objective-C. Указатель – это средство косвенного доступа к значению определенного элемента данных. Рассмотрим, как действуют указатели. Определим переменную с именем `count`

```
int count = 10;
```

С помощью следующего объявления мы можем определить другую переменную с именем `intPtr` для косвенного доступа к значению переменной `count`.

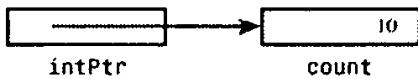
```
int *intPtr;
```

Звездочка показывает, что переменная `intPtr` является указателем на тип `int`. Это означает, что программа будет использовать `intPtr` для косвенного доступа к значению одной или нескольких целых переменных.

Мы уже использовали в предыдущих программах оператор `&` при обращении к `scanf`. Этот унарный оператор, который называют *адресным (address)* оператором, создает указатель на переменную в Objective-C. Например, если `x` – переменная определенного типа, то выражение `&x` является указателем на эту переменную. Если нужно, вы можете присвоить выражение `&x` любой переменной-указателю, объявленной как указатель на тип, который имеет переменная `x`. Оператор

```
intPtr = &count;
```

задает косвенную связь между `intPtr` и `count`. Адресный оператор `&` присваивает переменной `intPtr` не значение переменной `count`, а указатель на переменную `count`. На рис. 13.1 показана связь между `intPtr` и `count`. Линия со стрелкой показывает, что `intPtr` содержит не само значение `count`, а указатель на переменную `count`.



**Рис. 13.1.** Указатель на переменную целого типа

Чтобы указать содержимое `count` с помощью переменной-указателя `intPtr`, используется оператор косвенного обращения «звездочка» (\*). Если переменная `x` была определена с типом `int`, то оператор

`x = *intPtr;`

присвоит переменной `x` значение, которое указывается путем косвенного обращения через `intPtr`. Поскольку выше мы задали, что `intPtr` указывает на переменную `count`, результатом этого оператора будет присваивание переменной `x` значения, содержащегося в переменной `count` (10).

В программе 13.8 показано использование двух основных операторов для указателей: адресного оператора (`&`) и оператора косвенного обращения (\*).

### Программа 13.8

```

// Программа, показывающая использование указателей

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 int count = 10, x;
 int *intPtr;

 intPtr = &count;
 x = *intPtr;

 NSLog(@"%@", @"count = %i, x = %i", count, x);

 [pool drain];
 return 0;
}

```

### Вывод программы 13.8

`count = 10, x = 10`

Переменные `count` и `x` объявляются обычным образом как целые переменные. В следующей строке переменная `intPtr` к объявлению является как «указатель на тип `int`». Эти две строки можно было бы объединить в одну:

```
int count = 10, x, *intPtr;
```

После этого к переменной `count` применяется адресный оператор (`&`), результатом которого является создание указателя на эту переменную, который затем присваивается переменной `intPtr`.

Выполнение строки

```
x = *intPtr;
```

происходит следующим образом. Оператор косвенного обращения (`*`) сообщает системе Objective-C, что переменная `intPtr` содержит указатель на другой элемент данных. Этот указатель используется для доступа к нужному элементу данных, тип которого задан при объявлении переменной-указателя. При объявлении переменной мы сообщили компилятору, что переменная `intPtr` указывает на целые элементы данных, поэтому компилятор «понимает», что значение, указываемое выражением `*intPtr`, является целым. Кроме того, в предыдущей строке мы задали, что `intPtr` указывает на целую переменную `count`, поэтому это выражение представляет косвенный доступ к значению `count`.

В программе 13.9 иллюстрируются некоторые интересные свойства переменных-указателей. В ней используется указатель на символ.

### Программа 13.9

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

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 char c = "Q";
 char *charPtr = &c;

 NSLog(@"%@", c, *charPtr);

 c = '/';
 NSLog(@"%@", c, *charPtr);

 *charPtr = '(';
 NSLog(@"%@", c, *charPtr);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.9

```
Q Q
//
()
```

Символьная переменная с определяется и инициализируется со значением 'Q'. В следующей строке этой программы определяется переменная charPtr как «указатель на тип char». Это означает, что любое значение, сохраняемое внутри этой переменной, следует интерпретировать как косвенное обращение (указатель) к символу. Этой переменной можно присвоить начальное значение обычным образом. В этой программе переменной charPtr присваивается указатель на переменную c, и для этого к переменной с применяется адресный оператор &. (Эта инициализация приведет к появлению ошибки компилятора, если определить с после этого оператора, поскольку любая переменная должна быть объявлена раньше ссылки на эту переменную в любом выражении.)

Объявление переменной charPtr и присваивание ее начального значения можно было бы представить в двух отдельных строках

```
char *charPtr;
charPtr = &c;
```

(Но не

```
char *charPtr;
*charPtr = &c;
```

как можно было бы предположить из объявления в одной строке.)

Значение указателя в Objective-C не существует, пока мы не зададим, на какой элемент данных он указывает.

В строке первого вызова NSLog выводится содержимое переменной с и содержимое этой переменной, указанное с помощью charPtr. Поскольку мы задали, что charPtr указывает на переменную с, выводится и содержимое с, что подтверждается первой строкой вывода программы.

В следующей строке программы символьной переменной с присваивается символ '/'. Поскольку charPtr по-прежнему указывает на с, в следующем вызове NSLog для значения \*charPtr на терминал выводится новое значение переменной с. Это важно. Если значение charPtr не изменяется, то выражение \*charPtr всегда означает доступ к значению с. Таким образом, если изменяется значение переменной с, то изменяется и значение выражения \*charPtr.

Мы говорили, что если значение charPtr не изменено, то выражение \*charPtr всегда является ссылкой на значение с. Поэтому в выражении

```
*charPtr = '(';
```

переменной с присваивается символ «левая круглая скобка». Более формально символ '(' присваивается переменной, на которую указывает charPtr. Мы знаем, что это переменная с, поскольку в начале программы мы присвоили charPtr указатель на с.

Эти концепции являются ключом к пониманию действия указателей. Продумайте их снова, если что-то осталось неясным.

## Указатели и структуры

Мы определил указатель для базовых типов данных, таких как `int` или `char`. Можно определить указатель и на структуру. Выше в этой главе мы определили структуру `date`.

```
struct date
{
 int month;
 int day;
 int year;
};
```

Мы определили переменные типа `struct date`, например,

```
struct date todaysDate;
```

Мы можем также определить переменную-указатель на переменную типа `struct date`:

```
struct date *datePtr;
```

Переменная `datePtr` может, например, указывать на переменную `todaysDate` с помощью оператора присваивания

```
datePtr = &todaysDate;
```

После этого присваивания можно осуществлять косвенный доступ к любому из компонентов структуры `date`:

```
(*datePtr).day = 21;
```

С помощью указателя `datePtr` в этом операторе задается значение дня структуры `date`, равное 21. Здесь нужны круглые скобки, поскольку оператор «точка» для компонента структуры имеет более высокий приоритет, чем оператор косвенного обращения `*`.

Чтобы проверить значение месяца (`month`), хранящееся в структуре `date`, на которую указывает `datePtr`, можно использовать оператор

```
if ((*datePtr).month == 12)
 ...
```

Указатели на структуры используются настолько часто, что для них в языке имеется специальный оператор. Оператор указателя структуры `->` (знаки «минус» и «больше») позволяет записывать выражения

```
(*x).y
```

в более понятном виде:

```
x->y
```

Таким образом, предыдущий оператор `if` можно записать в виде

```
if (datePtr->month == 12)
 ...
```

Перепишем программу 13.6, где впервые показали использование структур, с применением указателей структур (программа 13.10).

### Программа 13.10

```
// Программа, показывающая использование указателей структур
#import <Foundation/Foundation.h>

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

 struct date
 {
 int month;
 int day;
 int year;
 };

 struct date today, *datePtr;

 datePtr = &today;
 datePtr->month = 9;
 datePtr->day = 25;
 datePtr->year = 2009;

 NSLog (@"Today's date is %i/%i/.2i.",
 datePtr->month, datePtr->day, datePtr->year % 100);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.10

Today's date is 9/25/09. (Текущая дата – 25.9.09)

## Указатели, методы и функции

Вы можете передавать указатель как аргумент методу или функции. Метод или функция может возвращать результат в виде указателя. Кстати, именно это происходит с методами alloc и init: они возвращают указатели. Более подробно мы обсудим этот вопрос в конце главы. А теперь рассмотрим программу 13.11.

### Программа 13.11

```
// Указатели как аргументы, передаваемые функциям
#import <Foundation/Foundation.h>

void exchange (int *pint1, int *pint2)
```

```
{
int temp;

 temp = *pint1;
 *pint1 = *pint2;
 *pint2 = temp;
}

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 void exchange (int *pint1, int *pint2);
 int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

 NSLog (@"i1 = %i, i2 = %i", i1, i2);

 exchange (p1, p2);
 NSLog (@"i1 = %i, i2 = %i", i1, i2);

 exchange (&i1, &i2);
 NSLog (@"i1 = %i, i2 = %i", i1, i2);

 [pool drain];
 return 0;
}
```

### Вывод программы 13.11

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

Функция `exchange` выполняет обмен значений двух целых переменных, на которые указывают два ее аргумента. Внутри функции локальная целая переменная `temp` сохраняет одно из целых значений во время выполнения обмена. Ей присваивается целое значение, на которое указывает `pint1`. Затем целое значение, на которое указывает `pint2`, копируется в целую переменную, на которую указывает `pint1`. После этого значение `temp` сохраняется в целой переменной, на которую указывает `pint2`, и обмен значениями завершается.

В процедуре `main` определяются целые переменные `i1` и `i2` со значениями `-5` и `66` соответственно. Затем определяются два указателя на тип `int` (`p1` и `p2`), которые указывают соответственно на `i1` и `i2`. Программа выводит значения `i1` и `i2` и вызывает функцию `exchange`, передавая в качестве аргументов эти указатели (`p1` и `p2`). Функция `exchange` выполняет обмен значения, содержащегося в целой переменной, на которую указывает `p1`, со значением, содержащимся в целой переменной, на которую указывает `p2`. Поскольку `p1` указывает на `i1` и `p2` на `i2`, функция `exchange` обменивает местами значения `i1` и `i2`. Вывод с помощью второго вызова `NSLog` показывает, что обмен выполняется правильно.

Второй вызов `exchange` выглядит несколько интересней. На этот раз аргументы, передаваемые функции, являются указателями на `i1` и `i2`, которые создаются непосредственно при обращении в результате применения к этим переменным адресного оператора `&`. Поскольку выражение `&i1` представляет указатель на целую переменную `i1`, это согласуется с типом первого аргумента для функции (указатель на целое значение). То же самое относится ко второму аргументу. Вывод этой программы показывает, что функция `exchange` правильно выполнила свою работу и обменяла значения `i1` и `i2`.

Внимательно изучите программу 13.11. Она показывает ключевые концепции работы с указателями в Objective-C.

## Указатели и массивы

Если имеется массив с именем `values`, содержащий 100 целых элементов, то с помощью следующей строки можно определить указатель с именем `valuesPtr`, который можно использовать для доступа к целым элементам, содержащимся в этом массиве.

```
int *valuesPtr;
```

Определяя указатель, который будет использоваться для указания элементов массива, мы не определяем его как «указатель на массив». Он определяется как указатель на тип элементов, содержащихся в массиве.

Если имеется массив `fracts` с объектами класса `Fraction`, то с помощью следующего оператора можно определить указатель на элементы массива `fracts`:

```
Fraction **fractsPtr;
```

Такое же объявление используется для определения любого объекта класса `Fraction`.

Чтобы `valuesPtr` указывал на первый элемент массива `values`, достаточно написать строку

```
valuesPtr = values;
```

В данном случае адресный оператор `&` не используется, поскольку компилятор Objective-C интерпретирует появление имени массива без индекса как указатель на первый элемент этого массива. Таким образом, мы получаем указатель на первый элемент массива `values`.

Эквивалентный способ создать указатель на начало массива `values` — применить адресный оператор `&` к первому элементу этого массива:

```
valuesPtr = &values[0];
```

Чтобы вывести содержащийся в массиве `fracts` объект класса `Fraction`, на который указывает `fractsPtr`, нужно написать оператор

```
[*fractsPtr print];
```

Реальные возможности применения указателей к массивам начинают действовать при необходимости перебора элементов массива. Если указатель `valuesPtr` определен, как описано выше, и указывает на первый элемент массива `values`, то для доступа к первому элементу массива `values` (то есть `values[0]`) можно использовать выражение

`*valuesPtr`

Для доступа к `values[3]` с помощью переменной `valuesPtr` можно добавить 3 к `valuesPtr` и затем применить оператор косвенного доступа.

`*(valuesPtr + 3)`

Таким образом, доступ к значению, содержащемуся в `values[i]`, дает выражение

`*(valuesPtr + i)`

Например, чтобы присвоить элементу `values[10]` значение 27, можно написать следующее выражение

`values[10] = 27;`

Или, используя `valuesPtr`, можно написать

`*(valuesPtr + 10) = 27;`

Чтобы `valuesPtr` указывал на второй элемент массива `values`, нужно применить адресный оператор `&` к `values[1]` и присвоить результат переменной `valuesPtr`:

`valuesPtr = &values[1];`

Если `valuesPtr` указывает на `values[0]`, и нужно сделать так, чтобы он указывал на `values[1]`, достаточно добавить 1 к значению `valuesPtr`:

`valuesPtr += 1;`

Это вполне допустимое выражение в Objective-C, его можно использовать для указателей на любой тип данных.

В общем случае, если `a` — массив элементов типа `x`, `px` — указатель на `x`, `i` и `n` — целые константы, то в выражении

`px = a;`

`px` указывает на первый элемент `a`, и выражение

`*(px + i)`

дает доступ к значению, содержащемуся в `a[i]`. Кроме того, в выражении

`px += n;`

`px` указывает на `n` следующих элементов в массиве, независимо от их типа данных.

Предположим, что `fractsPtr` указывает на дробь (`fraction`), содержащуюся в массиве дробей. Ее нужно сложить с дробью, содержащейся в следующем эле-

менте этого массива, и присвоить результат объекту `result` класса `Fraction`. Для этого можно написать:

```
result = [*fractsPtr add: *(fractsPtr + 1)];
```

Операторы приращения (`++`) и уменьшения (`--`) особенно удобны при работе с указателями. Применение к указателю оператора (`++`) дает такой же результат, как прибавление к указателю 1, а оператор уменьшения (`--`) действует так же, как вычитание 1 из указателя. Таким образом, если `textPtr` определен как указатель на тип `char` и указывает на начало массива элементов типа `char` с именем `text`, то в результате оператора

```
++textPtr;
```

`textPtr` будет указывать на следующий символ в массиве `text`, то есть на `text[1]`. Аналогичным образом, в результате оператора

```
--textPtr;
```

`textPtr` будет указывать на предыдущий символ в массиве `text` (если, конечно, `textPtr` не указывал на начало массива `text` перед выполнением этого оператора).

В Objective-C вполне допустимо сравнение двух переменных-указателей. Это особенно полезно при сравнении двух указателей на элементы одного массива. Так можно убедиться, что указатель `valuesPtr` не указывает дальше конца массива, содержащего 100 элементов. Для этого нужно сравнить этот указатель с указателем на последний элемент массива. Например, выражение

```
valuesPtr > &values[99]
```

будет иметь значение `TRUE` (ненулевое значение), если `valuesPtr` указывает дальше последнего элемента массива `values`, и будет иметь значение `FALSE` (нулевое значение) в противном случае. В соответствии с предыдущим описанием мы можем заменить это выражение на его эквивалент

```
valuesPtr > values + 99
```

`values` без индекса — это указатель на начало массива `values`. (Эквивалентно `&values[0]`.)

В программе 13.12 показано использование указателей на массивы. Функция `arraySum` вычисляет сумму целых элементов, содержащихся в массиве.

### Программа 13.12

```
// Функция для вычисления суммы элементов, содержащихся в массиве целого типа

#import <Foundation/Foundation.h>

int arraySum (int array[], int n)
{
 int sum = 0, *ptr;
 int *arrayEnd = array + n;

 for (ptr = array; ptr < arrayEnd; ++ptr)
```

```
sum += *ptr;

return (sum);
}

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 int arraySum (int array[], int n);
 int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

 NSLog (@"The sum is %i", arraySum (values, 10));
 [pool drain];
 return 0;
}
```

### Вывод программы 13.12

The sum is 21 (Сумма равна 21)

Внутри функции `arraySum` определен указатель `arrayEnd` на тип `int`, который указывает «ячейку» непосредственно вслед за последним элементом массива. Затем для последовательного перебора элементов массива используется цикл `for`. При входе в цикл `ptr` указывает на начало массива. На каждом шаге цикла к сумме прибавляется элемент массива, на который указывает `ptr`. Затем в цикле `for` значение `ptr` наращивается, чтобы указывать на следующий элемент массива. Когда `ptr` указывает «ячейку» после конца массива, происходит выход из цикла `for`, и значение `sum` возвращается вызывающей процедуре.

### Это массив или это указатель?

Чтобы передать массив функции, нужно просто передать имя этого массива, как при вызове функции `arraySum`. Но мы говорили, что для создания указателя на массив достаточно задать имя этого массива. Из этого следует, что при вызове функции `arraySum` функции передается указатель на массив `values`. Именно это и происходит, поэтому мы можем изменять элементы массива внутри функции.

Но если функции передается указатель на массив, то почему формальный параметр внутри функции не объявлен как указатель? Иначе говоря, почему при объявлении `array` в функции `arraySum` не используется следующее объявление:

```
int *array;
```

Не следует ли все обращения к массиву внутри функции выполнять с помощью переменных-указателей?

Чтобы ответить на эти вопросы, мы должны сначала вернуться к тому, что уже говорили об указателях и массивах. Если `valuesPtr` указывает на тип элемента, содержащегося в массиве с именем `values`, то выражение `*(valuesPtr + i)` эквивалентно выражению `values[i]` в предположении, что `valuesPtr` сначала указывал на

начало массива `values`. Из этого следует, что мы можем использовать выражение `*(values + i)` для обращения к  $i$ -му элементу массива `values`. В общем случае, если `x` – массив любого типа, то выражение `x[i]` всегда можно записать в эквивалентной форме `*(x + i)`.

Как видите, указатели и массивы тесно связаны друг с другом, и поэтому внутри функции `arraySum` можно объявить массив как «массив целых значений (типа `int`)» или как «указатель на тип `int`».

Если использовать индексы для доступа к элементам массива, то соответствующий формальный параметр нужно объявить как массив. Если использовать аргумент как указатель на массив, то его нужно объявить как указатель.

### Указатели на символьные строки

Чаще всего указатель на массив используется как указатель на символьную строку. Чтобы показать, насколько просты в работе указатели на символьные строки, напишем функцию с именем `copyString` для копирования одной строки в другую. Используя обычные методы написания такой функции с помощью индексирования массивов, можно составить следующий код:

```
void copyString (char to[], char from[])
{
 int i;

 for (i = 0; from[i] != '\0'; ++i)
 to[i] = from[i];

 to[i] = '\0';
}
```

Выход из цикла `for` происходит в тот момент, когда обнаруживается нуль-символ, что и делает последний оператор в этой функции.

Если написать функцию `copyString` с использованием указателей, то индексная переменная `i` не нужна. Версия с указателями показана в программе 13.13.

### Программа 13.13

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

void copyString (char *to, char *from)
{
 for (; *from != '\0'; ++from, ++to)
 *to = *from;

 *to = '\0';
}

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 void copyString (char *to, char *from);
```

```
char string1[] = "Копируемая строка"2;
char string2[50];

copyString (string2, string1);
NSLog (@"%s", string2);

copyString (string2, "Строка-константа");
NSLog (@"%s", string2);

[pool drain];
return 0;
}
```

### Вывод программы 13.13

Копируемая строка  
Строка-константа

В функции `copyString` определены два формальных параметра (`to` и `from`) как указатели на символы, а не на массивы символов, как в предыдущей версии `copyString`. Это показывает, каким образом эти две переменные будут использоваться в функции.

Затем происходит вход в цикл `for` (без начальных условий) для копирования строки, которую указывает параметр `from` (откуда) в строку, которую указывает параметр `to` (куда). На каждом шаге цикла выполняется увеличение указателей `from` и `to` на 1. В результате указатель `from` указывает на следующий символ для копирования из исходной строки, и указатель `to` указывает место, в котором будет сохранен этот символ. Когда указатель `from` указывает на нуль-символ, происходит выход из цикла `for`. Затем функция помещает нуль-символ в конец скопированной строки.

В процедуре `main` функция `copyString` вызывается дважды. В первый раз в строку `string2` копируется содержимое строки `string1`, во второй раз в строку `string2` копируется содержимое константной символьной строки.

---

<sup>2</sup> «Копируемая строка» и «Строка-константа» – это не строчные объекты, а символьные строки в стиле C, С-строки (поскольку перед строкой нет символа @). Эти два типа не являются взаимозаменяемыми. Если функция должна получать в качестве аргумента массив элементов типа `char`, то ей можно передавать массив типа `char` или литеральную символьную строку в стиле C, но не объект символьной строки.

## Константные символьные строки и указатели

Из

```
copyString (string2, "Строка-константа");
```

следует, что если константная символьная строка передается как аргумент функции, эта символьная строка фактически передается указателю. Это верно не только в данном случае. Обобщая, можно сказать, что если константная символьная строка используется в Objective-C, то на эту символьную строку создается указатель.

Это может вызвать некоторую путаницу, но, как мы отмечали в главе 4, константные символьные строки называются строками в стиле С (С-строками). Они не являются объектами. Объект константной символьной строки создается в том случае, если перед строкой поставлен знак @, например, @"This is okay", и мы не можем использовать один вид вместо другого.

Таким образом, если переменная `textPtr` объявлена как указатель на символ,

```
char *textPtr;
```

оператор

```
textPtr = "Символьная строка";
```

присваивает переменной `textPtr` указатель на константную символьную строку "Символьная строка". Будьте внимательны, чтобы отличать указатели на символьные строки от символьных массивов, поскольку приведенное выше присваивание не подходит для символьного массива. Например, если определить `text` как массив символов с помощью следующего оператора

```
char text[80];
```

то мы не сможем написать такой оператор, как

```
text = "Это неправильно";
```

Единственный случай, когда Objective-C допускает такой тип присваивания для символьного массива – инициализация массива:

```
char text[80] = "Это правильно";
```

При такой инициализации массива `text` не происходит создание указателя на символьную строку "Это правильно" внутри `text`. Вместо этого происходит запись самих символов с конечным нуль-символом в соответствующие элементы массива `text`.

Если `text` – это указатель на тип `char`, то в случае инициализации `text` с помощью оператора

```
char *text = "Это правильно";
```

ему будет присвоен указатель на символьную строку "Это правильно".

### Снова об операторах приращения (++) и уменьшения (--)

До настоящего момента оператор приращения или уменьшения был единственным оператором в выражении. Если мы пишем выражение `++x`, это означает,

что значение *x* увеличивается на 1. И, как вы уже видели, если *x* – указатель на массив, эта операция делает *x* указателем на следующий элемент массива.

Операторы приращения и уменьшения можно использовать в выражениях, где присутствуют другие операторы. В таких случаях важно знать, как они действуют.

Операторы приращения (*increment*) и уменьшения (*decrement*) всегда помещаются перед соответствующими наращиваемыми и уменьшаемыми переменными. Например, для приращения переменной *i* достаточно написать:

`++i;`

Оператор приращения можно также поместить после переменной:

`i++;`

Оба выражения допустимы, и оба дают одинаковый результат: увеличение на 1 значения *i*. В первом случае, когда `++` помещается перед операндом, операция увеличения называется *предувеличением* (*pre-increment*). Во втором случае, когда `++` помещается после операнда, операция увсличения называется *постуувеличением*.

То же самое относится к оператору уменьшения. Поэтому оператор

`--i;`

выполняет предуменьшение *i*, а оператор

`i--;`

выполняет постуменьшение *i*. Оба дают одинаковый результат: вычитание 1 из значения *i*.

Если операторы приращения и уменьшения используются в более сложных выражениях, то отличия между пред- и пост-операторами могут быть существенными.

Предположим, что у нас есть две целые переменные: *i* и *j*. Если присвоить *i* значение 0 и затем написать оператор

`j = ++i;`

то переменной *j* будет присвоено значение 1, а не 0. В случае оператора предуувеличения переменная наращивается до того, как ее значение используется в выражении. В приведенном выше выражении значение *i* сначала увеличивается с 0 до 1, а затем ее значение присваивается переменной *j*, как в следующих двух операторах.

`++i;`

`j = i;`

Если использовать оператор постуувеличения

`j = i++;`

значение *i* будут увеличено после того, как она будет присвоена переменной *j*. Если *i* было присвоено значение 0 перед выполнением оператора постуувелич-

ния, то переменной *j* будет присвоено значение 0, и затем значение *i* будет увеличено на 1, как в следующих операторах.

```
j = i;
++i;
```

Еще один пример. Если *i* равна 1, то в результате оператора

```
x = a[--i];
```

переменной *x* будет присвоено значение *a[0]*, так как переменная *i* уменьшается до того, как ее значение используется для индексации *a*. Оператор

```
x = a[i--];
```

присваивает *x* значение *a[1]*, поскольку *i* уменьшается после того, как ее значение используется для индексации *a*.

Третий пример отличий между пред- и пост-операторами приращения и уменьшения. При вызове функции

```
NSLog(@"%@", ++i);
```

происходит увеличение *i*, после чего ее значение передается функции *NSLog*, а при вызове

```
NSLog(@"%@", i++);
```

увеличение *i* происходит после того, как ее значение передано этой функции. Таким образом, если значение *i* было равно 100, при первом вызове *NSLog* на терминал будет выведено 101, а при втором вызове *NSLog* – 100. В обоих случаях значение *i* будет равно 101 после выполнения оператора.

И последний пример. Если *textPtr* – указатель на тип *char*, то в выражении

```
*(++textPtr)
```

*textPtr* сначала увеличивается на 1, а затем извлекается символ, на который указывает *textPtr*. Однако в выражении

```
*(textPtr++)
```

сначала извлекается символ, на который указывает *textPtr*, а затем увеличивается его значение. В обоих случаях круглые скобки не обязательны, поскольку операторы *\** и *++* имеют одинаковый приоритет, но применяются справа налево.

Вернемся к функции *copyString* из программы 13.13 и перепишем ее для включения операций приращения непосредственно в оператор присваивания.

Поскольку указатели *to* и *from* наращиваются каждый раз после выполнения оператора присваивания внутри цикла *for*, их следует включить в оператор присваивания как операции пост-увеличения. Модифицированный цикл *for* из программы 13.13 принимает вид

```
for (; *from != '\0';)
 *to++ = *from++;
```

Выполнение оператора присваивания внутри цикла происходит следующим образом. Считывается символ, на который указывает *from*, затем происходит наращивание *from*, чтобы он указывал следующий символ в исходной строке. Соответствующий символ будет сохранен в позиции, на которую указывает *to*; затем происходит наращивание *to* так, чтобы он указывал следующую позицию в конечной строке.

Приведенный цикл не слишком пригоден, поскольку не имеет начального выражения и не содержит выражения цикла. Для этой логики больше подходит цикл *while*. Он показан в программе 13.14, где представлена новая версия функции *copyString*. В этом цикле *while* учитывается, что нуль-символ равен значению 0.

#### Программа 13.14

```
// Функция для копирования одной строки в другую
// с помощью указателей, версия 2

#import <Foundation/Foundation.h>
void copyString (char *to, char *from)
{
 while (*from)
 *to++ = *from++;
 *to = "\0";
}

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 void copyString (char *to, char *from);
 char string1[] = "Копируемая строка";
 char string2[50];

 copyString (string2, string1);
 NSLog (@"%s", string2);

 copyString (string2, "Строка-константа");
 NSLog (@"%s", string2);

 [pool drain];
 return 0;
}
```

#### Вывод программы 13.14

Копируемая строка  
Строка-константа

## Операции с указателями

Мы можем добавлять или вычитать целые значения из указателей. Два указателя можно сравнивать. Помимо этих операций, два указателя одного типа можно вычесть. Результатом вычитания двух указателей в Objective-C является число элементов, содержащихся между этими двумя указателями. Так, если *a* указывает на массив элементов какого-либо типа, а *b* указывает на другой элемент с более высоким номером в том же массиве, то выражение *b - a* представляет число элементов между этими двумя указателями. Например, если *p* указывает на какой-либо элемент в массиве *x*, то оператор

*p = p - x;*

присваивает переменной *p* (в предположении, что это целая переменная) индекс элемента внутри массива *x*, на который указывает *p*. Следовательно, если *p* указывает на 100-й элемент в *x*

*p = &x[99];*

то значение *p* после вычитания будет равно 99.

### Указатели на функции

Для полноты изложения введем чуть более сложное понятие указателя на функцию. При работе с указателями на функции компилятору Objective-C требуется знать не только то, что указатель указывает на функцию, но и тип значения, возвращаемого этой функцией, число и типы ее аргументов. Чтобы объявить переменную *fnPtr* как указатель на функцию, которая возвращает значение типа *int* и не требует никаких аргументов, нужно написать строку

*int (\*fnPtr) (void);*

Круглые скобки вокруг *\*fnPtr* обязательны; в противном случае компилятор Objective-C будет интерпретировать этот оператор как объявление функции с именем *fnPtr*, которая возвращает указатель на тип *int* (оператор вызова функции () имеет более высокий приоритет, чем оператор косвенного обращения через указатель \*).

Чтобы задать, что указатель указывает определенную функцию, достаточно присвоить ему имя этой функции. Например, если *lookup* – это функция, которая возвращает значение типа *int* и не требует никаких аргументов, то оператор

*fnPtr = lookup;*

сохраняет указатель на эту функцию в переменной-указателе функции *fnPtr*. Написание имени функции без последующих круглых скобок интерпретируется как имя массива без индекса. Компилятор Objective-C автоматически создает указатель на указанную функцию. Перед именем функции можно поставить амперсанд, но это не обязательно.

Если функция `lookup` не была ранее определена в программе, ее следует объявить до приведенного выше оператора, например,

```
int lookup (void);
```

Для косвенного обращения к этой функции через переменную-указатель нужно применить к этому указателю оператор вызова функций с указанием всех аргументов функции в круглых скобках. Например,

```
entry = fnPtr ();
```

вызывает функцию, на которую указывает `fnPtr`, и возвращаемое значение сохраняется в переменной `entry`.

Указатели на функции часто передаются в качестве аргументов другим функциям. В библиотеке Standard Library функция `qsort` выполняет *быструю сортировку* массива элементов данных. Эта функция принимает в качестве одного из своих аргументов указатель на функцию, которая вызывается, когда `qsort` требуется сравнение двух элементов в сортируемом массиве. Это позволяет использовать `qsort` для сортировки массивов любого типа, поскольку конкретное сравнение любых двух элементов в массиве выполняется с помощью функции пользователя, а не самой функцией `qsort`.

В Foundation framework некоторые методы принимают в качестве аргумента указатель на функцию. Например, метод `sortUsingFunction:context:` определен в классе `NSMutableArray` и вызывает указанную функцию, когда требуется сравнение двух элементов массива.

Еще один применением указателей на функции является создание *таблицы вызовов* (*dispatch table*). Мы не можем сохранять сами функции в элементах массива, но можем сохранять внутри массива указатели на функции. Это позволяет создавать таблицы, которые содержат указатели на вызываемые функции. Например, можно создать таблицу обработки различных команд, которые вводит пользователь. Каждая запись в таблице может содержать как имя команды, так и указатель на функцию, вызываемую для обработки этой конкретной команды. Когда пользователь вводит команду, ее можно найти в этой таблице и вызвать соответствующую функцию для ее обработки.

## Указатели и адреса памяти

Прежде чем закончить разговор об указателях в Objective-C, мы должны описать их реализацию. Память компьютера можно рассматривать как последовательный набор ячеек памяти. Каждая ячейка памяти компьютера имеет свой номер, называемый *адресом*. Обычно первый адрес памяти имеет номер 0. В большинстве компьютерных систем ячейка занимает 1 байт.

Компьютер использует память для хранения команд программы и хранения значений переменных, связанных с программой. Например, если мы объявим переменную с именем `count` типа `int`, то система выделит ячейки в памяти, чтобы сохранять значение `count` во время выполнения программы. Это может быть, например, адрес  $1000FF_{16}$  в памяти компьютера.

К счастью, нам не нужно думать о конкретных адресах памяти, связанных с переменными, поскольку система делает это автоматически. Однако знание того, что каждая переменная связана со своим адресом в памяти, помогает понять, как действуют указатели.

Когда мы применяем адресный оператор `&` к переменной в Objective-C, генерируемое значение — это конкретный адрес данной переменной в памяти компьютера. (Именно поэтому мы называем этот оператор адресным.) Например, оператор

```
intPtr = &count;
```

присваивает указателю `intPtr` адрес в памяти компьютера, который был выделен для переменной `count`. Так, если `count` размещена по адресу `1000FF16`, этот оператор присвоит указателю `intPtr` значение `0x1000FF`.

Применение оператора косвенного доступа `*` к переменной-указателю, как в выражении

```
*intPtr
```

означает интерпретацию значения, содержащегося в этой переменной, как адреса памяти. Значение, хранящееся по этому адресу, считывается и интерпретируется в соответствии с типом, объявленным для этой переменной. Например, если переменная `intPtr` является указателем на тип `int`, то система будет интерпретировать значение, хранящееся по адресу памяти, заданному с помощью `*intPtr`, как целое значение.

## Объединения

Одной из наиболее необычных конструкций в языке программирования Objective-C является *объединение* (*union*). Эта конструкция используется в основном для сложных программных приложений, когда требуется сохранять различные типы данных в одном и том же месте памяти. Предположим, что нужно определить одну переменную с именем `x`, которую можно было бы использовать для хранения одного символа, числа с плавающей точкой или целого числа. Для этого можно определить объединение, например, с именем `mixed`.

```
union mixed
{
 char c;
 float f;
 int i;
};
```

Объявление объединения совпадает со структурой (за исключением ключевого слова `union`). Принципиальным отличием между структурами и объединениями является способ выделения памяти. Объявление переменной типа `union mixed`, как в

```
union mixed x;
```

не означает, что `x` содержит три отдельных компонента с именами `c`, `f` или `i`. На самом деле `x` содержит один компонент, который называется `c`, `f` или `i`. Тем самым мы можем использовать переменную `x` для хранения элемента типа `char`, `float` или `int`, но не всех трех типов одновременно. Чтобы сохранить какой-либо символ в переменной `x`, используется оператор

```
x.c = 'K';
```

Чтобы сохранить в `x` значение с плавающей точкой, используется форма записи `x.f`:

```
x.f = 786.3869;
```

И, наконец, чтобы сохранить в `x` результат деления целой переменной `count` на 2, используется оператор

```
x.i = count / 2;
```

Поскольку элементы `x` типа `float`, `char` и `int` находятся в одном месте памяти, мы можем одновременно сохранять в `x` только одно из этих значений. Кроме того, значение, которое считывается из объединения, должно соответствовать тому, что было в последний раз записано в это объединение.

При определении объединения имя объединения указывать не обязательно, и переменные можно объявлять одновременно с определением этого объединения. В объединениях можно объявлять указатели с таким же синтаксисом и правилами выполнения операций, как для структур. Мы можем инициализировать переменную типа `union` следующим образом.

```
union mixed x = { '#' };
```

Первому члену `x`, то есть `c`, присваивается символ `#`. Определенный член объединения можно также инициализировать по имени следующим образом.

```
union mixed x = {.f=123.4};
```

Мы можем инициализировать автоматическую `union`-переменную типа `@@`, присвоив ей другую `union`-переменную того же типа.

С помощью объединения можно определять массивы для хранения элементов данных различного типа. Ниже задается массив с именем `table`, содержащий `kTableEntries` элементов.

```
struct
{
 char *name;
 int type;
 union
 {
 int i;
 float f;
 char c;
 } data;
} table [kTableEntries];
```

Каждый элемент этого массива содержит структуру, включающую указатель на типа `char` с именем `name`, целый компонент с именем `type` и `union`-компонент с именем `data`. Каждый элемент `data` может содержать компонент типа `int`, `float` или `char`. Целый компонент `type` позволяет следить за типом значения, сохраняемого в `data`. Мы можем присвоить `type` значение `INTEGER` (определенное соответствующим образом), если содержится значение типа `int`; значение `FLOATING`, если содержится значение типа `float`; `CHARACTER`, если содержится значение типа `char`. Эта информация позволяет узнать, как обращаться к определенному компоненту структуры `data` определенного элемента массива.

Чтобы сохранить символ '#' в элементе `table[5]` и затем записать в поле `type`, что в этом месте хранится символ, применяются два оператора

```
table[5].data.c = '#';
table[5].type = CHARACTER;
```

Во время перебора элементов `table` можно определять тип значения данных, хранящегося в каждом элементе, с помощью набора проверок. Например, в следующем цикле выводится имя и его значение из массива `table`.

```
enum symbolType { INTEGER, FLOATING, CHARACTER };
...
for (j = 0; j < kTableEntries; ++j)
{
 NSLog(@"%@", table[j].name);

 switch (table[j].type)
 {
 case INTEGER:
 NSLog(@"%@", table[j].data.i);
 break;
 case FLOATING:
 NSLog(@"%@", table[j].data.f);
 break;
 case CHARACTER:
 NSLog(@"%@", table[j].data.c);
 break;
 default:
 NSLog(@"Unknown type (%i), element %i",
 table[j].type, j);
 break;
 }
}
```

На практике такое приложение можно использовать для хранения таблицы символов, содержащей имя каждого символа, его тип и его значение (и другую информацию).

## Это не объекты!

Теперь вы знаете, как определять массивы, структуры, символьные строки и объединения и как работать с ними в программах. Помните главное: они *не являются* объектами. Мы не можем передавать им сообщения. Они не позволяют полностью использовать такие удобные возможности, как стратегию выделения памяти, которая обеспечивается в Foundation framework. Это одна из причин, по которым я советовал пропустить эту главу и вернуться к ней позже. Вы лучше подготовлены к изучению того, как использовать классы Foundation, в которых массивы и строки определяются как объекты, чем к использованию таких средств, встроенных в язык. Прибегайте к использованию типов, описанных в этой главе, только в случае реальной необходимости!

## Различные средства языка

Некоторые средства языка не совсем согласуются с материалом других глав, поэтому мы включили их в этот раздел.

### Составные литералы

*Составной литерал* (*compound literal*) представляет собой имя типа, заключенное в круглые скобки, после которого следует список инициализации. В результате создается неименованное значение указанного типа, область действия которого ограничена блоком, в котором оно создано, или глобальной областью действия, если оно определено вне блока. В последнем случае все инициализаторы должны включать только константные выражения.

Рассмотрим следующий пример.

```
(struct date) {.month = 7, .day = 2, .year = 2004}
```

Это выражение создает структуру типа `struct date` с указанными начальными значениями. Ее можно присвоить другой структуре типа `struct date`, например,

```
theDate = (struct date) {.month = 7, .day = 2, .year = 2004};
```

Ее можно передать функции или методу как аргумент типа `struct date`, например,

```
setStartDate ((struct date) {.month = 7, .day = 2, .year = 2004});
```

Кроме того, можно определять типы, отличные от структур. Например, если `IntPtr` имеет тип `int *`, оператор

```
IntPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99};
```

который может появиться в любом месте программы, задает `IntPtr`, указывающий на массив из 100 целых элементов, первые 3 элемента которого инициализируются указанным образом.

Если размер массива не задан, он определяется списком инициализации.

## Оператор *goto*

Оператор *goto* вызывает непосредственный переход в указанную точку программы. Чтобы указать это место, требуется метка. *Метка (label)* – это имя, формируемое по таким же правилам, как имена переменных; сразу после него ставится двоеточие. Метка ставится непосредственно перед оператором, на который выполняется переход, и должна присутствовать в той же функции или методе, где находится соответствующий *goto*.

Например, оператор

```
goto out_of_data;
```

вызывает переход к оператору, перед которым стоит метка *out\_of\_data*. Эта метка должна находиться где-либо в функции или методе (до или после *goto*) и может использоваться, например, как показано ниже.

```
out_of_data: NSLog(@"Unexpected end of data.");
```

```
...
```

Ленивые программисты часто злоупотребляют оператором *goto* для перехода к другим частям своего кода. Оператор *goto* нарушает нормальную последовательность программы, что затрудняет отслеживание ее выполнения. В практике надежного программирования не рекомендуется использовать операторы *goto*.

## Пустой оператор

Objective-C позволяет помещать отдельный символ «точка с запятой» в любом месте, где можно ставить обычный программный оператор. Такой оператор (его называют *пустым (null) оператором*) ничего не выполняет. Это может показаться бесполезным, но программисты часто используют его в операторах *while*, *for* и *do*. Например, следующий оператор используется для сохранения всех символов, прочитанных со *стандартного устройства ввода* (по умолчанию это терминал), в массив символов, на который указывает *text*. Ввод продолжается, пока не встретится символ новой строки (перевода строки). В этом операторе используется библиотечная процедура *getchar*, которая читает и возвращает по одному символу со стандартного устройства ввода.

```
while ((*text++ = getchar ()) != '\n')
;
```

Все операции выполняются в соответствии с условиями цикла оператора *while*. Здесь требуется пустой оператор, поскольку компилятор интерпретирует оператор, который следует за выражением цикла, как тело цикла. Без пустого оператора компилятор будет обрабатывать как тело цикла любой следующий программный оператор.

## Оператор «запятая»

На самом нижнем уровне приоритетов находится оператор «запятая». В главе 5 мы указывали, что внутри оператора *for* можно включать в любое из полей несколько выражений, отделяя их запятыми. Например, в операторе

```
for (i = 0, j = 100; i != 10; ++i, j -= 10)
...
```

до начала цикла инициализируется значение *i*, равное 0, и *j*, равное 100. После выполнения тела цикла значение *i* увеличивается на 1, а из значения *j* вычитается 10.

Поскольку все операторы в Objective-C дают какое-то значение, значением оператора «запятая» является результат правого выражения.

## Оператор sizeof

В программах никогда не следует делать какие-либо предположения о размере определенного типа данных, но иногда нужно знать эту информацию – например, при выделении динамической памяти, использовании библиотечных процедур, при записи или архивации данных в файл. В Objective-C имеется оператор с именем `sizeof`, который можно использовать для определения размера типа данных или объекта. Оператор `sizeof` возвращает размер указанного элемента в байтах. Аргументом для оператора `sizeof` может быть переменная, имя массива, имя базового типа данных, объект, имя производного типа данных или выражение. Например, написав

```
sizeof (int)
```

мы получим число байтов для сохранения целого значения. На моем MacBook Air получается значение 4 (то есть 32 бита). Если объявить *x* как массив из 100 значений типа `int`, то выражение

```
sizeof (x)
```

даст количество памяти, необходимой для сохранения 100 целых элементов массива *x*.

Если *myFract* является объектом класса `Fraction`, содержащим две переменных экземпляра типа `int` (`numerator` и `denominator` – числитель и знаменатель), то выражение

```
sizeof (myFract)
```

даст значение 4 на любом компьютере, где для представления указателей используются 4 байта. На самом деле `sizeof` дает это значение для любого объекта, потому что мы запрашиваем размер указателя на данные объекта. Чтобы получить размер реальной структуры данных для хранения экземпляра объекта класса `Fraction`, нужно написать

```
sizeof (*myFract)
```

На моем MacBook Air это дает значение 12. Эту сумму складывается из 4 байтов для `numerator`, 4 байтов для `denominator`, плюс еще 4 байта для наследуемого компонента `isa`, о котором говорится в разделе «Как это действует» в конце главы. Выражение

```
sizeof (struct data_entry)
```

дает количество памяти, необходимой для хранения одной структуры `data_entry`. Если `data` определен как массив элементов `struct data_entry`, то выражение

`sizeof (data) / sizeof (struct data_entry)`

дает число элементов, содержащихся в `data` (`data` должен быть заранее определенным массивом, а не формальным параметром или массивом по внешней ссылке). Выражение

`sizeof (data) / sizeof (data[0])`

дает тот же результат.

Используйте оператор `sizeof`, чтобы избежать вычислений или задания фиксированных размеров в программах.

## Аргументы командной строки

Довольно часто программы запрашивают на терминале пользователя ввод небольшого количества информации.

Вместо запроса такую информацию можно вводить при запуске программы. Это осуществляется с помощью *аргументов командной строки* (*command-line arguments*).

Мы уже указывали, что единственным отличительным свойством функции `main` является специальное имя; она указывает, где должно начинаться выполнение программы. На самом деле функцию `main` в начале выполнения программы вызывает система выполнения (`runtime`) так же, как мы вызываем функцию из своей собственной программы. Когда `main` заканчивает выполнение, управление передается системе `runtime`, которая знает, что ваша программа завершена.

Когда система `runtime` вызывает `main`, этой функции передаются два аргумента. Первый аргумент, который называется `argc` (сокращение от *argument count* – число аргументов), является целым значением, которое указывает число аргументов, вводимых в командной строке. Второй аргумент для `main` – это массив указателей на символьные значения с именем `argv` (сокращение от *argument vector*). В этом массиве содержатся `argc + 1` символьных указателей. Первым элементом этого массива является указатель на имя выполняемой программы или указатель на нуль-строку, если имя программы недоступно. Последующие записи этого массива указывают значения, заданные в той же строке, что и команда, инициировавшая выполнение данной программы. Последний указатель массива `argv`, `argv[argc]`, определен как пустой указатель.

Для доступа к аргументам командной строки функция `main` должна быть объявлена с двумя аргументами. Во всех программах этой книги мы использовали объявление

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

Напомним, что объявление `argv` определяет массив, который содержит элементы типа «указатель на тип `char`». Для практического использования аргументов командной строки предположим, что мы разрабатываем программу, которая ищет нужное слово в словаре и выводит его смысл. С помощью следующей команды можно использовать аргументы командной строки, чтобы

слово, смысл которого нужно определить, было указано одновременно с запуском программы.

```
lookup aerie
```

Это позволяет обойтись без запроса ввода пользователя, поскольку слово вводится из командной строки.

При запуске этой команды система автоматически передает функции main указатель на символьную строку "ærie" в argv[1]. Напомним, что argv[0] содержит указатель на имя программы (в данном случае это "lookup").

Процедура main может иметь следующий вид.

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

int main (int argc, char *argv[])
{
 struct entry dictionary[100] =
 {
 { "aardvark", "a burrowing African mammal" },
 { "abyss", "a bottomless pit" },
 { "acumen", "mentally sharp; keen" },
 { "addle", "to become confused" },
 { "aerie", "a high nest" },
 { "affix", "to append; attach" },
 { "agar", "a jelly made from seaweed" },
 { "ahoy", "a nautical call of greeting" },
 { "aigrette", "an ornamental cluster of feathers" },
 { "ajar", "partially opened" } };

 int entries = 10;
 int entryNumber;
 int lookup (struct entry dictionary [], char search[],
 int entries);

 if (argc != 2)
 {
 NSLog(@"No word typed on the command line.");
 return (1);
 }

 entryNumber = lookup (dictionary, argv[1], entries);

 if (entryNumber != -1)
 NSLog(@"%@", dictionary[entryNumber].definition);
 else
 NSLog(@"Sorry, %s is not in my dictionary.", argv[1]);

 return (0);
}
```

Процедура `main` при запуске программы проверяет, было ли введено слово после имени программы. Если не было или было введено больше одного слова, то значение `argc` не равно 2, программа выводит сообщение об ошибке и завершает работу, возвращая при выходе значение состояния 1.

Если значение `argc` равно 2, то вызывается функция `lookup` для поиска в словаре слова, на которое указывает `argv[1]`. Если это слово найдено, выводится его смысл (определение).

Аргументы командной строки всегда сохраняются как символьные строки. Например, при запуске программы `power` (возвведение в степень) с аргументами командной строки 2 и 16

```
power 2 16
```

в `argv[1]` сохраняется указатель на символьную строку "2" и в `argv[2]` сохраняется указатель на символьную строку "16". Если программа должна интерпретировать аргументы как числа (например, в случае программы `power`), их должна преобразовывать сама эта программа. Для таких преобразований в библиотеке программ содержится несколько процедур: `sscanf`, `atof`, `atoi`, `strtod` и `strtol`. В части II вы узнаете, как использовать класс `NSProcessInfo` для доступа к аргументам командной строки как к строковым объектам, а не C-строкам.

## Как это действует

Было бы упущением закончить эту главу без описания связей между некоторыми элементами. Поскольку в основе языка Objective-C лежит язык C, имеет смысл описать некоторые связи между ними. Обратите внимание на эти детали реализации, чтобы лучше понять, как все это действует. Мы не будем здесь вдаваться в подробности, а просто приведем четыре факта о связях между Objective-C и C.

### Факт 1: переменные экземпляра сохраняются в структурах

Когда мы определяем новый класс и его переменные экземпляра, эти переменные на самом деле сохраняются в структуре. На самом деле это структуры, компонентами которых являются наши переменные экземпляра. Поэтому наследуемые переменные экземпляра плюс переменные экземпляра, которые мы добавляем в свой класс, представляют одну структуру. Если мы выделяем память для нового объекта с помощью `alloc`, резервируется достаточное пространство, чтобы включить одну из этих структур.

Одним из наследуемых компонентов этой структуры (он поступает из корневого объекта) является защищенный компонент с именем `isa`, который указывает класс, которому принадлежит объект. Поскольку этот компонент является частью структуры (и тем самым частью объекта), он переносится вместе с объектом. Это позволяет системе `runtime` всегда идентифицировать класс объекта (даже если он присваивается обобщенной переменной-объекту типа `id`) по информации его компонента `isa`.

Чтобы получить непосредственный доступ к компонентам структуры объекта, можно объявить их как `@public` (см. главу 10). Если сделать это, например,

для компонентов numerator и denominator класса Fraction, то можно писать в программах такие выражения, как

```
myFract->numerator
```

для непосредственного доступа к компоненту numerator объекта myFract класса Fraction. Но мы настоятельно рекомендуем не делать этого. Как говорилось в главе 10, это противоречит основам инкапсуляции данных.

## **Факт 2: переменная-объект на самом деле является указателем**

Определяя переменную-объект класса Fraction, например,

```
Fraction *myFract;
```

мы фактически определяем переменную-указатель с именем myFract. Эта переменная определяется для указания элемента типа Fraction (это имя нашего класса). При выделении памяти для нового экземпляра типа Fraction с помощью строки

```
myFract = [Fraction alloc];
```

мы выделяем пространство в памяти для хранения нового объекта класса Fraction (то есть пространство для структуры) и затем сохраняем указатель на эту структуру, который возвращается в переменной-указателе myFract.

Присваивая один объект-переменную другому, как в

```
myFract2 = myFract1;
```

мы просто копируем указатели. В результате обе переменные будут указывать на одну структуру, хранящуюся в определенном месте памяти. Поэтому внесение изменений в один из компонентов, который указывается с помощью myFract2, вызывает изменения в той же переменной экземпляра (то есть в компоненте структуры), которую указывает myFract1.

## **Факт 3: методы и функции, а также выражения с сообщениями – это вызовы функций**

Методы – это на самом деле функции. При вызове метода мы вызываем функцию, связанную с классом получателя. Аргументы, передаваемые функции, это аргументы получателя (*self*) и метода. Поэтому все правила, касающиеся передачи аргументов функциям, возвращаемых значений, а также автоматических и статических переменных, одинаковы для функции и метода. Компилятор Objective-C создает уникальное имя для каждой функции в виде комбинации из имени класса и имени метода.

## **Факт 4: тип id – это обобщенный тип указателя**

Поскольку обращение к объектам выполняется через указатели, которые являются просто адресами памяти, мы можем легко присваивать им переменные типа id. Поэтому метод, который возвращает тип id, просто возвращает указа-

тель на некоторый объект в памяти. Мы можем затем присвоить это значение любому объекту-переменной. Поскольку объект при его перемещении сопровождается своим компонентом `isa`, его класс можно всегда идентифицировать, даже если он хранится в обобщенном объекте-переменной типа `id`.

## Упражнения

1. Напишите функцию, которая вычисляет среднее значение 10 элементов массива с плавающей точкой и возвращает результат.
2. Метод `reduce` из класса `Fraction` находит наибольший общий делитель чисителя и знаменателя (`numerator` и `denominator`) для сокращения дроби. Внесите изменения в этот метод, чтобы в нем можно было использовать функцию `gcd` из программы 13.5. Где следует поместить определение этой функции? Будет ли удобнее сделать эту функцию статической? Какой подход вы считаете более подходящим: использование функции `gcd` или включение этого кода непосредственно в метод, как мы делали это раньше? Почему?
3. Алгоритм, известный под названием «*Решето Эратосфена*», позволяет получать простые числа. Ниже приводится алгоритм для этой процедуры. Напишите программу, которая реализует этот алгоритм. Сделайте так, чтобы программа находила все простые числа до  $n = 150$ . Что вы можете сказать об этом алгоритме по сравнению с алгоритмами в этой книге для расчета простых чисел?  
Шаг 1. Определить массив  $P$  с целыми значениями. Присвоить всем элементам  $P_i$  значение 0,  $2 \leq i \leq n$ .  
Шаг 2. Присвоить  $i$  значение 2.  
Шаг 3. Если  $i > n$ , алгоритм завершается.  
Шаг 4. Если  $P_i$  равно 0,  $i$  – простое число.  
Шаг 5. Для всех положительных целых значений  $j$ , удовлетворяющих условию  $i \times j \leq n$ , присвоить  $P_{i \times j}$  значение 1.  
Шаг 6. Увеличить  $i$  на 1 и перейти к шагу 3.
4. Напишите функцию для сложения всех дробей (объектов `Fraction`), передаваемых ей в массиве, и возвращения результата в виде объекта `Fraction`.
5. Напишите определение `typedef` для структуры `struct date` с именем `Date`, чтобы в вашей программе можно было делать, например, следующие объявления.  
`Date todaysDate;`

6. Определение класса `Date` вместо структуры `date` больше согласуется с принципами объектно-ориентированного программирования. Определите такой класс с соответствующими методами-установщиками (`setter`) и получателями (`getter`). Добавьте метод `dateUpdate`, чтобы возвращать день по его аргументу.

Покажите преимущества определения `Date` в виде класса, а не в виде структуры.

Усматриваете ли вы какие-то недостатки?

7. В соответствии со следующими определениями

```
char *message = "Программировать на Objective-C интересно";
char message2[] = "Вы сказали это";
int x = 100;
```

определите, является ли допустимым каждый вызов NSLog в следующих наборах и является ли вывод одинаковым для всех вызовов из этого набора.

```
/** набор 1 **/
NSLog (@"Программировать на Objective-C интересно");
NSLog (@"%s", "Программировать на Objective-C интересно");
NSLog (@"%s", message);
```

```
/** набор 2 **/
NSLog (@"Вы сказали это");
NSLog (@"%s", message2);
NSLog (@"%s", &message2[0]);
```

```
/** набор 3 **/
NSLog (@"сказали это");
NSLog (@"%s", message2 + 4);
NSLog (@"%s", &message2[4]);
```

8. Напишите программу, которая выводит на терминал все аргументы командной строки (по одному на строку). Обратите внимание на использование кавычек для аргументов, которые содержат пробелы.
9. Какие из следующих операторов дают на выходе строку "Это проверка"? Объясните результаты.

```
NSLog (@"Это проверка");
NSLog ("Это проверка");
```

```
NSLog (@"%s", "Это проверка");
NSLog (@"%s", @"Это проверка");
```

```
NSLog ("%s", "Это проверка");
NSLog ("%s", @"Это проверка");
```

```
NSLog (@"%@", @"Это проверка");
NSLog (@"%@", "Это проверка");
```

## Глава 14

# Введение в Foundation Framework

Фреймворк (framework) – это набор классов, методов, функций и документации, логически сгруппированных для упрощения разработки программ. В системе Mac OS X имеется более 80 фреймворков для разработки приложений. Они позволяют упростить работу со структурой Mac Address Book, выполнять запись на CD, воспроизведение DVD, воспроизведение фильмов с помощью QuickTime, воспроизведение музыки и т.д.

Фреймворк, который является базой для разработки программ, называется Foundation framework. Этот фреймворк, (он является темой второй части книги) позволяет работать с базовыми объектами, такими как числа и строки, а также с коллекциями объектов, такими как массивы, словари и наборы (множества). Имеются также средства для работы с датой и временем, автоматического управления памятью, работы с базовыми файловыми системами, сохранения (или *архивации*) объектов, а также для работы с геометрическими структурами данных, такими как точки (point) и прямоугольники (rectangle).

Фреймворк Application Kit содержит обширный набор классов и методов для разработки интерактивных графических приложений. Они позволяют легко работать с текстами, меню, панелями инструментов, таблицами, документами, компоновочными буферами (pasteboard) и окнами. В Mac OS X термин *Cocoa* означает совместное использование фреймворков Foundation framework и Application Kit framework. Термин *Cocoa Touch* означает совместное использование фреймворков Foundation framework и UIKit framework. Эта тема описывается в части III настоящей книги. Многие источники информации приводятся в приложении D.

## Документация Foundation

Для справки мы приводим местонахождение заголовочных файлов (.h) Foundation. Это папка

`/System/Library/Frameworks/Foundation.framework/Headers`

**Примечание.** Эти заголовочные файлы на самом деле на самом деле привязаны к другой папке, где они хранятся, но для вас это не имеет значения.

Перейдите в эту папку на своем компьютере и ознакомьтесь с ее содержимым. Обратитесь также к документации Foundation framework, которая хранится на вашем компьютере (в подпапках папки /Developer/Documentation) и доступна на веб-сайте Apple. Большинство документации доступно в форме HTML-файлов для просмотра браузером или pdf-файлов Acrobat. В ней содержится описание всех классов Foundation и всех реализованных методов и функций.

Если вы используете Xcode для разработки своих программ, то имеете простой доступ к этой документации через окно Documentation, которое вызывается с помощью меню Help Xcode. Из этого окна можно легко выполнять поиск и осуществлять доступ к документации, которая хранится локально на вашем компьютере или доступна в Интернет. На рис. 14.1 показаны результаты поиска строки «foundation framework» в окне документации Xcode. Из панели под заголовком «Foundation Framework Reference» (Справка по Foundation Framework) вы можете легко выполнять доступ к документации по всем классам Foundation.

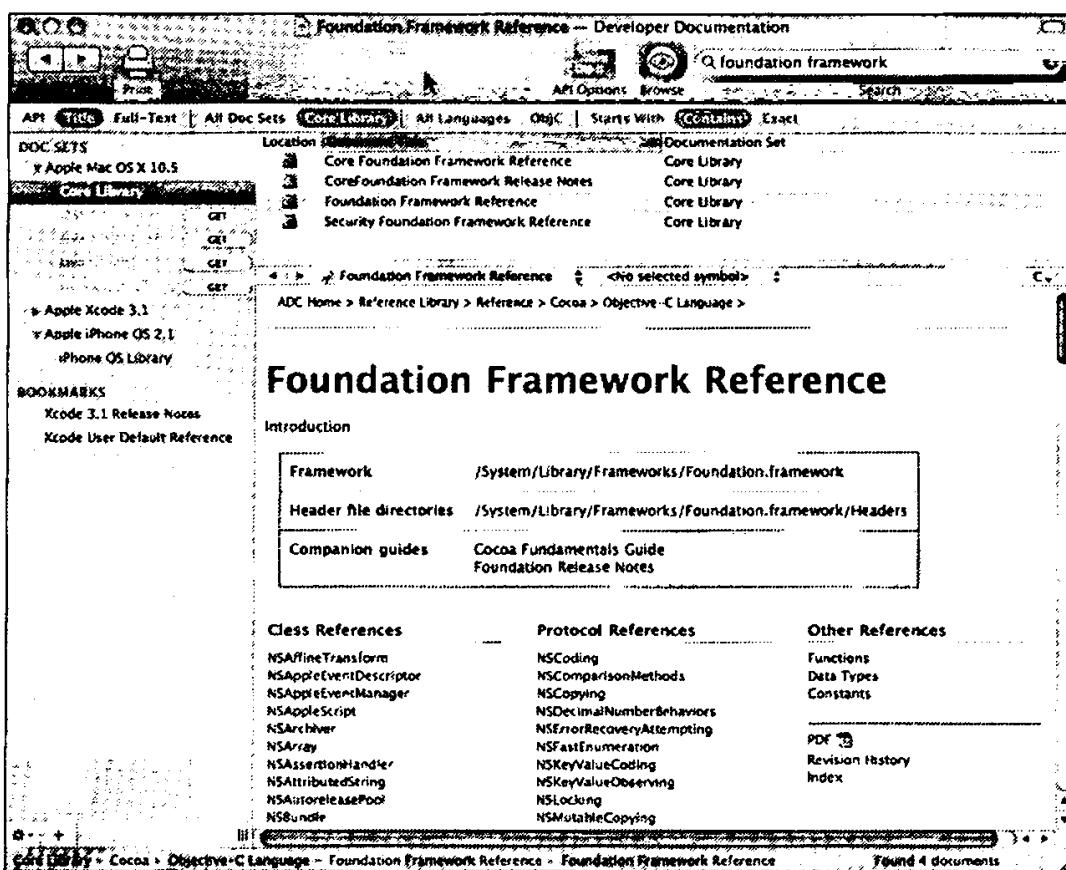


Рис. 14.1. Доступа к справочной документации Foundation из Xcode

Если вы редактируете файл в Xcode и хотите получить непосредственный доступ к документации по определенному заголовочному файлу, методу или классу, достаточно выделить этот текст в окне редактора и щелкнуть на нем правой кнопкой. В появившемся меню можно выбрать Find Selected Text in Documentation (Найти выбранный текст в документации) или Find Selected Text in API Reference (Найти выбранный текст в справке API). Xcode найдет нужную библиотеку документации и выведет результат в соответствии с вашим запросом.

Класс `NSString` – это класс Foundation, который используется для работы со строками. (Его описание см. в следующей главе.) Предположим, что вы редактируете программу, в которой используется этот класс, и вам нужно получить информацию о нем и его методах. Нужно выделить слово `NSString` в любом месте окна редактирования и щелкнуть на нем правой кнопкой. Если выбрать в появившемся меню пункт Find Selected Text in API Reference, появится окно документа (рис. 14.2).

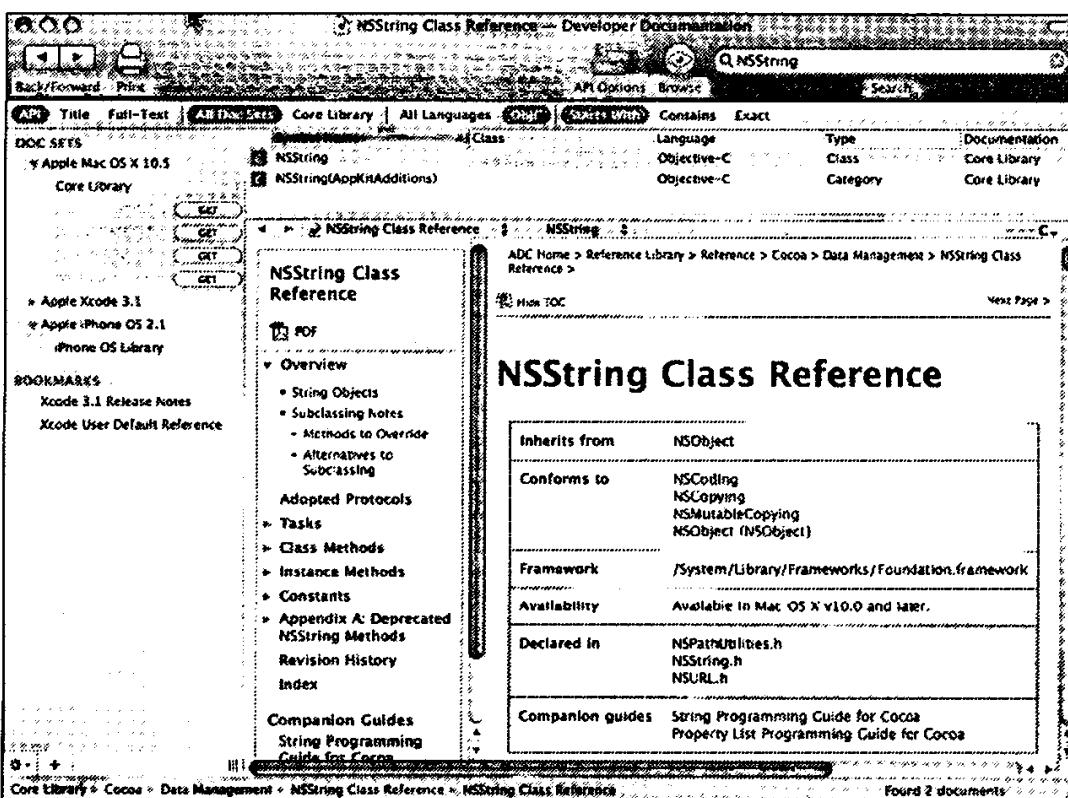


Рис. 14.2. Получение документации по классу `NSString`

Выполняя прокрутку панели под заголовком `NSString Class Reference` (Справка по классу `NSString`), вы увидите (среди прочей информации) список всех методов, которые поддерживаются этим классом. Это позволяет легко находить информацию о методах, реализуемых определенным классом, включая описание их работы и аргументов, которые они принимают.

К этой документации можно также выполнять доступ по адресу [developer.apple.com/referencelibrary](http://developer.apple.com/referencelibrary) с переходом к справочной документации Foundation (по ссылкам Cocoa, Frameworks, Foundation Framework Reference). На этом веб-сайте можно найти разнообразные документы по определенным вопросам программирования, таким как управление памятью, строки и управление файлами.

Если вы не подписаны на определенный набор документов вместе с Xcode, то онлайн-документация может быть более свежей, чем на вашем диске.

На этом заканчивается краткое введение в Foundation framework. Теперь мы переходим к изучению некоторых классов этого фреймворка и способам их включения в ваши приложения.

## Глава 15

# Числа, строки и коллекции

В этой главе описывается работа с некоторыми базовыми объектами в Foundation framework. Это числа, строки и коллекции, позволяющие работать с группами объектов в форме массивов, словарей и наборов.

Foundation framework содержит множество классов, методов и функций. В Mac OS X доступны примерно 125 заголовочных (.h) файлов. Для доступа к ним используйте следующий оператор импорта.

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

Файл Foundation.h импортирует практически все другие заголовочные файлы Foundation. Xcode автоматически вставляет этот заголовочный файл в вашу программу, как во всех примерах этой книги.

Однако наличие этого оператора может существенно увеличить время компиляции. Чтобы избежать излишних затрат времени, используйте *заранее скомпилированные* заголовочные файлы. Это файлы, заранее обработанные компилятором. По умолчанию во всех проектах Xcode используются заранее скомпилированные заголовочные файлы.

В этой главе используются заголовочные файлы для каждого объекта, чтобы показать вам, что содержится в каждом заголовочном файле.

---

**Примечание.** При желании можно продолжать работу, просто импортируя Foundation.h. Но если вы хотите импортировать конкретные файлы, показанные в каждом примере, то удалите файл *имя\_проекта\_Prefix.pch*, который автоматически включается системой XCode при создании нового проекта Foundation Tool. При удалении этого файла из проекта обязательно выберите «Delete References» (Удалить ссылки), когда появится запрос Xcode.

---

## Числовые объекты

Все числовые типы данных, с которыми мы работали до сих пор (такие, как `int`, `float` и `long`) – это базовые типы данных в Objective-C. Они не являются объектами, мы не можем передавать им сообщения. Однако время от времени нам нужно работать с ними как с объектами. Например, объект класса Foundation `NSArray` позволяет задавать массив, в котором можно сохранять значения. Эти значения должны быть объектами. В таких массивах нельзя непосредственно сохранять базовые типы данных. Вместо этого для сохранения любого из базовых числовых типов данных (включая тип данных `char`) служит класс `NSNumber`. Он позволяет создавать объекты из этих типов данных (см. программу 15.1).

### Программа 15.1

```
// Работа с числами

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSNumber.h>

#import <Foundation/NSString.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSNumber *myNumber, *floatNumber, *intNumber;
 NSInteger myInt;

 // целое значение (int)

 intNumber = [NSNumber numberWithInt: 100];
 myInt = [intNumber integerValue];
 NSLog(@"%@", (long) myInt);

 // значение удвоенной длины (long)

 myNumber = [NSNumber numberWithLong: 0xabcdef];
 NSLog(@"%@", [myNumber longValue]);

 // значение типа char

 myNumber = [NSNumber numberWithChar: 'X'];
 NSLog(@"%@", [myNumber stringValue]);

 // значение с плавающей точкой (float)

 floatNumber = [NSNumber numberWithFloat: 100.00];
 NSLog(@"%@", [floatNumber floatValue]);
```

```
// значение с двойной точностью (double)

myNumber = [NSNumber numberWithDouble: 12345e+15];
NSLog(@"%@", [myNumber doubleValue]);

// Здесь неверный доступ

 NSLog(@"%@", [myNumber integerValue]);

// Сравнение двух чисел на равенство

if ([intNumber isEqualToNumber: floatNumber] == YES)
 NSLog(@"Numbers are equal"); (Числа равны)
else
 NSLog(@"Numbers are not equal"); (Числа не равны)

// Сравнение одного числа со вторым (<, == или >)

if ([intNumber compare: myNumber] == NSOrderedAscending)
 NSLog(@"First number is less than second"); (1-е число меньше 2-го)

[pool drain];
return 0;
}
```

### Вывод программы 15.1

```
100
abcdef
X
100
1.2345e+19
0
Numbers are equal (Числа равны)
First number is less than second (1-е число меньше 2-го)
```

Файл `<Foundation/NSNumber.h>` требуется для работы с числами из класса `NSNumber`.

### Краткий обзор автоматически высвобождаемого пула (autorelease pool)

Первая строка программы 15.1 присутствовала в каждой программе этой книги. В следующей строке выполняется резервирование пространства в памяти для автоматически высвобождаемого пула (пула автоматического освобождения памяти, autorelease-пул), который назначается для переменной `pool`.

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

Autorelease-пул автоматически освобождает память, занимаемую объектами, когда объект добавляется в этот пул. Объект добавляется в пул, когда ему передается сообщение `autorelease`. Когда пул высвобождается, то высвобождаются и все объекты, которые были в него добавлены. Все такие объекты ликвидируются, если для них не было указано, что они существуют вне области действия `autorelease`-пула (это указывается *счетчиками ссылок — reference count*).

Обычно вам не нужно думать о высвобождении объекта, возвращаемого каким-либо методом Foundation. Иногда этим объектом владеет метод, который возвращает его. Иногда объект создается заново и добавляется в `autorelease`-пул самим методом. Однако, как описывается в части I, вы все же должны высвобождать любые объекты (включая объекты Foundation), которые создаете явным образом с помощью метода `alloc`, когда прекращаете их использование.

---

**Примечание.** Вы также должны высвобождать объекты, создаваемые методом `copy` (см. главу 17).

---

В главе 17 дается полное описание счетчиков ссылок, `autorelease`-пула и концепции автоматической сборки мусора (*garbage collection*).

Вернемся к программе 15.1. Класс `NSNumber` содержит много методов, позволяющих создавать объекты `NSNumber` с начальными значениями. Например, в строке

```
intNumber = [NSNumber numberWithInt: 100];
```

создается объект из целой переменной, значение которой равно 100.

Значение, которое считывается из объекта класса `NSNumber`, должно быть согласовано с типом значения объекта. Так, в следующем операторе программы выражение

```
[intNumber integerValue]
```

используется для считывания целого значения, хранящегося в `intNumber`, и оно сохраняется в переменной `myInt` типа `NSInteger`. Отметим, что `NSInteger` — не объект, а `typedef`-определение для базового типа данных. Это тип `long` для 64-битных систем или тип `int` для 32-битных систем. Аналогичный оператор `typecast` задан для `NSUInteger`, чтобы работать с целыми без знака (`unsigned`).

При вызове `NSLog` выполняется приведение типа `NSInteger` `myInt` к `long`. Символы формата `%li` обеспечивают корректную передачу и вывод значения, даже если программа компилируется для 32-битной архитектуры.

Для каждого базового значения метод `class` выделяет память для объекта `NSNumber` и присваивает ему указанное значение. Имена этих методов начинаются с `numberWith`, после чего следует тип, например, `numberWithLong:`, `numberWithFloat:` и т.д. Кроме того, можно использовать методы экземпляра, чтобы присвоить объекту `NSNumber` (для которого была выделена память) указанное значение. Имена этих методов начинаются с `initWith`, например, `initWithLong:` и `initWithFloat:`.

В таблице 15.1 приводится список методов класса и экземпляра, с помощью которых можно задавать значения для объектов @@ `NSNumber`, и соответствующие методы экземпляра для считывания их значений.

**Табл. 15.1.** Методы создания и считывания значений объектов NSNumber

| Метод класса<br>для создания<br>и инициализации | Метод экземпляра<br>для инициализации | Метод экземпляра<br>для считывания |
|-------------------------------------------------|---------------------------------------|------------------------------------|
| numberWithChar:                                 | initWithChar:                         | charValue                          |
| numberWithUnsignedChar:                         | initWithUnsignedChar:                 | unsignedCharValue                  |
| numberWithShort:                                | initWithShort:                        | shortValue                         |
| numberWithUnsignedShort:                        | initWithUnsignedShort:                | unsignedShortValue                 |
| numberWithInteger:                              | initWithInteger:                      | integerValue                       |
| numberWithUnsignedInteger:                      | initWithUnsignedInteger:              | unsignedIntegerValue               |
| numberWithInt:                                  | initWithInt:                          | intValueunsigned                   |
| numberWithUnsignedInt:                          | initWithUnsignedInt:                  | unsignedIntValue                   |
| numberWithLong:                                 | initWithLong:                         | longValue                          |
| numberWithUnsignedLong:                         | initWithUnsignedLong:                 | unsignedLongValue                  |
| numberWithLongLong:                             | initWithLongLong:                     | longlongValue                      |
| numberWithUnsignedLongLong:                     | initWithUnsignedLongLong:             | unsignedLongLongValue              |
| numberWithFloat:                                | initWithFloat:                        | floatValue                         |
| numberWithDouble:                               | initWithDouble:                       | doubleValue                        |
| numberWithBool:                                 | initWithBool:                         | boolValue                          |

Вернемся к программе 15.1. Методы класса служат для создания объектов NSNumber типа long, char, float и double. Рассмотрим, что происходит после создания объекта типа double с помощью строки

```
myNumber = [NSNumber numberWithDouble: 12345e+15];
```

и последующей (неверной) попытки считывания и вывода его значения с помощью строки

```
NSLog(@"%@", [myNumber integerValue]);
```

Выводится результат

```
0
```

Кроме того, мы не получаем от системы сообщения об ошибке. В общем случае именно вы должны обеспечить правильность считывания значения после его сохранения в объекте NSNumber.

В операторе if в выражении с сообщением

```
[intNumber isEqualToNumber: floatNumber]
```

метод isEqualToNumber: выполняет числовое сравнение двух объектов NSNumber по возвращаемому булевому значению.

Метод compare: позволяет сравнивать числовые значения. Выражение с сообщением

```
[intNumber compare: myNumber]
```

возвращает значение `NSOrderedAscending`, если числовое значение, хранящееся в `intNumber`, меньше числового значения, содержащегося в `myNumber`; возвращает значение `NSOrderedSame`, если эти два числа равны; возвращает значение `NSOrderedDescending`, если первое число больше второго. Эти возвращаемые значения определены в заголовочном файле `NSObject.h`.

Отметим, что вы не можете инициализировать значение созданного ранее объекта `NSNumber`. Например, с помощью приведенного ниже оператора нельзя задать значение целого элемента, сохраненного в объекте `NSNumber` `myNumber`.

```
[myNumber initWithInt: 1000];
```

При выполнении программы этот оператор даст ошибку. Все числовые объекты должны создаваться заново, а это означает, что нужно вызывать либо один из методов первого столбца таблицы 15.1 класса `NSNumber`, либо один из методов столбца 2 с результатом метода `alloc`.

```
myNumber = [[NSNumber alloc] initWithInt: 1000];
```

Конечно, при таком способе создания объекта `myNumber` вы сами должны высвобождать занимаемую им память с помощью оператора

```
[myNumber release];
```

Мы будем использовать объекты класса `NSNumber` в остальных программах этой главы.

## Строковые объекты

Мы уже работали со строковыми объектами в предыдущих главах. Заключая последовательность символов в кавычки, как в

`@"Programming is fun"` (Программировать интересно),

мы создаем в Objective-C объект символьной строки. Для работы с объектами символьных строк Foundation framework поддерживает класс с именем `NSString`. С-строки состоят из символов типа `char`, а объекты класса `NSString` состоят из символов типа `unichar`. `Unichar`-символ — это многобайтный символ, соответствующий стандарту Unicode. Это позволяет работать с наборами символов, содержащими буквально миллионы символов. Вам не нужно заботиться о внутреннем представлении этих символов в строках, поскольку класс `NSString` автоматически делает это для вас.<sup>1</sup> С помощью методов из этого класса легко разрабатывать приложения, доступные для локализации, то есть для работы на различных языках по всему миру.

---

<sup>1</sup> В настоящее время `unichar`-символы занимают 16 битов, но стандарт Unicode предусматривает также символы большего размера. Главное — никогда не делать никаких предположений о размере символа Unicode.

Как вы уже знаете, для создания в Objective-C объекта константной символьной строки нужно поместить символ @ перед строкой символов, заключенной в кавычки. Например, с помощью выражения

```
@"Программировать интересно"
```

создается объект константной символьной строки. Это константная символьная строка, принадлежащая классу `NSConstantString`. `NSConstantString` – это подкласс класса строковых объектов `NSString`. Чтобы использовать строковые объекты в программах, нужно включать в них строку

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

## Дополнительно о функции `NSLog`

В программе 15.2 показано, как определить объект класса `NSString` и присвоить ему начальное значение и как использовать символы формата %@ для вывода объекта `NSString`.

### Программа 15.2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *str = @"Программировать интересно";

 NSLog(@"%@", str);

 [pool drain];
 return 0;
}
```

### Вывод программы 15.2

```
Программировать интересно
В операторе
 NSString *str = @"Программировать интересно";
```

Объект константной строки Программировать интересно присваивается переменной str класса `NSString`. Затем ее значение выводится с помощью `NSLog`.

Символы формата %@ в `NSLog` позволяют выводить значения любых объектов. Например, если задано

```
NSNumber *intNumber = [NSNumber numberWithInt: 100];
```

то в результате вызова `NSLog`

```
NSLog(@"%@", intNumber);
```

выводится

100

Мы можем даже применять символы формата %@ для вывода всего содержимого массивов, словарей и наборов. С их помощью можно также выводить объекты ваших собственных классов, если вы замещаете метод description, наследуемый вашим классом. Если нет замещения этого метода, то NSLog просто выводит имя класса и адрес объекта в памяти (что является реализацией по умолчанию метода description, наследуемого из класса NSObject).

## Мутабельные (mutable) и немутабельные (immutable) объекты

С помощью выражения

@"Программировать интересно"

мы создаем объект, содержимое которого нельзя изменить. Такой объект называется *немутабельным (immutable)*. Класс NSString используется для немутабельных строк, но нередко при работе со строками требуется изменять символы внутри строки, например, удалить или заменить некоторые символы. Для работы со строками этого типа предназначен класс NSMutableString.

В программе 15.3 показаны основные способы работы с немутабельными символьными строками.

### Программа 15.3

```
// Основные операции со строками

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
 NSString *str1 = @"This is string A";
 NSString *str2 = @"This is string B";
 NSString *res;
 NSComparisonResult compareResult;

 // Подсчет числа символов

 NSLog(@"%@", [str1 length]);

 // Копирование одной строки в другую

 res = [NSString stringWithString: str1];
 NSLog(@"%@", res);
```

```
// Копирование одной строки в конец другой

str2 = [str1 stringByAppendingString: str2];
NSLog (@"Concatentation: %@", str2);

// Проверка на равенство двух строк

if ([str1 isEqualToString: res] == YES)
 NSLog (@"str1 == res");
else
 NSLog (@"str1 != res");

// Сравнение одной строки с другой (<, == или >)

compareResult = [str1 compare: str2];

if (compareResult == NSOrderedAscending)
 NSLog (@"str1 < str2");
else if (compareResult == NSOrderedSame)
 NSLog (@"str1 == str2");
else // NSOrderedDescending
 NSLog (@"str1 > str2");

// Преобразование символов строки в верхний регистр (в прописные буквы)

res = [str1 uppercaseString];
NSLog (@"Uppercase conversion: %s", [res UTF8String]);

// Преобразование символов строки в нижний регистр (в строчные буквы)

res = [str1 lowercaseString];
NSLog (@"Lowercase conversion: %@", res);

NSLog (@"Original string: %@", str1);

[pool drain];
return 0;
}
```

### Вывод программы 15.3

```
Length of str1: 16 (Длина str1)
Copy: This is string A (Копирование: это строка A)
Concatentation: This is string AThis is string B (Конкатенация: это строка Aэто строка B))
str1 == res
str1 < str2
Uppercase conversion: THIS IS STRING A (Преобразование в верхний регистр: ЭТО
СТРОКА А)
```

Lowercase conversion: this is string a (Преобразование в нижний регистр: это строка a)

Original string: This is string A (Исходная строка: это строка A)

В программе 15.3 сначала объявляются три немутабельных объекта класса `NSString`: `str1`, `str2` и `res`. Первые два инициализируются как объекты константных символьных строк. В объявлении

```
NSComparisonResult compareResult;
```

указывается, что `compareResult` будет содержать результат операции сравнения строк, которая будет выполнена в этой программе.

Метод `length` можно использовать для подсчета числа символов строки. Он возвращает целое значение без знака типа `NSUInteger`. Вывод подтверждает, что строка @"This is string A" действительно содержит 16 символов. Оператор

```
res = [NSString stringWithFormat: str1];
```

показывает, как создать новую символьную строку с содержимым другой строки. Результатирующий объект класса `NSString` присваивается объекту `res` и затем выводится для подтверждения результатов. Здесь создается фактическая копия содержимого строки, а не только другая ссылка на ту же строку в памяти. Это означает, что `str1` и `res` ссылаются на два различных строковых объекта, что отличается от простого присваивания

```
res = str1;
```

где просто создается еще одна ссылка на тот же объект в памяти.

Метод `stringByAppendingString:` позволяет объединять две символьные строки. Например, с помощью выражения

```
[str1 stringByAppendingString: str2]
```

создается новый строковый объект, содержащий символы из `str1`, после которых следуют символы из `str2`, и этот объект возвращается как результат. Эта операция не влияет на исходные строковые объекты `str1` и `str2`, поскольку они являются немутабельными строковыми объектами.

Затем метод `isEqualToString:` проверяет две строки на равенство символов. Метод `compare:` позволяет определить их упорядоченность, например, сортировку массива строк. Аналогично методу `compare:`, который мы использовали выше для сравнения двух объектов класса `NSNumber`, результатом сравнения является `NSOrderedAscending`, если первая строка лексически меньше второй строки, `NSOrderedSame`, если строки равны, и `NSOrderedDescending`, если первая строка лексически больше второй. Если вам не нужно учитывать регистр букв, используйте метод `caseInsensitiveCompare:` вместо метода `compare:`. В этом случае два строковых объекта @"Gregory" и @"gregory" будут считаться равными.

Методы `uppercaseString` и `lowercaseString` преобразуют строки в верхний и нижний регистр соответственно. Это преобразование тоже не влияет на исходные строки, что подтверждается последней строкой вывода.

В программе 15.4 показаны дополнительные методы для работы со строками. С помощью этих методов можно извлекать подстроки и проверять одну строку на присутствие в другой строке.

В некоторых методах требуется идентифицировать подстроку, указав диапазон. *Диапазон (range)* задается с помощью начального индекса (порядкового номера) и числа символов. Порядковые номера начинаются с нуля; например, первые три символа строки указываются парой чисел {0, 3}. В некоторых методах класса *NSString* (а также других классов Foundation) для описания диапазона используется специальный тип данных *NSRange*. Он определен в файле *<Foundation/NSRange.h>* (который включен для вас из *<Foundation/NSString.h>*) и является на самом деле *typedef*-определением структуры, содержащей два компонента: *location* (позиция) и *length* (длина), каждый из которых определен с типом *NSUInteger*. Этот тип данных используется в программе 15.4.

---

**Примечание.** Описание структур см. в главе 13 и в следующих разделах этой главы.

---

#### Программа 15.4

```
// Основные операции со строками - Продолжение

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
 // Извлечение первых трех символов из строки

 res = [str1 substringToIndex: 3];
 NSLog (@"First 3 chars of str1: %@", res);

 // Извлечение символов до конца строки, начиная с индекса 5

 res = [str1 substringFromIndex: 5];
 NSLog (@"Chars from index 5 of str1: %@", res);

 // Извлечение символов, начиная с номера 8, до номера 13 (6 символов)

 res = [[str1 substringFromIndex: 8] substringToIndex: 6];
 NSLog (@"Chars from index 8 through 13: %@", res);

 // Более простой способ сделать то же самое

 res = [str1 substringWithRange: NSMakeRange (8, 6)];
 NSLog (@"Chars from index 8 through 13: %@", res);
```

```
// Проверка присутствия одной строки внутри другой

subRange = [str1 rangeOfString: @"string A"];
NSLog (@"String is at index %lu, length is %lu",
 subRange.location, subRange.length);

subRange = [str1 rangeOfString: @"string B"];

if (subRange.location == NSNotFound)
 NSLog (@"String not found");
else
 NSLog (@"String is at index %lu, length is %lu",
 subRange.location, subRange.length);

[pool drain];
return 0;
}
```

**Вывод программы 15.4**

First 3 chars of str1: Thi (Первые 3 символа str1)  
 Chars from index 5 of str1: is string A (Символы str1, начиная с номера 5)  
 Chars from index 8 through 13: string (Символы с номерами 8-13)  
 Chars from index 8 through 13: string  
 String is at index 8, length is 8 (Строка, начиная с номера 8, длина 8)  
 String not found (Строка не найдена)

Метод `substringToIndex`: создаст подстроку от первого символа указанной строки до символа с указанным индексом (но не включая сам этот символ). Поскольку отсчет индексов начинается с 0, аргумент со значением 3 означает, что нужно извлечь из строки символы с номерами 0, 1 и 2 и возвратить результирующий строковый объект. Для любого из строковых методов, которым в качестве одного из аргументов передается индекс, выводится сообщение об ошибке «*Range or index out of bounds*» (Диапазон или индекс вне допустимых границ), если указан неверный индекс для данной строки.

Метод `substringFromIndex`: возвращает подстроку из строки-получателя сообщения от символа с указанным индексом до конца строки. Выражение

```
res = [[str1 substringFromIndex: 8] substringToIndex: 6];
```

показывает, как сочетать эти методы для извлечения подстроки символов изнутри строки. Сначала используется метод `substringFromIndex`: для извлечения символов, начиная с номера 8, вплоть до конца строки. Затем к результату применяется метод `substringToIndex`: для получения первых 6 символов. Конечным результатом является подстрока, представляющая диапазон символов {8, 6} из исходной строки.

Метод `substringWithRange:` делает за один шаг то, что мы только что сделали за два. Ему передается диапазон, и он возвращает строку в указанном диапазоне.

#### Специальная функция

`NSMakeRange (8, 6)`

создает диапазон из своего аргумента и возвращает результат, который передается как аргумент методу `substringWithRange:`.

Чтобы найти одну строку внутри другой, можно использовать метод `rangeOfString:`. Если указанная строка найдена внутри строки-получателя, возвращаемое значение диапазона точно указывает, где найдена эта строка. Если строка не найдена, то компонент `location` содержит значение `NSNotFound`. Например, оператор

```
subRange = [str1 rangeOfString: @"string A"];
```

присваивает структуру `NSRange`, возвращаемую этим методом, переменной `subRange` типа `NSRange`. Обратите внимание, что `subRange` – это не переменная-объект, а переменная-структура (объявление `subRange` в этой программе не содержит звездочки). Компоненты этой структуры можно получать с помощью оператора «точка». Таким образом, выражение `subRange.location` дает значение компонента `location` (позиция) этой структуры, а `subRange.length` дает значение компонента `length` (длина). Эти значения передаются для вывода функции  `NSLog`.

## Мутабельные строки

Для создания строковых объектов, символы которых доступны для изменения, применяется класс `NSMutableString`. Поскольку он является подклассом `NSString`, можно использовать все методы класса `NSString`.

Когда мы сравниваем возможности мутабельных строковых объектов с немутабельными, то имеем в виду изменение конкретных символов внутри строки. Как мутабельному, так и немутабельному строковому объекту во время выполнения программы можно присвоить совершенно другой строковый объект.

```
str1 = @"This is a string";
...
str1 = [str1 substringFromIndex: 5];
```

В данном случае объекту `str1` сначала присваивается константный строковый объект, затем в программе ему присваивается подстрока. В данном случае `str1` можно объявить и как мутабельный, и как немутабельный строковый объект. В программе 15.5 показаны некоторые способы работы с мутабельными строками в программах.

**Программа 15.5**

```
// Основные операции со строками - мутабельные строки

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *str1 = @"This is string A";
 NSString *search, *replace;
 NSMutableString *mstr;
 NSRange substr;

 // Создание мутабельных строк из немутабельных

 mstr = [NSMutableString stringWithString: str1];
 NSLog(@"%@", mstr);

 // Вставка символов

 [mstr insertString: @" mutable" atIndex: 7];
 NSLog(@"%@", mstr);

 // Фактическая конкатенация при вставке в конец

 [mstr insertString: @" and string B" atIndex: [mstr length]];
 NSLog(@"%@", mstr);

 // Или непосредственное использование appendString

 [mstr appendString: @" and string C"];
 NSLog(@"%@", mstr);

 // Удаление подстроки с указанным диапазоном

 [mstr deleteCharactersInRange: NSMakeRange (16, 13)];
 NSLog(@"%@", mstr);

 // Сначала определение диапазона и затем его использование для удаления

 substr = [mstr rangeOfString: @"string B and "];

 if (substr.location != NSNotFound) {
 [mstr deleteCharactersInRange: substr];
```

```
 NSLog(@"%@", mstr);
}

// Непосредственное задание мутабельной строки

[mstr setString: @"This is string A"];
NSLog(@"%@", mstr);

// Теперь заменяем диапазон символов другой строкой
[mstr replaceCharactersInRange: NSMakeRange(8, 8)
 withString: @"a mutable string"];
NSLog(@"%@", mstr);

// Поиск и замена

search = @"This is";
replace = @"An example of";

substr = [mstr rangeOfString: search];

if (substr.location != NSNotFound) {
 [mstr replaceCharactersInRange: substr
 withString: replace];
 NSLog(@"%@", mstr);
}

// Поиск и замена всех экземпляров

search = @"a";
replace = @"X";

substr = [mstr rangeOfString: search];

while (substr.location != NSNotFound) {
 [mstr replaceCharactersInRange: substr
 withString: replace];
 substr = [mstr rangeOfString: search];
}

NSLog(@"%@", mstr);

[pool drain];
return 0;
}
```

### Вывод программы 15.5

```

This is string A (Это строка A)
This is mutable string A (Это мутабельная строка A)
This is mutable string A and string B (Это мутабельные строка A и строка B)
This is mutable string A and string B and string C (Это мутабельные строка A, строка B и строка C)
This is mutable string B and string C (Это мутабельные строка B и строка C)
This is mutable string C (Это мутабельная строка C)
This is string A (Это строка A)
This is a mutable string (Это мутабельная строка)
An example of a mutable string
An exXmple of X mutXble string

```

Объявление

```
NSMutableString *mstr;
```

определяет mstr как переменную со строковым объектом, содержимое которого может изменяться во время выполнения программы. В операторе

```
mstr = [NSMutableString stringWithString: str1];
```

переменной mstr присваивается строковый объект, содержимое которого является копией символов, содержащихся в str1, то есть "This is string A". Если метод `stringWithString:` передается классу `NSMutableString`, возвращаемым результатом является мутабельный строковый объект. Если он передается классу `NSString`, как в программе 15.5, то мы получаем немутабельный строковый объект.

Метод `insertStringAtIndex:` выполняет вставку указанной символьной строки в строку-получатель, начиная с указанного индекса (порядкового номера). В данном случае выполняется вставка @"mutable" в строку, начиная с индекса 7, то есть перед восьмым символом строки. В отличие от методов для немутабельных строковых объектов, никакого значения не возвращается, поскольку модифицируется строка-получатель. При втором вызове `insertStringAtIndex:` используется метод `length` для вставки одной символьной строки в конец другой. Метод `appendString:` позволяет выполнить эту задачу несколько проще.

С помощью метода `deleteCharactersInRange:` можно удалить заданное число символов из строки. Применение диапазона {16, 13} к строке

```
This is mutable string A and string B and string C
```

позволяет удалить 13 символов "string A and", начиная с индекса 16 (то есть с 17-го символа строки). Это показано на рис. 15.1.

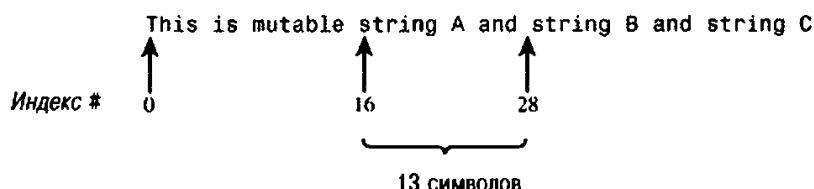


Рис. 15.1. Индексы позиций в строке

В следующих строках программы 15.5 метод `rangeOfString:` позволяет сначала найти строку, а затем удалить ее. После проверки, что строка @"string B and" действительно содержит `mstr`, применяется метод `deleteCharactersInRange:` для удаления этих символов с использованием диапазона, возвращаемого методом `rangeOfString:`, в качестве аргумента.

Метод `setString:` применяется для непосредственного задания содержимого мутабельного строкового объекта. После присвоения строки @"This is string A" несколько символов этой строки заменяются другой строкой с помощью метода `replaceCharactersInRange:`. Размеры заменяемой и заменяющей строк могут быть неодинаковыми; одну строку можно заменить строкой равного или неравного размера. Например, в операторе

```
[mstr replaceCharactersInRange: NSMakeRange(8, 8)
 withString: @"a mutable string"];
```

8 символов "string A" заменяются на 16 символов "a mutable string".

В остальной части этой программы показано, как выполнять операции поиска и замены. В первом случае внутри строки `mstr` ищется строка @"This is" (сначала `mstr` была присвоена строка @"This is a mutable string"). Эта строка находится в строке поиска и заменяется строкой @"An example of". В результате содержимое `mstr` заменяется строкой @"An example of a mutable string".

Далее в программе выполняется цикл, показывающий, как выполнить операцию «найти и заменить все». Строке поиска присваивается значение @"a", и строке замены присваивается значение @"X".

Если строка замены содержит также строку поиска (например, нужно заменить строку "a" на строку "aX"), то мы получим бесконечный цикл.

Во-вторых, в случае пустой строки замены (когда она не содержит никаких символов) мы фактически удаляем все экземпляры строки поиска. Пустая константная символьная строка задается парой смежных кавычек без пробелов:

```
replace = @"";
```

Конечно, если мы хотим просто удалить экземпляр строки, то можно использовать метод `deleteCharactersInRange:`.

И, наконец, класс `NSMutableString` содержит метод с именем `replaceOccurrencesOfString:withString:options:range:` для операции «найти и заменить все» в строке. Цикл while программы 15.5 можно было бы заменить одним оператором.

```
[mstr replaceOccurrencesOfString: search
 withString: replace
 options: nil
 range: NSMakeRange (0, [mstr length])];
```

Так можно избежать бесконечного цикла, поскольку это предусмотрено в самом методе.

## Откуда берутся все эти объекты?

В программах 15.4 и 15.5 выполняется работа со многими строковыми объектами, которые создаются и возвращаются разными методами классов `NSString` и `NSMutableString`. Вам не нужно думать об освобождении памяти, занимаемой этими объектами; это предусмотрено в методах-создателях объектов. Предполагается, что создатели добавили все эти объекты в автоматически высвобождаемый пул (`autorelease`-пул), и объекты будут высвобождены, когда будет высвобожден сам пул. Однако если вы разрабатываете программу, которая создает много временных объектов, то память, используемая ими, может накапливаться. В таких случаях, возможно, придется освобождать память во время выполнения программы. Это описывается в главе 17. На данный момент просто утите, что объекты занимают память, которая может увеличиваться по мере выполнения программы.

Класс `NSString` содержит более 100 методов, которые могут работать с немутабельными строковыми объектами. В таблице 15.2 приводятся наиболее распространенные методы, а в таблице 15.3 – некоторые дополнительные методы из класса `NSMutableString`. Другие важные методы класса `NSString` (например, работающие с именами путей и читающие содержимое файла в строку) будут вводиться далее.

В таблицах 15.2 и 15.3, *url* – это объект `NSURL`, *path* – это объект `NSString`, указывающий путь к файлу, *nsstring* – это объект `NSString`, *i* – это значение типа `NSUInteger`, представляющее допустимый номер символа в строке, *enc* – это объект `NSStringEncoding`, указывающий кодировку символов, *err* – это объект `NSError`, который описывает ошибку, если она возникла, *size* (размер) и *opts* (опции) имеют тип `NSUInteger`, и *range* – это объект `NSRange`, указывающий допустимый диапазон символов в списке.

**Табл. 15.2. Наиболее распространенные методы класса `NSString`**

| Метод                                                                       | Описание                                                                                                                                                                            |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>+ (id) stringWithContentsOfFile: path encoding: enc error: err</code> | Создает новую строку и присваивает ей содержимое файла, указанного аргументом <i>path</i> , в кодировке символов <i>enc</i> ; возвращает ошибку в <i>err</i> , если не <i>nil</i> . |
| <code>+ (id) stringWithContentsOfURL: url encoding: enc error: err</code>   | Создает новую строку и присваивает ей содержимое <i>url</i> в кодировке символов <i>enc</i> ; возвращает ошибку в <i>err</i> , если не <i>nil</i> .                                 |
| <code>+ (id) string</code>                                                  | Создает новую пустую строку.                                                                                                                                                        |
| <code>+ (id) stringWithString: nsstring</code>                              | Создает новую строку и присваивает ей <i>nsstring</i> .                                                                                                                             |
| <code>- (id) initWithString: nsstring</code>                                | Присваивает вновь выделяемой строке <i>nsstring</i> .                                                                                                                               |
| <code>- (id) initWithContentsOfFile: path encoding: enc error: err</code>   | Присваивает строке содержимое файла, указанного аргументом <i>path</i> .                                                                                                            |

| Метод                                                                       | Описание                                                                                                                                                                          |
|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>- (id) initWithContentsOfURL:<br/>url encoding: enc error: err</code> | Присваивает строке содержимое <code>url</code> ( <code>NSURL *</code> ) в кодировке символов <code>enc</code> ; возвращает ошибку в <code>err</code> , если не <code>nil</code> . |
| <code>- (NSUInteger) length</code>                                          | Возвращает число символов в строке.                                                                                                                                               |
| <code>- (unichar) characterAtIndex: i</code>                                | Возвращает символ Unicode, находящийся в позиции с индексом <code>i</code> .                                                                                                      |
| <code>- (NSString *) substringFromIndex: i</code>                           | Возвращает подстроку от символа с индексом <code>i</code> до конца.                                                                                                               |
| <code>- (NSString *)<br/>substringWithRange: range</code>                   | Возвращает подстроку в соответствии с указанным диапазоном.                                                                                                                       |
| <code>- (NSString *) substringToIndex: i</code>                             | Возвращает подстроку от начала строки вплоть до символа с индексом <code>i</code> .                                                                                               |
| <code>- (NSComparator *)<br/>caseInsensitiveCompare: nsstring</code>        | Сравнение двух строк независимо от регистра букв.                                                                                                                                 |
| <code>- (NSComparator *) compare:<br/>nsstring</code>                       | Сравнение двух строк.                                                                                                                                                             |
| <code>- (BOOL) hasPrefix: nsstring</code>                                   | Проверка, что строка начинается с <code>nsstring</code> .                                                                                                                         |
| <code>- (BOOL) hasSuffix: nsstring</code>                                   | Проверка, что строка заканчивается <code>nsstring</code> .                                                                                                                        |
| <code>- (BOOL) isEqualToString: nsstring</code>                             | Проверка, что две строки равны.                                                                                                                                                   |
| <code>- (NSString *) capitalizedString</code>                               | Возвращает строку, делая прописной первую букву каждого слова (и строчными все остальные буквы каждого слова).                                                                    |
| <code>- (NSString *) lowercaseString</code>                                 | Возвращает строку, преобразованную в нижний регистр.                                                                                                                              |
| <code>- (NSString *) uppercaseString</code>                                 | Возвращает строку, преобразованную в верхний регистр.                                                                                                                             |
| <code>- (const char *) UTF8String</code>                                    | Возвращает строку, преобразованную в C-строку в кодировке UTF-8.                                                                                                                  |
| <code>- (double) doubleValue</code>                                         | Возвращает строку, преобразованную в значение типа <code>double</code> .                                                                                                          |
| <code>- (float) floatValue</code>                                           | Возвращает строку, преобразованную в значение с плавающей точкой.                                                                                                                 |
| <code>- (NSInteger) integerValue</code>                                     | Возвращает строку, преобразованную в целое значение <code>NSInteger</code> .                                                                                                      |
| <code>- (int) intValue</code>                                               | Возвращает строку, преобразованную в целое значение.                                                                                                                              |

Методы, которые приводятся в таблице 15.3, создают или модифицируют объекты класса `NSMutableString`.

**Табл. 15.3.** Наиболее распространенные методы класса `NSMutableString`

| Метод                                                                                                       | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>+ (id) stringWithCapacity: size</code>                                                                | Создает строку, содержащую первоначально <code>size</code> символов.                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>- (id) initWithCapacity: size</code>                                                                  | Инициализирует строку с начальной длиной <code>size</code> символов.                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>- (void) setString: nsstring</code>                                                                   | Присваивает строке <code>nsstring</code> .                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>- (void) appendString: nsstring</code>                                                                | Добавляет <code>nsstring</code> в конец строки-получателя.                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>- (void) deleteCharactersInRange: range</code>                                                        | Удаляет символы в указанном диапазоне.                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>- (void) insertString: nstring atIndex: i</code>                                                      | Выполняет вставку <code>nsstring</code> в строку-получатель, начиная с индекса <code>i</code> .                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>- (void) replaceCharactersInRange: range withString: nsstring</code>                                  | Выполняет замену символов в указанном диапазоне на <code>nsstring</code> .                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>- (void) replaceOccurrencesOfString: nsstring withString: nsstring2 options: opts range: range</code> | Выполняет замену всех экземпляров <code>nsstring</code> на <code>nsstring2</code> в указанном диапазоне и в соответствии с опциями <code>opts</code> . Опции могут представлять побитовую OR-комбинацию <code>NSBackwardsSearch</code> ( поиск на чинается с конца диапазона), <code>NSAnchoredSearch</code> (совпадение <code>nsstring</code> должно быть только с начала диапазона), <code>NSLiteralSearch</code> (выполняется побайтовое сравнение) и <code>NSCaseInsensitiveSearch</code> . |

Объекты класса `NSString` широко используются в этой книге. Для разбора строк на маркеры нужно рассмотреть класс `Foundation NSScanner`.

## Объекты-массивы

Массив в Foundation – это упорядоченный набор объектов. Чаще всего (но не обязательно) элементы массива имеют один определенный тип. Аналогично мутабельным и немутабельным строкам, существуют мутабельные и немутабельные массивы. Для работы с *немутабельными* (*immutable*) массивами используется класс `NSArray`; для *мутабельных* (*mutable*) массивов используется класс `NSMutableArray`. Последний класс является подклассом предыдущего и наследует его методы. Для работы с объектами-массивами в программах нужно включить строку

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

В программе 15.6 задается массив для хранения названий месяцев, а затем выводятся названия месяцев.

### Программа 15.6

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

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

 // Создание массива, содержащего названия месяцев

 NSArray *monthNames = [NSArray arrayWithObjects:
 @"January", @"February", @"March", @"April",
 @"May", @"June", @"July", @"August", @"September",
 @"October", @"November", @"December", nil];

 // Теперь вывод всех элементов этого массива

 NSLog(@"Month Name");
 NSLog(@"===== =====");

 for (i = 0; i < 12; ++i)
 NSLog(@"%2i %@", i + 1, [monthNames objectAtIndex: i]);

 [pool drain];
 return 0;
}
```

### Вывод программы 15.6

| Month | Name      |
|-------|-----------|
| 1     | January   |
| 2     | February  |
| 3     | March     |
| 4     | April     |
| 5     | May       |
| 6     | June      |
| 7     | July      |
| 8     | August    |
| 9     | September |
| 10    | October   |
| 11    | November  |
| 12    | December  |

Метод этого класса `arrayWithObjects:` создает массив со списком объектов в виде его элементов. Объекты перечисляются по порядку и разделяются запятыми. Это специальный синтаксис методов, которые принимают переменное число аргументов. Чтобы закончить список, нужно поставить `nil` в качестве последнего значения списка (`nil` не сохраняется внутри массива).

В программе 15.7 массиву `monthNames` присваиваются 12 строковых значений в виде аргументов для метода `arrayWithObjects:`.

Элементы массива идентифицируются своим индексом (порядковым номером). Аналогично объектам класса `NSString`, индексирование начинается с нуля, поэтому массив, содержащий 12 элементов, имеет допустимые индексы 0-11. Метод `objectAtIndex:` считывает элемент массива по его индексу.

В этой программе каждый элемент из массива считывается с помощью метода `objectAtIndex:` в цикле `for`. Каждый считываемый элемент выводится с помощью  `NSLog`.

В программе 15.7 создается таблица простых (`prime`) чисел. Поскольку простые числа добавляются в массив по мере их создания, здесь требуется мутабельный массив. Память для объекта `primes` класса `NSMutableArray` выделяется с помощью метода `arrayWithCapacity:`. Указанный аргумент 20 задаст начальный размер массива; размер мутабельного массива автоматически увеличивается во время выполнения программы.

Хотя простые числа являются целыми, мы не можем сохранять значения типа `int` внутри этого массива. Данный массив может содержать только объекты, поэтому нам нужно сохранять в массиве `primes` целые объекты класса `NSNumber`.

### Программа 15.7

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSNumber.h>
```

```
#define MAXPRIME 50

int main (int argc, char *argv[])
{
 int i, p, prevPrime;
 BOOL isPrime;
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 // Создание массива для хранения простых чисел

 NSMutableArray *primes =
 [NSMutableArray arrayWithCapacity: 20];

 // Сохранение в массиве первых двух простых чисел (2 и 3)

 [primes addObject: [NSNumber numberWithInteger: 2]];
 [primes addObject: [NSNumber numberWithInteger: 3]];

 // Вычисление остальных простых чисел

 for (p = 5; p <= MAXPRIME; p += 2) {
 // проверяется, что p - простое число

 isPrime = YES;

 i = 1;

 do {
 prevPrime = [[primes objectAtIndex: i] integerValue];

 if (p % prevPrime == 0)
 isPrime = NO;
 ++i;
 } while (isPrime == YES && p / prevPrime >= prevPrime);

 if (isPrime)
 [primes addObject: [NSNumber numberWithInteger: p]];
 }

 // Вывод результатов

 for (i = 0; i < [primes count]; ++i)
 NSLog(@"%@", (long) [[primes objectAtIndex: i] integerValue]);

 [pool drain];
 return 0;
}
```

**Вывод программы 15.7**

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

`kMaxPrime` определяется как максимальное простое число для вычислений (в данном случае 50).

После выделения памяти для массива `primes` задаются два первых элемента массива с помощью следующих операторов.

```
[primes addObject: [NSNumber numberWithInt: 2]];
[primes addObject: [NSNumber numberWithInt: 3]];
```

Метод `addObject:` добавляет объект в конец массива. В данном случае добавляются объекты класса `NSNumber`, создаваемые соответственно из целых значений 2 и 3.

Затем в программе начинается цикл `for` для поиска простых чисел, начиная с 5, вплоть до `kMaxPrime` с пропуском промежуточных четных чисел (`r += 2`).

Для каждого возможного числа `r` проверяется его делимость на предыдущие простые числа. В случае делимости `r` не является простым числом. Для ускорения мы проверяем делимость только на простые числа, не превышающие квадратный корень из `r`. Дело в том, что если число не является простым, оно должно делиться на простое число, которое не больше его квадратного корня. Поэтому выражение

```
r / prevPrime >= prevPrime
```

верно, если `prevPrime` меньше, чем квадратный корень из `r`.

Если при выходе из цикла `do-while` флаг `isPrime` по-прежнему равен YES, значит, мы нашли еще одно простое число. В этом случае `r` добавляется в массив `primes`, и выполнение программы продолжается.

Краткое замечание по эффективности. Классы Foundation очень удобны для работы с массивами, однако при работе с большими массивами чисел и сложными алгоритмами нужно научиться выполнять такие задачи с помощью низкоуровневых конструкций языка для массивов, которые могут оказаться более эффективными с точки зрения использования памяти и скорости выполнения. См. раздел «Массивы» в главе 13.

## Создание адресной книги

Рассмотрим пример создания адресной книги.<sup>2</sup> Она будет содержать адресные карточки. Для упрощения эти адресные карточки будут содержать только имя человека и его адрес электронной почты. Можно легко расширить эти данные, добавив другую информацию, например, почтовый адрес и номер телефона, но мы оставляем это вам как упражнение в конце главы.

### Создание адресной карточки

Мы начинаем с определения нового класса `AddressCard`. Нам нужны средства для создания новой адресной карточки, задания ее полей имени (`name`) и электронной почты (`email`), чтения этих полей и вывода карточки. В графической среде можно было бы использовать некоторые удобные процедуры, например, из фреймворка `Application Kit`, чтобы рисовать карточку на экране.

В программе 15.8 показан файл секции `interface` для нового класса `AddressCard`. Мы не будем пока синтезировать методы доступа (`accessor methods`); мы напишем их сами в качестве упражнения.

**Программа 15.8.** Файл секции `interface AddressCard.h`

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

@interface AddressCard: NSObject

{
 NSString *name;
 NSString *email;
}

-(void) setName: (NSString *) theName;
-(void) setEmail: (NSString *) theEmail;

-(NSString *) name;
-(NSString *) email;

-(void) print;

@end
```

<sup>2</sup> Mac OS X содержит полный фреймворк `Address Book`, предоставляющий исключительно мощные возможности для работы с адресными книгами.

Это легко сделать с помощью файла секции `implementation` программы 15.8.

**Программа 15.8.** Файл секции `implementation` `AddressCard.m`

```
#import "AddressCard.h"

@implementation AddressCard

-(void) setName: (NSString *) theName
{
 name = [[NSString alloc] initWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
 email = [[NSString alloc] initWithString: theEmail];
}

-(NSString *) name
{
 return name;
}

-(NSString *) email
{
 return email;
}

-(void) print
{
 NSLog (@"=====");
 NSLog(@"%@", name);
 NSLog(@"%@", email);
 NSLog(@"%@", "\n");
 NSLog(@"%@", "\n");
 NSLog(@"%@", "\n");
 NSLog(@"%010d %010d", 0, 0);
 NSLog(@"=====");
}
@end
```

Можно было бы сделать так, чтобы методы `setName:` и `setEmail:` сохраняли объекты непосредственно в своих переменных экземпляра с помощью следующих определений методов.

```
-(void) setName: (NSString *) theName
{
 name = theName;
```

```
)

-(void) setEmail: (NSString *) theEmail
{
 email = theEmail;
}
```

Но тогда объект класса AddressCard не будет владеть своими объектами-членами. Мы уже говорили в главе 8 о получении объектом владения применительно к классу Rectangle, владеющему своим объектом origin.

Определение этих методов следующим способом тоже неверно, поскольку методы AddressCard тоже не будут владеть своими объектами name и email — ими будет владеть NSString.

```
- (void) setName: (NSString *) theName
{
 name = [NSString stringWithFormat: theName];
}

- (void) setEmail: (NSString *) theEmail
{
 email = [NSString stringWithFormat: theEmail];
}
```

Вернемся к программе 15.8. Метод print представляет пользователя в виде адресной карточки в формате, напоминающем карточку Rolodex (они использовались в картотеках). Символы %-31s при вызове NSLog указывают вывод в виде С-строки UTF8 при ширине поля 31 символ с выравниванием по левому краю. Так пользователь сможет брать карточку за правый край.

После создания класса AddressCard мы можем написать тестовую программу для создания адресной карточки, задания ее значений и ее вывода (см. программу 15.8).

#### Программа 15.8. Тестовая программа

```
#import "AddressCard.h"
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 AddressCard *card1 = [[AddressCard alloc] init];
```

```
[card1 setName: aName];
[card1 setEmail: aEmail];

[card1 print];

[card1 release];
[pool drain];
return 0;
}
```

### Вывод программы 15.8

```
=====
| Julia Kochan
| jewls337@axlc.com
|
| 0 0
=====
```

В этой программе строка

```
[card1 release];
```

применяется для освобождения памяти, которая занята адресной карточкой. Из предыдущих глав вы должны понимать, что высвобождение объекта класса AddressCard таким способом не приводит к освобождению памяти, которую мы выделили для его членов name и email. Чтобы избежать утечки памяти для класса AddressCard, нужно заместить метод dealloc, высвобождая эти члены при освобождении памяти для объекта AddressCard. Ниже приводится замещающий метод dealloc для класса AddressCard.

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

Метод dealloc должен высвобождать свои собственные переменные экземпляра до использования super для ликвидации самого объекта. Объект становится недействительным после того, как освобождена память объекта (dealloc).

Чтобы избежать утечки памяти для класса AddressCard, нужно также внести изменения в методы setName: и setEmail: и освобождать память, которая используется объектами, сохраненными в их переменных экземпляра. Если кто-то изменяет имя на карточке, мы должны освободить память, которая используется старым именем, прежде чем заменить его новым именем. Для адреса элек-

тронной почты мы тоже должны освободить память, которая используется для этого адреса, прежде чем заменить его новым.

Ниже приводятся новые методы `setName:` и `setEmail:` для класса, который правильно управляет памятью.

```
- (void) setName: (NSString *) theName
{
 [name release];
 name = [[NSString alloc] initWithString: theName];
}

-(void) setEmail: (NSString *) theEmail
{
 [email release];
 email = [[NSString alloc] initWithString: theEmail];
}
```

Мы можем отправлять сообщение `nil`-объекту; поэтому выражения с сообщениями

```
[name release];
и
[email release];
```

допустимы, даже если `name` или `email` не были заданы ранее.

## Синтезируемые методы AddressCard

Теперь, когда описан подходящий способ написания методов доступа `setName:` и `setEmail:`, мы можем вернуться к началу и позволить самой системе генерировать методы доступа (accessor method). Рассмотрим второй вариант файла секции `interface AddressCard`.

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

@interface AddressCard: NSObject
{
 NSString *name;
 NSString *email;
}

@property (copy, nonatomic) NSString *name, *email;
-(void) print;
@end

В строке

@property (copy, nonatomic) NSString *name, *email;
```

содержатся *атрибуты* *copy* и *nonatomic* для свойств (*property*). Атрибут *copy* указывает, что нужно создать копию переменной экземпляра в ее методе-установщике (*setter*), как мы делали в предыдущей версии. Действие по умолчанию – не создавать копию, а просто выполнить присваивание (атрибут по умолчанию *assign*), что является неверным подходом, как мы выяснили выше.

Атрибут *nonatomic* указывает, что метод-получатель (*getter*) не должен *удерживать* (*retain*) или *автоматически высвобождать* (*autorelease*) переменную экземпляра, прежде чем возвратить ее значение. В главе 18 эта тема описывается более подробно.

Программа 15.9 – это новый файл секции *implementation* *AddressCard.m*, который указывает, что методы доступа будут синтезированы.

**Программа 15.9.** Файл секции *implementation* *AddressCard.m* с синтезируемыми методами

```
#import "AddressCard.h"
@implementation AddressCard

@synthesize name, email;
-(void) print
{
 NSLog (@"=====");
 NSLog (@" |");
 NSLog (@" | % -31s |", [name UTF8String]);
 NSLog (@" | % -31s |", [email UTF8String]);
 NSLog (@" |");
 NSLog (@" |");
 NSLog (@" |");
 NSLog (@" | 0 0");
 NSLog (@"=====");
}

@end
```

Мы оставляем вам в качестве упражнения проверку того, что это новое определение *AddressCard* с синтезируемыми методами доступа работает с тестовой программой, показанной в программе 15.9.

Теперь добавим еще один метод в класс *AddressCard*. Предположим, что мы хотим задавать поля *name* и *email* с помощью одного вызова. Чтобы сделать это, мы добавим новый метод *setName:andEmail:*.<sup>3</sup> Он имеет следующий вид.

```
- (void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
 self.name = theName;
 self.email = theEmail;
}
```

<sup>3</sup> Вам может потребоваться метод инициализации *initWithName:andEmail:*, но мы не будем показывать его здесь.

Полагаясь на синтезируемые методы-установщики для задания соответствующих переменных экземпляра (вместо их непосредственного задания внутри самого метода), мы повышаем уровень абстрагирования, делая программу более независимой от внутренних структур данных. Мы также используем свойства синтезируемого метода; в данном случае это копирование (*copy*) вместо присваивания (*assign*) значения переменной экземпляра.

Этот метод тестируется в программе 15.9.

### Программа 15.9. Тестовая программа

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

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

 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";

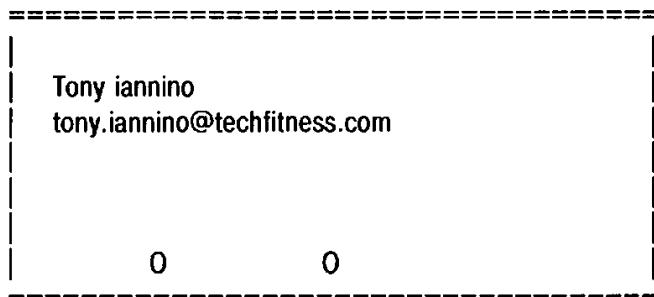
 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];

 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];

 [card1 print];
 [card2 print];
 [card1 release];
 [card2 release];
 [pool drain];
 return 0;
}
```

### Вывод программы 15.9

```
=====
| Julia Kochan
| jewls337@axlc.com
|
| 0 0
=====
```



Класс `AddressCard`, казалось бы, действует правильно. Но как быть, если нужно работать с большим числом адресных карточек (объектов `AddressCard`)? Видимо, имеет смысл собрать их вместе. Именно это мы и сделаем, определив новый класс `AddressBook` (Адресная книга). В классе `AddressBook` будет храниться имя адресной книги и набор адресных карточек в объекте-массиве. Начнем со средств создания новой адресной книги, добавления в нее новых адресных карточек, определения числа содержащихся в ней записей и вывода списка ее содержимого. Затем нам потребуются средства поиска в этой адресной книге, удаления записей, редактирования существующих записей, их сортировки и создания копии содержимого.

Начнем с простого файла секции `interface` (`Addressbook.h`).

#### Программа 15.10. Файл секции `interface` `Addressbook.h`

```

#import <Foundation/NSArray.h>
#import "AddressCard.h"

@interface AddressBook: NSObject
{
 NSString *bookName;
 NSMutableArray *book;
}

-(id) initWithName: (NSString *) name;
-(void) addCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;
-(void) dealloc;

@end

```

Метод `initWithName:` создает начальный массив для адресных карточек и сохраняет имя адресной книги. Метод `addCard:` добавляет отдельную адресную карточку (`AddressCard`) в книгу. Метод `entries` возвращает число адресных карточек в книге, а метод `list` выводит краткий список всего содержимого книги. В программе 15.10 показан файл секции `implementation` для класса `AddressBook`.

**Программа 15.10. Addressbook.m Implementation File**

```
#import "AddressBook.h"

@implementation AddressBook;

// задание имени адресной книги и пустой книги

-(id) initWithName: (NSString *) name
{
 self = [super init];
 if (self) {
 bookName = [[NSString alloc] initWithString: name];
 book = [[NSMutableArray alloc] init];
 }
 return self;
}

-(void) addCard: (AddressCard *) theCard
{
 [book addObject: theCard];
}

-(int) entries
{
 return [book count];
}

-(void) list
{
 NSLog (@"===== Contents of: %@ =====", bookName);
 for (AddressCard *theCard in book)
 NSLog (@"%-20s %-32s", [theCard.name UTF8String],
 [theCard.email UTF8String]);
 NSLog (@"===== ===== ===== ===== =====");
}

-(void) dealloc
{
 [bookName release];
 [book release];
 [super dealloc];
}
@end
```

Метод `initWithName:` сначала вызывает метод `init` для суперкласса, чтобы выполнить его инициализацию. Затем он создает строковый объект (с использованием `alloc`, то есть он владеет им) и присваивает ему имя адресной книги, переданное как `name`. После этого выполняется выделение памяти и инициализация пустого мутабельного массива, который сохраняется в переменной экземпляра `book`.

Как мы определили, метод `initWithName:` возвращает объект типа `id`, а не объект `AddressBook`. Если класс `AddressBook` имеет подкласс, то аргумент для `initWithName:` не является объектом `AddressBook`; его тип определяется подклассом. Поэтому мы определяем тип возвращаемого объекта как обобщенный тип объекта.

Отметим также, что в `initWithName:` мы получаем владение переменными экземпляра `bookName` и `book` с помощью `alloc`. Например, если создать массив для `book` с помощью метода `NSMutableArray array`, как в

```
book = [NSMutableArray array];
```

вы все же не будете владельцем массива `book`; им будет владеть `NSMutableArray`. Таким образом, вы не сможете освободить его память, когда будете освобождать память для объекта `AddressBook`.

Метод `addCard:` принимает в качестве аргумента объект `AddressCard` и добавляет его в адресную книгу.

Метод `count` выдает число элементов массива. Метод `entries` использует это значение, возвращая число адресных карточек, хранящихся в адресной книге.

## Быстрое перечисление

В цикле `for` метода `list` показана новая конструкция.

```
for (AddressCard *theCard in book)
 NSLog(@"%@", [theCard.name UTF8String],
 [theCard.email UTF8String]);
```

Здесь применяется метод под названием *быстрое перечисление (fast enumeration)*, который осуществляет перебор всех элементов массива `book`. У него простой синтаксис: определяется переменная, которая содержит по очереди каждый элемент массива (`AddressCard *theCard`). После ключевого слова `in` указывается имя массива. При выполнении цикла `for` указанной переменной сначала присваивается первый элемент массива, а затем выполняется тело цикла. На следующем шаге цикла этой переменной присваивается второй элемент массива и выполняется тело цикла. Цикл выполняется до тех пор, пока переменной не будут присвоены все элементы массива (каждый раз с выполнением тела цикла).

Если бы мы определили ранее `theCard` как объект `AddressCard`, то цикл `for` выглядел бы проще:

```
for (theCard in book)
 ...
```

Ниже приводится тестовая программа 15.10 для проверки нового класса AddressBook.

#### Программа 15.10. Тестовая программа

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

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

 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";
 NSString *cName = @"Stephen Kochan";
 NSString *cEmail = @"steve@kochan-wood.com";
 NSString *dName = @"Jamie Baker";
 NSString *dEmail = @"jbaker@kochan-wood.com";

 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];
 AddressCard *card3 = [[AddressCard alloc] init];
 AddressCard *card4 = [[AddressCard alloc] init];

 AddressBook *myBook = [AddressBook alloc];

 // Сначала задаем четыре адресные карточки

 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];
 [card3 setName: cName andEmail: cEmail];
 [card4 setName: dName andEmail: dEmail];

 // Теперь инициализируем адресную книгу

 myBook = [myBook initWithName: @"Linda's Address Book"];

 NSLog(@"Entries in address book after creation: %i",
 [myBook entries]);

 // Добавляем несколько карточек в адресную книгу

 [myBook addCard: card1];
 [myBook addCard: card2];
 [myBook addCard: card3];
 [myBook addCard: card4];
```

```

 NSLog(@"Entries in address book after adding cards: %i",
 [myBook entries]);

// Теперь вывод всех записей адресной книги

[myBook list];

[card1 release];
[card2 release];
[card3 release];
[card4 release];
[myBook release];
[pool drain];
return 0;
}

```

**Вывод программы 15.10**

Entries in address book after creation: 0 (Записей в адресной книге после создания)  
Entries in address book after adding cards: 4 (Записей в адресной книге после добавления карточек)

```

===== Contents of: Linda's Address Book ===== (Содержимое книги:)
Julia Kochan jewls337@axlc.com
Tony Iannino tony.iannino@techfitness.com
Stephen Kochan steve@kochan-wood.com
Jamie Baker jbaker@kochan-wood.com
=====

```

В программе задаются четыре адресные карточки и создается новая адресная книга с именем Linda's Address Book. Эти четыре карточки добавляются затем в адресную книгу с помощью метода addCard:, после чего метод list выводит и проверяет содержимое адресной книги.

**Поиск в адресной книге**

Если адресная книга большая, то вы не будете выводить все ее содержимое каждый раз, чтобы найти конкретного человека. Добавим для этого соответствующий метод. Назовем этот метод lookup: (поиск); он будет принимать в качестве аргумента имя, которое нужно найти. Этот метод будет выполнять поиск соответствия в адресной книге (без учета регистра букв) и возвращать соответствующую запись, если она найдена. Если указанного имени нет в адресной книге, возвращается nil.

Ниже приводится метод lookup:.

```

// поиск адресной карточки по имени - требуется точное совпадение

-(AddressCard *) lookup: (NSString *) theName
{

```

```
for (AddressCard *nextCard in book)
 if ([[nextCard name] caseInsensitiveCompare: theName] == NSOrderedSame)
 return nextCard;
 return nil;
}
```

Поместим объявление этого метода в файл секции `interface`, а его определение – в файл секции `implementation` и напишем тестовую программу для опробования этого метода. Это программа 15.11 и ее вывод.

### Программа 15.11. Тестовая программа

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

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

 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";
 NSString *cName = @"Stephen Kochan";
 NSString *cEmail = @"steve@kochan-wood.com";
 NSString *dName = @"Jamie Baker";
 NSString *dEmail = @"jbaker@kochan-wood.com";
 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];
 AddressCard *card3 = [[AddressCard alloc] init];
 AddressCard *card4 = [[AddressCard alloc] init];

 AddressBook *myBook = [AddressBook alloc];
 AddressCard *myCard;

 // Сначала задаем четыре адресные карточки
 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];
 [card3 setName: cName andEmail: cEmail];
 [card4 setName: dName andEmail: dEmail];

 myBook = [myBook initWithName: @"Linda's Address Book"];

 // Добавляем несколько карточек в адресную книгу
 [myBook addCard: card1];
 [myBook addCard: card2];
```

```
[myBook addCard: card3];
[myBook addCard: card4];

// Поиск человека по имени

NSLog(@"%@", @"Lookup: Stephen Kochan");
myCard = [myBook lookup: @"stephen kochan"];
if (myCard != nil)
 [myCard print];
else
 NSLog(@"Not found!");

// Еще одна попытка поиска

NSLog(@"%@", @"Lookup: Haibo Zhang");
myCard = [myBook lookup: @"Haibo Zhang"];

if (myCard != nil)
 [myCard print];
else
 NSLog(@"Not found!");

[card1 release];
[card2 release];
[card3 release];
[card4 release];
[myBook release];

[pool drain];
return 0;
}
```

### Вывод программы 15.11

Lookup: Stephen Kochan (Найти:)

```
=====
| Stephen Kochan
| steve@kochan-wood.com
|
| 0 0
=====
```

Lookup: Haibo Zhang (Поиск:)

Not found! (Не найден!)

Когда метод `lookup:` нашел в адресной книге имя `Stephen Kochan` (совпадение без учета регистра букв), этот метод передал результирующую адресную карточку методу `AddressCard print` для ее вывода. При второй попытке поиска имя `Haibo Zhang` не было найдено.

Этот слишком простой метод поиска, поскольку он требует точного совпадения всего имени. Более подходящим был бы поиск частичного соответствия, обрабатывающий несколько соответствий. Например, выражение с сообщением

```
[myBook lookup: @"steve"]
```

позволило бы выбрать записи “`Steve Kochan`”, `Fred Stevens`” и “`steven levy`”. Поскольку может быть обнаружено несколько соответствий, имеет смысл создать массив, содержащий все соответствия, и возвращать этот массив вызывающему методу (см. упражнение 2 в конце главы), например,

```
matches = [myBook lookup: @"steve"];
```

### Удаление записи из адресной книги

Никакой диспетчер адресных книг, позволяющий добавлять записи, не будет полным без возможности удалять записи. Вы можете создать метод `removeCard:` для удаления конкретной адресной карточки (объекта `AddressCard`) из адресной книги, или метод `remove:`, который удаляет конкретного человека по его имени (см. упражнение 6 в конце главы).

Поскольку в файл секции `interface` внесены некоторые изменения, мы снова показываем его в программе 15.12 с новым методом `removeCard:`.

### Программа 15.12. Файл секции `interface Addressbook.h`

```
#import <Foundation/NSArray.h>
#import "AddressCard.h"

@interface AddressBook: NSObject
{
 NSString *bookName;
 NSMutableArray *book;
}

-(AddressBook *) initWithName: (NSString *) name;

-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;

-(AddressCard *) lookup: (NSString *) theName;
-(int) entries;
-(void) list;

@end
```

Ниже приводится новый метод removeCard.

```
-{void} removeCard: (AddressCard *) theCard
{
 [book removeObjectIdenticalTo: theCard];
}
```

Для *идентичных* объектов мы применяем одно местоположение в памяти. В методе removeObjectIdenticalTo: не считаются идентичными две адресные карточки, которые содержат одинаковую информацию, но находятся в разных местах памяти (что может, например, произойти, если создать копию объекта AddressCard).

Кстати, метод removeObjectIdenticalTo: удаляет все объекты, идентичные его аргументу. Но это важно только в том случае, если в массиве содержится несколько экземпляров одного и того же объекта.

Можно усложнить подход к равенству объектов, применяя метод removeObject: и написав метод isEqual: для проверки того, что два объекта равны. Если мы используем removeObject:, система автоматически вызывает метод isEqual: для каждого элемента массива, передавая ему два элемента для сравнения. В данном случае адресная книга содержит в качестве своих элементов объекты AddressCard, поэтому в этот класс необходимо добавить метод isEqual: (с замещением метода, который наследуется этим классом из NSObject). В самом методе нужно решить, как определяется равенство. Имеет смысл сравнивать соответствующие имена (name) и адреса электронной почты (email). Если обе пары равны, метод может возвращать значение YES; в противном случае он может возвращать значение NO. Этот метод может иметь следующий вид.

```
-(BOOL) isEqual: (AddressCard *) theCard
{
 if ([name isEqualToString: theCard.name] == YES &&
 [email isEqualToString: theCard.email] == YES)
 return YES;
 else
 return NO;
}
```

Отметим, что другие методы класса NSArray, такие как containsObject: и indexOfObject:, тоже основываются на стратегии isEqual: при проверке на равенство двух объектов.

Новый метод removeCard: тестируется в программе 15.12.

#### Программа 15.12. Тестовая программа

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>
```

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

 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";
 NSString *cName = @"Stephen Kochan";
 NSString *cEmail = @"steve@kochan-wood.com";
 NSString *dName = @"Jamie Baker";
 NSString *dEmail = @"jbaker@kochan-wood.com";

 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];
 AddressCard *card3 = [[AddressCard alloc] init];
 AddressCard *card4 = [[AddressCard alloc] init];

 AddressBook *myBook = [AddressBook alloc];
 AddressCard *myCard

 // Сначала создаем четыре адресные карточки

 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];
 [card3 setName: cName andEmail: cEmail];
 [card4 setName: dName andEmail: dEmail];

 myBook = [myBook initWithName: @"Linda's Address Book"];

 // Добавляем несколько карточек в адресную книгу

 [myBook addCard: card1];
 [myBook addCard: card2];
 [myBook addCard: card3];
 [myBook addCard: card4];

 // Поиск человека по имени

 NSLog(@"Lookup: Stephen Kochan");
 myCard = [myBook lookup: @"Stephen Kochan"];

 if (myCard != nil)
 [myCard print];
 else
 NSLog(@"Not found!");
```

```
// Теперь удаление записи из адресной книги

[myBook removeCard: myCard];
[myBook list]; // проверка, что ее больше нет

[card1 release];
[card2 release];
[card3 release];
[card4 release];
[myBook release];
[pool drain];

return 0;
}
```

### Вывод программы 15.12

Lookup: Stephen Kochan (Поиск:)

```
=====
| Stephen Kochan
| steve@kochan-wood.com
|
| 0 0
=====
```

===== Contents of: Linda's Address Book ===== (Содержимое книги)

Julia Kochan jewls337@axlc.com  
Tony Iannino tony.iannino@techfitness.com  
Jamie Baker jbaker@kochan-wood.com

После того, как запись Stephen Kochan в адресной книге найдена, мы передаем результирующий объект AddressCard новому методу `removeCard:` для удаления. Вывод списка адресной книги подтверждает, что удаление было сделано.

## Сортировка массивов

Если адресная книга содержит много записей, ее удобно упорядочить в алфавитном порядке. Добавим метод `sort` в класс `AddressBook` и применим метод `sortUsingSelector:` класса `NSMutableArray`. В этом методе в качестве аргумента служит селектор, применяемый методом `sortUsingSelector:` для сравнения двух элементов. Массивы могут содержать объекты любого типа, поэтому единственный способ реализации обобщенного метода сортировки – это проверка порядка элементов массива. Для этого необходимо добавить метод, выполняющий сравне-

ние двух элементов массива.<sup>4</sup> Результат, возвращаемый этим методом, должен иметь тип `NSComparisonResult`. Метод должен возвращать значение `NSOrderedAscending` (по возрастанию), если нужно, чтобы метод помещал первый элемент перед вторым элементом в массиве; значение `NSOrderedSame`, если два элемента равны; значение `NSOrderedDescending` (по убыванию), если первый элемент должен следовать после второго элемента.

Сначала приводим новый метод сортировки из класса `AddressBook`.

```
- (void) sort
{
 [book sortUsingSelector: @selector(compareNames:)];
}
```

Как известно из главы 9, выражение

```
@selector (compareNames:)
```

создает селектор с типом `SEL` из имени указанного метода; это метод, который используется `sortUsingSelector:` для сравнения двух элементов массива. Когда требуется выполнить такое сравнение, он вызывает указанный метод, отправляя сообщение первому элементу массива (получателю) для сравнения с аргументом. Возвращаемое значение должно иметь тип `NSComparisonResult`.

Поскольку элементы нашей адресной книги – это объекты класса `AddressCard`, метод сравнения должен быть добавлен в класс `AddressCard`. Мы должны вернуться к нашему классу `AddressCard` и добавить в него метод `compareNames:`.

```
// Сравнение двух имен из указанных адресных карточек
-(NSComparisonResult) compareNames: (id) element
{
 return [name compare: [element name]];
}
```

Для строкового сравнения двух имен из адресной книги можно использовать метод `NSString compare::`

Если добавить метод `sort` в класс `AddressBook` и метод `compareNames:` в класс `AddressCard`, то мы получим тестовую программу 15.13.

---

<sup>4</sup> Метод `sortUsingFunction:context:` позволяет использовать функцию вместо метода для выполнения сравнения.

**Программа 15.13.** Тестовая программа

```
#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

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

 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";
 NSString *cName = @"Stephen Kochan";
 NSString *cEmail = @"steve@kochan-wood.com";
 NSString *dName = @"Jamie Baker";
 NSString *dEmail = @"jbaker@kochan-wood.com";

 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];
 AddressCard *card3 = [[AddressCard alloc] init];
 AddressCard *card4 = [[AddressCard alloc] init];

 AddressBook *myBook = [AddressBook alloc];

 // Сначала задаем четыре адресные карточки

 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];
 [card3 setName: cName andEmail: cEmail];
 [card4 setName: dName andEmail: dEmail];

 myBook = [myBook initWithName: @"Linda's Address Book"];

 // Добавляем несколько карточек в адресную книгу

 [myBook addCard: card1];
 [myBook addCard: card2];
 [myBook addCard: card3];
 [myBook addCard: card4];

 // Вывод неотсортированной книги

 [myBook list];

 // Ее сортировка и повторный вывод
```

```
[myBook sort];
[myBook list];

[card1 release];
[card2 release];
[card3 release];
[card4 release];
[myBook release];
[pool drain];
return 0;
}
```

### Вывод программы 15.13

```
===== Contents of: Linda's Address Book =====
```

|                |                              |
|----------------|------------------------------|
| Julia Kochan   | jewls337@axlc.com            |
| Tony Iannino   | tony.iannino@techfitness.com |
| Stephen Kochan | steve@kochan-wood.com        |
| Jamie Baker    | jbaker@kochan-wood.com       |

```
=====
```

```
===== Contents of: Linda's Address Book =====
```

|                |                              |
|----------------|------------------------------|
| Jamie Baker    | jbaker@kochan-wood.com       |
| Julia Kochan   | jewls337@axlc.com            |
| Stephen Kochan | steve@kochan-wood.com        |
| Tony Iannino   | tony.iannino@techfitness.com |

```
=====
```

Отметим, что сортировка выполняется в порядке возрастания. Вы можете выполнить сортировку в порядке убывания, внеся изменения в метод compareNames: класса AddressCard, обратив смысл возвращаемых значений.

Для работы с объектами-массивами имеется более 50 методов. В таблицах 15.4 и 15.5 приводится список наиболее распространенных методов для работы с немутабельными и мутабельными массивами. NSMutableArray наследует методы класса NSArray, поскольку является его подклассом.

В таблицах 15.4 и 15.5 *obj*, *obj1* и *obj2* являются произвольными объектами; *i* – это значение типа NSInteger, представляющее допустимый номер элемента в массиве, *selector* – это объект-селектор типа SEL, *size* имеет тип NSUInteger.

**Табл. 15.4.** Наиболее распространенные методы класса NSArray

| Метод                                                                  | Описание                                                                                                                        |
|------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| + (id) arrayWithObjects:<br><i>obj1, obj2, ... nil</i>                 | Создает новый массив с элементами <i>obj1, obj2, ...</i>                                                                        |
| - (BOOL) containsObject: <i>obj</i>                                    | Определяет, содержится ли <i>obj</i> в массиве (используется метод isEqual:).                                                   |
| - (NSUInteger) count                                                   | Указывает число элементов в массиве.                                                                                            |
| - (NSUInteger) indexOfObject: <i>obj</i>                               | Определяет номер первого элемента, содержащего <i>obj</i> (используется метод isEqual:).                                        |
| - (id) objectAtIndex: <i>i</i>                                         | Указывает объект, хранящийся в элементе <i>i</i> .                                                                              |
| - (void) makeObjectsPerform<br>Selector: (SEL) <i>selector</i>         | Передает каждому элементу массива сообщение, которое указывает <i>selector</i> .                                                |
| - (NSArray *) sortedArrayUsing<br>Selector: (SEL) <i>selector</i>      | Сортирует массив в соответствии с методом сравнения, который указывает <i>selector</i> .                                        |
| - (BOOL) writeToFile: <i>path</i><br>automatically: (BOOL) <i>flag</i> | Записывает массив в указанный с помощью <i>path</i> файл, создавая сначала временный файл, если <i>flag</i> имеет значение YES. |

**Табл. 15.5.** Наиболее распространенные методы класса NSMutableArray

| Метод                                                             | Описание                                                                                 |
|-------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| + (id) array                                                      | Создает пустой массив.                                                                   |
| + (id) arrayWithCapacity: <i>size</i>                             | Создает массив с указанным начальным размером.                                           |
| - (id) initWithCapacity: <i>size</i>                              | Инициализирует новый выделенный (alloc) массив с указанным начальным размером.           |
| - (void) addObject: <i>obj</i>                                    | Добавляет <i>obj</i> в конец массива.                                                    |
| - (void) insertObject: <i>obj</i> atIndex: <i>i</i>               | Выполняет вставку <i>obj</i> в элемент <i>i</i> массива.                                 |
| - (void) replaceObjectAtIndex:<br><i>i</i> withObject: <i>obj</i> | Заменяет объект в элементе <i>i</i> массива на <i>obj</i> .                              |
| - (void) removeObject: <i>obj</i>                                 | Удаляет все экземпляры <i>obj</i> из массива.                                            |
| - (void) removeObjectAtIndex: <i>i</i>                            | Удаляет элемент <i>i</i> из массива, смешая влево все элементы, начиная с <i>i+1</i> .   |
| - (void) sortUsingSelector: (SEL)<br><i>selector</i>              | Сортирует массив в соответствии с методом сравнения, который указывает <i>selector</i> . |

## Объекты-словари

*Словарь* (*dictionary*) – это коллекция данных, состоящая из пар ключ-объект. Как в обычном словаре, мы получаем из словаря Objective-C значение (объект) по его ключу. Ключи в словаре должны быть уникальными, и они могут быть объектом любого типа, хотя обычно это строки. Значение, соответствующее ключу, тоже может быть объектом любого типа, но не должно быть значение *nil*.

Словари могут быть мутабельными или немутабельными; в первом случае в них можно динамически добавлять и удалять записи. В словарях можно выполнять поиск по определенному ключу, их содержимое можно делать перечислимым. В программе 15.14 создается словарь терминов Objective-C, и в нем заполняются первые три записи.

Для использования словарей нужно включить следующую строку:

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

### Программа 15.14

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>

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

 NSMutableDictionary *glossary = [NSMutableDictionary dictionary];

 // Сохранение трех записей в этом словаре

 [glossary setObject: @"A class defined so other classes can inherit from it"
 forKey: @"abstract class"];
 [glossary setObject: @"To implement all the methods defined in a protocol"
 forKey: @"adopt"];
 [glossary setObject: @"Storing an object for later use"
 forKey: @"archiving"];

 // Их считывание и вывод

 NSLog(@"abstract class: %@", [glossary objectForKey: @"abstract class"]);
 NSLog(@"adopt: %@", [glossary objectForKey: @"adopt"]);
 NSLog(@"archiving: %@", [glossary objectForKey: @"archiving"]);

 [pool drain];
 return 0;
}
```

### Вывод программы 15.14

abstract class: A class defined so other classes can inherit from it  
(абстрактный класс: класс, определенный таким образом, чтобы другие классы могли наследовать из него)  
adopt: To implement all the methods defined in a protocol  
(принять: для реализации всех методов, определенных в протоколе)  
archiving: Storing an object for later use  
(архивация: сохранение объекта для дальнейшего использования)

С помощью выражения

[NSMutableDictionary dictionary]

создается пустой мутабельный словарь. Мы можем добавлять в этот словарь пары ключ-значение с помощью метода  `setObject:forKey:`. После создания словаря мы можем считывать значение для заданного ключа с помощью метода  `objectForKey:`. В программе 15.14 показано считывание и вывод этих трех записей. В более близком к практике приложении пользователь вводит слово, и программа выполняет поиск определения этого слова в словаре.

## Перечисление записей словаря

В программе 15.15 показано, как можно определить словарь с начальными параметрами «ключ-значение» с помощью метода  `dictionaryWithObjectsAndKeys:`. Мы создадим немутабельный словарь и покажем, как применяется цикл с быстрым перечислением для считывания каждого элемента из словаря, по одному ключу за шаг. В отличие от объектов-массивов, объекты-словари не упорядочиваются, поэтому первая пара ключ-объект, помещенная в словарь, не обязательно будет первым ключом, извлекаемым при перечислении.

### Программа 15.15

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>

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

 NSDictionary *glossary =
 [NSDictionary dictionaryWithObjectsAndKeys:
 @{@"A class defined so other classes can inherit from it": @"abstract class",
 @"To implement all the methods defined in a protocol": @"adopt",
 @"Storing an object for later use": @"archiving",
 @"archiving": @"archiving"}];
```

```

 nil
];
// Вывод всех пар ключ-значение из словаря

for (NSString *key in glossary)
 NSLog(@"%@", key, [glossary objectForKey: key]);

[pool drain];
return 0;
}

```

### Вывод программы 15.15

abstract class: A class defined so other classes can inherit from it

adopt: To implement all the methods defined in a protocol

archiving: Storing an object for later use

Аргументом для метода `dictionaryWithObjectsAndKeys:` является список пар объект-ключ (именно в этом порядке), разделяемых запятой. Этот список должен заканчиваться специальным объектом `nil`.

После создания словаря в цикле перечисляется его содержимое. Как уже говорилось, ключичитываются из словаря по очереди, без специального порядка. Если требуется вывести содержимое словаря в алфавитном порядке, можно прочитать все ключи из словаря, отсортировать их и затем считывать по порядку все значения для отсортированных ключей. Половину этой работы выполняет для вас метод `keysSortedByValueUsingSelector:`, возвращая отсортированные ключи в виде массива в соответствии с критериями сортировки.

Мы только что показали некоторые базовые операции со словарями. В таблицах 15.6 и 15.7 приводятся наиболее распространенные методы для работы с немутабельными и мутабельными словарями. Поскольку `NSTMutableDictionary` является подмножеством `NSDictionary`, он наследует его методы.

В таблицах 15.6 и 15.7 ключи и объекты `key`, `key1`, `key2`, `obj`, `obj1` и `obj2` – произвольные объекты, и `size` – целое без знака (`unsigned int`) типа `NSUInteger`.

**Табл. 15.6. Наиболее распространенные методы класса `NSDictionary`**

| Метод                                             | Описание                                                                                                                                          |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>+ (id) dictionaryWithObjectsAndKeys:</code> | Создает словарь с парами ключ-объект { <code>key1, obj1</code> , { <code>key1, obj1</code> }, { <code>key2, obj2</code> }, ... }                  |
| <code>- (id) initWithObjectsAndKeys:</code>       | Инициализирует новый выделенный ( <code>alloc</code> ) словарь с парами ключ-объект { <code>key1, obj1</code> }, { <code>key2, obj2</code> }, ... |
| <code>- (unsigned int) count</code>               | Возвращает число записей, содержащихся в словаре.                                                                                                 |
| <code>- (NSEnumerator *) keyEnumerator</code>     | Возвращает объект класса <code>NSEnumerator</code> для всех ключей в словаре.                                                                     |

**Табл. 15.6.** Наиболее распространенные методы класса NSDictionary (окончание)

| Метод                                                             | Описание                                                                                                                              |
|-------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| -NSArray *(<br>keysSortedByValueUsingSelector:<br>(SEL) selector) | Возвращает массив ключей из словаря, отсортированных в соответствии с методом сравнения, который указывает селектор <i>selector</i> . |
| -NSEnumerator *(<br>objectEnumerator)                             | Возвращает объект класса NSEnumerator для всех значений из словаря.                                                                   |
| -id objectForKey: key                                             | Возвращает объект для указанного ключа <i>key</i> .                                                                                   |

**Табл. 15.7.** Наиболее распространенные методы класса NSMutableDictionary

| Метод                              | Описание                                                                                                |
|------------------------------------|---------------------------------------------------------------------------------------------------------|
| +(id) dictionaryWithCapacity: size | Создает мутабельный словарь с указанным начальным размером <i>size</i> .                                |
| -(id) initWithCapacity: size       | Инициализирует новый выделенный (alloc) словарь с указанным начальным размером <i>size</i> .            |
| -(void) removeAllObjects           | Удаляет все записи из словаря.                                                                          |
| -(void) removeObjectForKey: key    | Удаляет из словаря запись с указанным ключом <i>key</i> .                                               |
| -(void) setObject: obj forKey: key | Добавляет в словарь <i>obj</i> для ключа <i>key</i> и заменяет значение, если этот ключ уже существует. |

## Объекты-наборы

*Набор, или множество (set)* – это коллекция уникальных объектов. Набор может быть мутабельным или немутабельным. Для наборов можно выполнять операции поиска, добавления и удаления членов (мутабельные наборы), сравнения, поиск пересечения (*intersect*) и объединения (*union*).

Для работы с наборами в программе нужно включить следующую строку.

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

В программе 15.16 показаны основные операции с наборами. Предположим, что нам нужно выводить содержимое наборов во время выполнения программы. Создаем новый метод с именем print и добавляем метод print в класс NSSet, создавая новую категорию с именем Printing. NSMutableSet — это подкласс NSSet, поэтому мутабельные наборы тоже могут использовать новый метод print.

### Программа 15.16

```
#import <Foundation/NSObject.h>
#import <Foundation/NSSet.h>
#import <Foundation/NSValue.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>

// Создание объекта целого типа
#define INTOBJ(v) [NSNumber numberWithInteger: v]

// Добавление в NSSet метода print с помощью категории Printing

@interface NSSet (Printing)
-(void) print;
@end

@implementation NSSet (Printing)
-(void) print {
 printf ("{");

 for (NSNumber *element in self)
 printf (" %li ", (long) [element integerValue]);

 printf ("}\n");
}
@end

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

 NSMutableSet *set1 = [NSMutableSet setWithObjects:
 INTOBJ(1), INTOBJ(3), INTOBJ(5), INTOBJ(10), nil];
 NSSet *set2 = [NSSet setWithObjects:
 INTOBJ(-5), INTOBJ(100), INTOBJ(3), INTOBJ(5), nil];
 NSSet *set3 = [NSSet setWithObjects:
 INTOBJ(12), INTOBJ(200), INTOBJ(3), nil];
}
```

```
NSLog(@"%@", set1);
[set1 print];
NSLog(@"%@", set2);
[set2 print];

// Проверка на равенство
if ([set1 isEqualToSet: set2] == YES)
 NSLog(@"set1 equals set2");
else
 NSLog(@"set1 is not equal to set2");

// Проверка членства в наборе

if ([set1 containsObject: INTOBJ(10)] == YES)
 NSLog(@"set1 contains 10");
else
 NSLog(@"set1 does not contain 10");

if ([set2 containsObject: INTOBJ(10)] == YES)
 NSLog(@"set2 contains 10");
else
 NSLog(@"set2 does not contain 10");

// Добавление и удаление объектов из мутабельного набора set1

[set1 addObject: INTOBJ(4)];
[set1 removeObject: INTOBJ(10)];
NSLog(@"set1 after adding 4 and removing 10: ");
[set1 print];

// Получение пересечения двух наборов

[set1 intersectSet: set2];
NSLog(@"set1 intersect set2: ");
[set1 print];

// Объединение двух наборов

[set1 unionSet: set3];
NSLog(@"set1 union set3: ");
[set1 print];

[pool drain];
return 0;
}
```

### Выход программы 15.16

```
set1: (набор 1)
{ 3 10 1 5 }
set2: (набор 2)
{ 100 3 -5 5 }
set1 is not equal to set2 (set1 не равен набору set2)
set1 contains 10 (set1 содержит 10)
set2 does not contain 10 (set2 не содержит 10)
set1 after adding 4 and removing 10: (set1 после добавления 4 и удаления 10)
{ 3 1 5 4 }
set1 intersect set2: (пересечение set1 с set2)
{ 3 5 }
set1 union set3: (объединение set1 с set3)
{ 12 3 5 200 }
```

В методе `print` используется описанный ранее метод быстрого перечисления для считывания каждого элемента из набора и определяется макрос с именем `INTOBJ` для создания объекта из целого значения. Это позволяет сделать программу короче и исключить необязательный ввод. Конечно, наш метод `print` не является достаточно обобщенным, поскольку он работает только с наборами, содержащими целые элементы. Но это хороший пример, напоминающий, как добавлять методы в класс с помощью категорий.<sup>5</sup> (Отметим, что в методе `print` используется процедура `printf` библиотеки C для вывода элементов каждого набора в одной строке.)

Метод `setWithObjects:` создает новый набор из списка объектов, заканчивающегося объектом `nil`. После создания трех наборов программа выводит первые два набора с помощью нового метода `print`. Затем метод `isEqualToString` проверяет равенство набора `set1` набору `set2` (они не равны).

Метод `containsObject:` проверяет сначала, содержится ли целый элемент 10 в наборе `set1`, и затем делает то же самое для набора `set2`. Булевые значения, возвращаемые этим методом, показывают, что данный элемент содержится в первом наборе и не содержится во втором.

Затем в программе используются методы  `addObject:` и  `removeObject:`, чтобы добавить 4 и удалить 10 из `set1`. Вывод содержимого этого набора показывает, что операции выполнены успешно.

Методы `intersect:` и `union:` используются, чтобы вычислять пересечение и объединение двух наборов. В обоих случаях результат операции заменяет получателя сообщения.

В Foundation framework имеется также класс `NSCountedSet`. Наборы могут содержать более одного экземпляра одного и того же объекта, однако вместо нескольких представлений этого объекта в наборе поддерживается счетчик экземпляров. При первом добавлении объекта в набор его счетчик равен 1. При

<sup>5</sup> В более обобщенном методе можно было бы вызывать метод описания каждого объекта для вывода каждого элемента набора. Это позволило бы выводить в удобочитаемом формате наборы, содержащие любые типы объектов. Вы можете выводить содержимое любой коллекции с помощью одного вызова  `NSLog`, используя символы формата «вывода объекта» `%@`.

последующем добавлении этого объекта в набор происходит наращивание его счетчика, а при удалении объекта счетчик уменьшается на 1. Когда счетчик становится равным 0, объект удаляется из набора. Метод `countForObject:` читает счетчик для указанного объекта в наборе.

Наборы со счетчиками могут применяться, например, в приложении для подсчета количества слов. При каждом обнаружении слова в некотором тексте его можно добавить в набор со счетчиками. По окончании просмотра текста можно читать слово из набора вместе с его счетчиком, показывающим, сколько раз данное слово встречается в этом тексте.

Мы только что показали некоторые основные операции с наборами. В таблицах 15.8 и 15.9 приводятся наиболее распространенные методы для работы с мутабельными и немутабельными наборами. Поскольку `NSMutableSet` является подклассом класса `NSSet`, он наследует его методы.

В таблицах 15.8 и 15.9 *obj*, *obj1* и *obj2* являются произвольными объектами, *nsset* – это объект класса `NSSet` или `NSMutableSet`, *size* – целый элемент типа `NSUInteger`.

**Табл. 15.8.** Наиболее распространенные методы класса `NSSet`

| Метод                                                                   | Описание                                                                                               |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>+ (id) setWithObjects: <i>obj1</i>, <i>obj2</i>, ..., nil</code>  | Создает новый набор из списка объектов.                                                                |
| <code>- (id) initWithObjects: <i>obj1</i>, <i>obj2</i>, ..., nil</code> | Инициализирует новый выделенный ( <code>alloc</code> ) набор со списком объектов.                      |
| <code>- (NSUInteger) count</code>                                       | Возвращает число членов данного набора.                                                                |
| <code>- (BOOL) containsObject: <i>obj</i></code>                        | Определяет, содержится ли <i>obj</i> в данном наборе.                                                  |
| <code>- (BOOL) member: <i>obj</i></code>                                | Определяет, содержится ли <i>obj</i> в данном наборе (с использованием метода <code>isEqual:</code> ). |
| <code>- (NSEnumerator *) objectEnumerator</code>                        | Возвращает объект класса <code>NSEnumerator</code> для всех объектов набора.                           |
| <code>- (BOOL) isSubsetOfSet: <i>nsset</i></code>                       | Определяет, содержит ли каждый член получателя в <i>nsset</i> .                                        |
| <code>- (BOOL) intersectsSet: <i>nsset</i></code>                       | Определяет, содержит ли хотя бы один член получателя в <i>nsset</i> .                                  |
| <code>- (BOOL) isEqualToSet: <i>nsset</i></code>                        | Проверяет равенство двух наборов.                                                                      |

**Табл. 15.9.** Наиболее распространенные методы класса `NSMutableSet`

| Метод                                             | Описание                                                                                          |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>- (id) setWithCapacity: <i>size</i></code>  | Создает новый набор с начальной емкостью для хранения <i>size</i> членов.                         |
| <code>- (id) initWithCapacity: <i>size</i></code> | Задает начальную емкость нового выделенного ( <code>alloc</code> ) набора для <i>size</i> членов. |
| <code>- (void) addObject: <i>obj</i></code>       | Добавляет <i>obj</i> в набор.                                                                     |

Табл. 15.9. Наиболее распространенные методы класса NSMutableSet (окончание)

| Метод                               | Описание                                                               |
|-------------------------------------|------------------------------------------------------------------------|
| - (void) removeObject: <i>obj</i>   | Удаляет <i>obj</i> из набора.                                          |
| - (void) removeAllObjects           | Удаляет всех членов набора-получателя.                                 |
| - (void) unionSet: <i>nsset</i>     | Добавляет каждого члена <i>nsset</i> в набор-получатель.               |
| - (void) minusSet: <i>nsset</i>     | Удаляет всех членов <i>nsset</i> из набора-получателя.                 |
| - (void) intersectSet: <i>nsset</i> | Удаляет из набора-получателя всех членов, не входящих в <i>nsset</i> . |

## Упражнения

1. Найдите класс `NSDateComponents` в своей документации. Добавьте в `NSDateComponents` новую категорию с именем `ElapsedDays`. В этой категории добавьте метод в соответствии со следующим объявлением этого метода.

```
- (unsigned long) number_ofElapsedDays: (NSDateComponents *) theDate;
```

Этот метод должен возвращать число дней (*elapsed days*), прошедших между датой получателя и датой аргумента. Напишите тестовую программу для проверки этого метода. (Подсказка: посмотрите метод `years:months:days:hours:minutes:seconds:sinceDate:..`)

2. Внесите изменения в метод `lookup:`, разработанный в этой главе для класса `AddressBook`, чтобы можно было проверять частичное совпадение с именем. Выражение с сообщением `[myBook lookup: @"»steve"]` должно определять соответствие записи, содержащей строку `steve` в любой части имени.
3. Внесите изменения в метод `lookup:`, разработанный в этой главе для класса `AddressBook`, чтобы можно было искать все соответствия в адресной книге. Этот метод должен возвращать массив, содержащий все соответствующие адресные карточки, или `nil`, если не найдено ни одного соответствия.
4. Добавьте новые поля по вашему выбору в класс `AddressCard`. Например, вы можете разделить поле `name` на поля имени и фамилии, а также добавить адрес (с полями штата, города, почтового кода и страны) и номер телефона. Напишите метод-установщик и метод-получатель, а также проследите, чтобы методы `print` и `list` правильно выводили поля.
5. После завершения упражнения 3 внесите изменения в метод `lookup:` из упражнения 2, чтобы выполнять поиск по всем полям адресной карточки. Как вы спроектировали бы свои классы `AddressCard` и `AddressBook`, чтобы в `AddressBook` не нужно было знать все поля, хранящиеся в `AddressCard`?
6. Добавьте метод `removeName:` в класс `AddressBook`, чтобы удалять запись адресной книги в соответствии со следующим объявлением этого метода.

```
- (BOOL) removeName: (NSString *) theName;
```

Используйте метод `lookup:`, разработанный в упражнении 2. Если имя не найдено или существует несколько записей, метод должен возвращать значение `NO`. Если запись успешно удалена, метод должен возвращать значение `YES`.

7. Используя класс `Fraction`, определенный в части I, создайте массив дробей (`fraction`) с некоторыми произвольными значениями. Затем напишите код для вычисления суммы всех дробей, хранящихся в этом массиве.
8. Используя класс `Fraction`, определенный в части I, создайте мутабельный массив дробей (`fraction`) с произвольными значениями. Затем отсортируйте этот массив с помощью метода `sortUsingSelector:` из класса `NSMutableArray`. Добавьте в класс `Fraction` категорию `Comparison` (сравнение) и реализуйте свой метод сравнения в этой категории.
9. Определите три новых класса с именами `Song`, `PlayList` и `MusicCollection`. Объект класса `Song` должен содержать информацию об определенной песне, например, ее название (`title`), исполнителя (`artist`), альбом (`album`) и время воспроизведения (`playing time`). Объект класса `PlayList` должен содержать имя списка воспроизведения и коллекцию песен. Объект класса `MusicCollection` должен содержать коллекцию списков воспроизведения, включая специальный мастер-список с именем `library` (библиотека), который содержит все песни этой коллекции. Определите эти три класса и напишите методы, чтобы выполнять следующее.
  - Создавать объект класса `Song` и задавать его информацию.
  - Создавать объект класса `PlayList` и добавлять или удалять песни из списка воспроизведения. Новая песня должна добавляться в мастер-список, если ее еще там нет. При удалении песни из мастер-списка она должна удаляться из всех списков воспроизведения этой музыкальной коллекции.
  - Создавать объект класса `MusicCollection` и добавлять в него или удалять из него объекты класса `PlayList` (списки воспроизведения)
  - Выполнять поиск и вывод информации о любой песне, любом списке воспроизведения или всей музыкальной коллекции.
- Проследите, чтобы во всех классах не было утечки памяти!
10. Напишите программу, которая создает из массива объектов типа `NSInteger` гистограмму, где показано каждое целое значение вместе с числом экземпляров этого значения (частотой появления) в массиве. Для создания счетчиков экземпляров используйте объект `NSCountedSet`.

## Глава 16

# Работа с файлами

Foundation framework позволяет получать доступ к файловой системе для выполнения основных операций с файлами и папками (каталогами) с помощью `NSFileManager`, методы которого позволяют выполнять следующие операции.

- Создание нового файла.
- Чтение из существующего файла.
- Запись данных в файл.
- Переименование файла.
- Удаление файла.
- Проверка существования файла.
- Определение размера файла, а также других атрибутов.
- Создание копии файла.
- Проверка двух файлов на совпадение содержимого.

Многие из этих операций можно также выполнять с папками. Например, можно создать папку, прочитать ее содержимое или удалить ее. Еще одна возможность – это возможность *привязки (link)* файлов. Привязка означает, что один и тот же файл может существовать под двумя именами и даже в двух различных папках.

Чтобы открыть файл и выполнить с этим файлом несколько операций чтения-записи, используются методы из `NSFileHandle`. Методы этого класса позволяют следующее.

- Открывать файл для чтения, записи или изменения (`update` – чтение и запись).
- Искать указанное местоположение в файле.
- Считывать или записывать заданное число байтов из файла или в файл.

Методы из `NSFileHandle` можно также применять к устройствами (сокетам). В этой главе мы будем работать только с обычными файлами.

## Управление файлами и папками: NSFileManager

Файл или папка уникально указывается для `NSFileManager` с помощью *имени* пути доступа к файлу (`pathname`). Имя пути – это объект класса `NSString`, который может представлять относительное или полное имя пути. *Относительное имя* пути определяется относительно текущей папки. Например, имя файла `copy1.m` означает, что файл `copy1.m` находится в текущей папке. Символы «слэш» являются разделителями папок в указанном пути. Имя файла `ch16/copy1.m` тоже является относительным именем пути, указывая файл `copy1.m`, хранящийся в папке `ch16`, которая содержится в текущей папке.

Полные имена пути, которые также называют *абсолютными* именами пути, начинаются с ведущего слэша (/). Слэш на самом деле представляет папку, которая называется *корневой (root)* папкой. На моем Mac полное имя пути к моей домашней папке – `/Users/stevekochan`. Этот путь представляет три папки: / (корневая папка), `Users` и `stevekochan`.

Специальный символ «тильда» (~) используется как сокращенное представление домашней папки пользователя. Так, `~linda` – это сокращение для домашней папки пользователя `linda`, которая может быть представлена в виде пути `/Users/linda`. Отдельный символ «тильда» указывает домашнюю папку текущего пользователя, путь `~/copy1.m` означает ссылку на файл `copy1.m`, хранящийся в домашней папке текущего пользователя. Другие специальные символы для пути в стиле UNIX, такие как точка (.) для текущей папки и .. для родительской папки, следует удалять из имен пути при работе с файлами в методах Foundation. Для этого можно использовать разнообразные утилиты, которые будут рассматриваться ниже в этой главе.

Избегайте фиксированных путей в своих программах. Как будет описано далее, имеются функции и методы, которые позволяют получать имя пути для текущей папки, домашней папки пользователя и папки для создания временных файлов. Обращайтесь к ним, насколько это возможно. Ниже мы покажем, что Foundation содержит функцию для получения списка специальных папок, таких как папки пользователя `Documents`.

В таблице 16.1 приводится сводка основных методов `NSFileManager` для работы с файлами. В этой таблице `path` (путь), `path1`, `path2`, `from` (из) и `to` (куда) – объекты класса `NSString`, `attr` (атрибут) – объект `NSDictionary`, `handler` – обработчик (хендлер) обратного вызова, который вы можете предоставлять для обработки ошибок. Если указать `nil` для `handler`, то выполняется действие по умолчанию. Для методов, возвращающих значение типа `BOOL`, это YES при успешном завершении операции и NO, если операцию не удалось выполнить. В этой главе не говорится о том, как писать хендлеры.

**Табл. 16.1.** Наиболее распространенные файловые методы класса NSFileManager

| Метод                                                                                    | Описание                                                                        |
|------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| -NSData * contentsAtPath: <i>path</i>                                                    | Читает данные из файла.                                                         |
| -BOOL createFileAtPath: <i>path</i> contents: (BOOL) <i>data</i> attributes: <i>attr</i> | Пишет данные ( <i>data</i> ) в файл.                                            |
| -BOOL removeFileAtPath: <i>path</i> handler: <i>handler</i>                              | Удаляет файл.                                                                   |
| -BOOL movePath: <i>from</i> toPath: <i>to</i> handler: <i>handler</i>                    | Переименовывает или перемещает файл ( <i>to</i> не может существовать заранее). |
| -BOOL copyPath: <i>from</i> toPath: <i>to</i> handler: <i>handler</i>                    | Копирует файл ( <i>to</i> не может существовать заранее).                       |
| -BOOL contentsEqualAtPath: <i>path1</i> andPath: <i>path2</i>                            | Сравнивает содержимое двух файлов.                                              |
| -BOOL fileExistsAtPath: <i>path</i>                                                      | Проверяет существование файла.                                                  |
| -BOOL isReadableFileAtPath: <i>path</i>                                                  | Проверяет, существует ли файл и доступен ли он для чтения.                      |
| -BOOL isWritableFileAtPath: <i>path</i>                                                  | Проверяет, существует ли файл и доступен ли он для записи.                      |
| -NSDictionary * fileAttributesAtPath: <i>path</i> traverseLink: (BOOL) <i>flag</i>       | Читает атрибуты файла.                                                          |
| -BOOL changeFileAttributes: <i>attr</i> atPath: <i>path</i>                              | Изменяет атрибуты.                                                              |

Каждый из этих файловых методов вызывается в объекте NSFileManager, который создается при отправке сообщения defaultManager этому классу.

```
NSFileManager *fm;
...
fm = [NSFileManager defaultManager];
```

Например, для удаления файла todolist из текущей папки нужно создать сначала объект класса NSFileManager, как показано выше, и затем вызвать метод removeFileAtPath::

```
[fm removeFileAtPath: @"todolist" handler: nil];
```

Возвращаемый результат можно проверить, чтобы убедиться, что удаление этого файла выполнено успешно.

```
if ([fm removeFileAtPath: @"todolist" handler: nil] == NO) {
 NSLog(@"Couldn't remove file todolist"); (Нельзя удалить файл todolist)
 return 1;
}
```

Словарь атрибутов позволяет, в частности, указывать разрешения доступа к создаваемому файлу или получать либо изменять информацию для существующего файла. Разрешения по умолчанию задаются при создании файла с указанием значения `nil` в качестве этого параметра. Метод `fileAttributesAtPath:traverseLink:` возвращает словарь, содержащий атрибуты указанного файла. Параметр `traverseLink:` имеет значение `YES` или `NO` для символических ссылок. Если файл задан символьской ссылкой и указано значение `YES`, то возвращаются атрибуты файла привязки; если указано значение `NO`, то возвращаются атрибуты самой привязки (ссылки).

Для уже существующих файлов в словарь атрибутов включается такая информация, как владелец файла, размер файла, дата его создания, и т.д. Каждый атрибут можно извлекать из словаря по его ключу; все эти ключи определены в `<Foundation/NSFileManager.h>`. Например, `NSFileSize` – это ключ для атрибута размера файла.

В программе 16.1 показаны основные операции с файлами. В этом примере предполагается, что в текущей папке есть файл `testfile`, содержащий следующие три строки текста.

```
This is a test file with some data in it. (Это тестовый файл с некоторыми данными.)
Here's another line of data. (Это еще одна строка данных.)
And a third. (И третья.)
```

### Программа 16.1

```
// Основные файловые операции
// Предполагается, что существует файл "testfile"
// в текущей рабочей папке

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSDictionary.h>

int main (int argc, char *argv[]){
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *fName = @"testfile";
 NSFileManager *fm;
 NSDictionary *attr;

 // Нужно создать экземпляр filemanager

 fm = [NSFileManager defaultManager];

 // Сначала проверим существование нашего тестового файла

 if ([fm fileExistsAtPath: fName] == NO) {
```

```
NSLog(@"File doesn't exist!"); (Файл не существует)
return 1;
}

// Теперь создадим копию

if ([fm copyPath: fName toPath: @"newfile" handler: nil] == NO) {
 NSLog(@"File copy failed!"); (Копирование файла не выполнено)
 return 2;
}

// Проверим эти два файла на идентичность

if ([fm contentsEqualAtPath: fName andPath: @"newfile"] == NO) {
 NSLog(@"Files are not equal!"); (Файлы не равны)
 return 3;
}

// Теперь переименуем копию

if ([fm movePath: @"newfile" toPath: @"newfile2"
 handler: nil] == NO) {
 NSLog(@"File rename failed!"); (Переименование файла не выполнено)
 return 4;
}

// Получим размер newfile2

if ((attr = [fm fileAttributesAtPath: @"newfile2"
 traverseLink: NO]) == nil) {
 NSLog(@"Couldn't get file attributes!"); (Невозможно получить атрибуты файла)
 return 5;
}

NSLog(@"File size is %i bytes",
 [[attr objectForKey: NSFileSize] intValue]);

// И, наконец, удалим исходный файл

if ([fm removeFileAtPath: fName handler: nil] == NO) {
 NSLog(@"File removal failed!"); (невозможно удалить файл)
 return 6;
}

NSLog(@"All operations were successful!");

// Вывод содержимого нового созданного файла
```

```

 NSLog(@"%@", [NSString stringWithContentsOfFile: @"newfile2" encoding:
 NSUTF8StringEncoding error: nil]);

 [pool drain];
 return 0;
}

```

### Вывод программы 16.1

File size is 84 bytes (Размер файла 84 байта)  
All operations were successful! (Все операции выполнены успешно)

This is a test file with some data in it. (Это тестовый файл с некоторыми данными.)  
Here's another line of data. (Это еще одна строка данных.)  
And a third. (И третья.)

Программа сначала проверяет, существует ли файл `testfile`. Если да, то программа создает его копию и затем проверяет эти файлы на совпадение. Опытные пользователи UNIX обратят внимание, что мы не можем переместить или скопировать файл в определенную папку, просто указав эту целевую папку для методов `copyPath:toPath:` и `movePath:toPath:`; в этой папке должно быть явно указано имя файла.

---

**Примечание.** Мы можем создать `testfile` с помощью Xcode, выбрав `New File...` (создать файл) в меню `File`. В появившейся левой панели нужно выделить `Other` (Другое) и затем выбрать в правой панели `Empty File` (Пустой файл). Введите `testfile` как имя файла и убедитесь, что он создается в той же папке, что и выполняемый файл — в вашей папке проекта `Build/Debug`.

Метод `movePath:toPath:` можно использовать для перемещения файла из одной папки в другую. (Или для перемещения целой папки.) Если оба пути указывают на одну и ту же папку (как в нашем примере), то результатом будет переименование файла. Например, в программе 16.1 мы переименовываем файл `newfile` в `newfile2`.

Как указывалось в таблице 16.1, при выполнении операций копирования, переименования или перемещения указанный целевой файл (`to`) не может существовать заранее, иначе операция не будет выполнена.

Размер файла `newfile2` определяется с помощью метода `fileAttributesAtPath:traverseLink:`. Мы проверяем, что возвращается непустой (не `nil`) словарь, и затем используем метод `NSDictionary objectForKey:` для получения из словаря размера файла с помощью ключа `NSFileSize`. Затем выводится целое значение из словаря.

Метод `removeFileAtPath:handler:` удаляет наш исходный файл `testfile`.

И, наконец, метод `NSString stringWithContentsOfFile:` читает содержимое файла `newfile2` в строковый объект, который затем передается как аргумент в `NSLog` для вывода.

В программе 16.1 проверяется успешность выполнения каждой файловой операции. Если операция не выполнена, выводится сообщение об ошибке с помощью `NSLog`, и программа завершает работу, возвращая ненулевое значение

состояния выхода. Каждое ненулевое значение, которое соответствует ошибке в программе, уникальным образом определяет тип ошибки. Если вы пишете средства, запускаемые из командной строки, это полезный способ, поскольку другая программа может проверять возвращаемое значение, например, из сценария оболочки.

## Работа с классом NSData

При работе с файлами часто требуется читать данные во временную область хранения в памяти, которую называют *буфером* (*buffer*). Буфер часто используется при сборе данных для последующего вывода в файл. Класс Foundation NSData позволяет легко создавать буфер, читать в него содержимое файла или писать содержимое буфера в файл. Для 32-битного приложения в буфере NSData можно хранить до 2 Гб. В случае 64-битного приложения в таком буфере можно хранить до 8 Эб (экзабайт), то есть 8000 Гб информации!

Можно определять немутабельные (NSData) или мутабельные (NSMutableData) области памяти. Мы ознакомим вас с методами данного класса в этой главе, а также в последующих главах.

В программе 16.2 показано, как читать содержимое файла в буфер, определенный в памяти.

Эта программа читает содержимое файла newfile2 и записывает его в новый файл с именем newfile3. В некотором смысле это реализация операции копирования файла, хотя и не столь простая, как метод copyPath:toPath:handler:.

### Программа 16.2

```
// Создание копии файла

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSFileManager *fm;
 NSData *fileData;

 fm = [NSFileManager defaultManager];

 // Чтение файла newfile2
 fileData = [fm contentsAtPath: @"newfile2"];

 if (fileData == nil) {
 NSLog (@"File read failed!");
 return 1;
 }
```

```

// Запись данных в newfile3

if ([fm createFileAtPath: @"newfile3" contents: fileData
 attributes: nil] == NO) {
 NSLog(@"Couldn't create the copy!"); (Невозможно создать копию.)
 return 2;
}

NSLog(@"File copy was successful!"); (Копирование файла выполнено успешно)

[pool drain];
return 0;
}

```

### Вывод программы 16.2

File copy was successful! (Копирование файла выполнено успешно)

Метод `NSData contentsAtPath:` просто принимает имя пути и читает содержимое указанного файла в область памяти (которую он создает). Метод возвращает в качестве результата объект области памяти или `nil`, если операцию чтения не удается выполнить (например, если этот файл не существует или недоступен для чтения).

Метод `createFileAtPath:contents:attributes:` создает файл с указанными атрибутами (или использует атрибуты по умолчанию, если для аргумента атрибутов указано значение `nil`). Затем содержимое указанного объекта `NSData` записывается в файл. В нашем примере эта область памяти содержит данные прочитанного ранее файла.

## Работа с папками

В таблице 16.2 приводятся методы `NSFileManager` для работы с папками (каталогами). Многие из этих методов аналогичны методам для обычных файлов из таблицы 16.1 (обозначения такие же, как в таблице 16.1).

**Табл. 16.2.** Наиболее распространенные методы `NSFileManager` для работы с папками

| Метод                                                                 | Описание                                                                                               |
|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>-	NSString * currentDirectoryPath</code>                        | Получает текущую папку.                                                                                |
| <code>-	BOOL changeCurrentDirectoryPath: path</code>                  | Изменяет текущую папку.                                                                                |
| <code>-	BOOL copyPath: from toPath: to handler: handler</code>        | Копирует структуру папки ( <i>to</i> не может существовать заранее).                                   |
| <code>-	BOOL createDirectoryAtPath: path attributes: attr</code>      | Создает новую папку.                                                                                   |
| <code>-	BOOL fileExistsAtPath: path isDirectory: (BOOL *) flag</code> | Проверяет, содержится ли данный файл в папке (результат YES/NO сохраняется в переменной <i>flag</i> ). |

**Табл. 16.2. Наиболее распространенные методы `NSFileManager` для работы с папками (окончание)**

| Метод                                                             | Описание                                                                         |
|-------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <code>-NSArray *directoryContentsAtPath: path</code>              | Создает список содержимого папки.                                                |
| <code>-NSDirectoryEnumerator *enumeratorAtPath: path</code>       | Перечисляет содержимое папки.                                                    |
| <code>-BOOL removeFileAtPath: path<br/>handler: handler</code>    | Удаляет пустую папку.                                                            |
| <code>-BOOL movePath: from toPath:<br/>to handler: handler</code> | Переименовывает или перемещает папку ( <i>to</i> не может существовать заранее). |

В программе 16.3 показаны основные операции с папками.

### Программа 16.3

```
// Основные операции с папками

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *dirName = @"testdir";
 NSString *path;
 NSFileManager *fm;

 // Нужно создать экземпляр filemanager

 fm = [NSFileManager defaultManager];

 // Получение текущей папки

 path = [fm currentDirectoryPath];
 NSLog (@"Current directory path is %@", path); (Путь к текущей папке)

 // Создание новой папки
```

```
if ([fm createDirectoryAtPath: dirName attributes: nil] == NO) {
 NSLog (@"Couldn't create directory!"); (Невозможно создать папку)
 return 1;
}

// Переименование новой папки

if ([fm movePath: dirName toPath: @"newdir" handler: nil] == NO) {
 NSLog (@"Directory rename failed!");
 return 2;
}

// Смена папки на другую папку

if ([fm changeCurrentDirectoryPath: @"newdir"] == NO) {
 NSLog (@"Change directory failed!"); (Невозможно сменить папку)
 return 3;
}

// Получение и вывод пути к текущей рабочей папке

path = [fm currentDirectoryPath];
NSLog (@"Current directory path is %@", path); (Путь к текущей папке)

NSLog (@"All operations were successful!"); (Все операции выполнены успешно)

[pool drain];
return 0;
}
```

### Вывод программы 16.3

```
Current directory path is /Users/stevekochan/progs/ch16 (Путь к текущей папке)
Current directory path is /Users/stevekochan/progs/ch16/newdir
All operations were successful! (Все операции выполнены успешно)
```

Работу программы 16.3 легко понять из текста самой программы. Сначала мы получаем путь к текущей папке для информативных целей. Затем в текущей папке создается новая папка testdir. Затем в программе используется метод movePath:toPath:handler: для переименования этой новой папки из testdir в newdir. Помните, что этот метод позволяет также перемещать всю структуру папки (включая ее содержимое) из одного места файловой системы в другое.

После переименования новой папки программа делает эту новую папку текущей с помощью метода changeCurrentDirectoryPath:. Затем выводится путь к текущей папке, чтобы убедиться, что изменение было выполнено успешно.

## Перечисление содержимого папки

Получим список содержимого папки. Этот процесс можно осуществить с помощью метода `enumeratorAtPath:` или `directoryContentsAtPath:`. В первом случае каждый файл указанной папки перечисляется по отдельности. Если один из этих файлов является папкой, то по умолчанию его содержимое тоже рекурсивно перечисляется. Во время этого процесса мы можем динамически запрещать рекурсию (отправив сообщение `skipDescendants` объекту перечисления), чтобы содержимое папки не перечислялось.

В случае метода `directoryContentsAtPath:` выполняется перечисление указанной папки, и метод возвращает массив со списком. Если какой-либо из этих файлов является папкой, его содержимое не перечисляется данным методом. В программе 16.4 показано, как использовать каждый из этих методов.

### Программа 16.4

```
// Перечисление содержимого папки

#import <Foundation/NSString.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSArray.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *path;
 NSFileManager *fm;
 NSDirectoryEnumerator *dirEnum;
 NSArray *dirArray;

 // Создание экземпляра filemanager

 fm = [NSFileManager defaultManager];

 // Получение пути к текущей рабочей папке

 path = [fm currentDirectoryPath];

 // Перечисление содержимого папки

 dirEnum = [fm enumeratorAtPath: path];
 NSLog(@"%@", path); (Содержимое папки)
```

```
while ((path = [dirEnum nextObject]) != nil)
 NSLog(@"%@", path);

// Еще один способ перечисления содержимого папки
dirArray = [fm directoryContentsAtPath:
 [fm currentDirectoryPath]];
NSLog(@"Contents using directoryContentsAtPath:"); // Содержимое с помощью ...

for (path in dirArray)
 NSLog(@"%@", path);

[pool drain];
return 0;
}
```

#### Вывод программы 16.4

```
Contents of /Users/stevekochan/mysrc/ch16: (Содержимое папки)
a.out
dir1.m
dir2.m
file1.m
newdir
newdir/file1.m
newdir/output
path1.m
testfile
```

```
Contents using directoryContentsAtPath: (Содержимое с помощью directoryContentsAtPath:)
a.out
dir1.m
dir2.m
file1.m
newdir
path1.m
testfile
```

Рассмотрим следующую последовательность кода.

```
dirEnum = [fm enumeratorAtPath: path];

NSLog(@"Contents of %@:", path);

while ((path = [dirEnum nextObject]) != nil)
 NSLog(@"%@", path);
```

Мы начинаем перечисление содержимого папки с отправки сообщения `enumerationAtPath:` объекту `filemanager`, в данном случае – `fm`. Метод `enumeratorAtPath:` возвращает объект `NSDirectoryEnumerator`, который сохраняется в `dirEnum`. Теперь каждый раз при отправке сообщения `nextObject` этому объекту метод возвращает путь к следующему файлу в перечисляемой папке. Когда не остается файлов для перечисления, метод возвращает значение `nil`.

Из вывода программы 16.4 видно, чем отличаются эти методы. Метод `enumeratorAtPath:` создает список содержимого папки `newdir`, а метод `directoryContentsAtPath:` не создает. Если бы папка `newdir` содержала подпапки, то они тоже были бы перечислены методом `enumeratorAtPath:`.

Как говорилось выше, при выполнении цикла `while` в программе 16.4 вы могли бы запретить перечисление содержимого любых подпапок, внеся в код следующее изменение.

```
while ((path = [dirEnum nextObject]) != nil) {
 NSLog(@"%@", path);

 [fm fileExistsAtPath: path isDirectory: &flag];

 if (flag == YES)
 [dirEnum skipDescendents];
}
```

Здесь флаг представлен переменной типа `BOOL`. Метод `fileExistsAtPath:` сохраняет значение `YES` в этом флаге, если указанный путь содержится в папке, иначе сохраняется значение `NO`.

Вы можете вывести все содержимое `dirArray` с помощью одного вызова `NSLog`

```
NSLog(@"%@", dirArray);
```

вместо быстрого перечисления, использованного в программе.

## Работа с путями: `NSPathUtilities.h`

В `NSPathUtilities.h` включены расширения функций и категорий к `NSString` для работы с путями. Они позволяют сделать ваши программы более независимыми от структуры файловой системы и местонахождения конкретных файлов и папок. В программе 16.5 показано, как работать с функциями и методами из `NSPathUtilities.h`.

### Программа 16.5

```
// Основные операции с путями

#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSPathUtilities.h>
```

```
int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *fName = @"path.m";
 NSFileManager *fm;
 NSString *path, *tempdir, *extension, *homedir, *fullpath;
 NSString *upath = @"~/stevekochan/progs/../ch16./path.m";

 NSArray *components;

 fm = [NSFileManager defaultManager];

 // Получение временной рабочей папки

 tempdir = NSTemporaryDirectory ();

 NSLog (@"Temporary Directory is %@", tempdir); (Временная папка)

 // Извлечение базовой папки из текущей папки

 path = [fm currentDirectoryPath];
 NSLog (@"Base dir is %@", [path lastPathComponent]); (Базовая папка)

 // Создание полного пути к файлу fName в текущей папке

 fullPath = [path stringByAppendingPathComponent: fName];
 NSLog (@"fullpath to %@ is %@", fName, fullPath); (Полный путь к ...)

 // Получение расширения имени файла

 extension = [fullPath pathExtension];
 NSLog (@"extension for %@ is %@", fullPath, extension); (Расширение имени для ...)

 // Получение домашней папки пользователя

 homedir = NSHomeDirectory ();
 NSLog (@"Your home directory is %@", homedir); (Ваша домашняя папка ...)

 // Разбиение пути на его компоненты

 components = [homedir pathComponents];

 for (path in components)
 NSLog (@"%@", path);

 // "Стандартизация" пути
```

```
NSLog(@"%@", upath ,
 [upath stringByStandardizingPath]);

 [pool drain];
 return 0;
}
```

### Вывод программы 16.5

```
Temporary Directory is /var/folders/HT/HTyGLvSNHTuNb6NrMu07QE+++T/-Tmp-/
(Временная папка)
Base dir is examples (Базовая папка - examples)
fullpath to path.m is /Users/stevekochan/progs/examples/path.m (полный путь к ...)
extension for /Users/stevekochan/progs/examples/path.m is m (расширение имени для
...)
Your home directory is /Users/stevekochan (Ваша домашняя папка ...)
/
Users
stevekochan
~stevekochan/progs/../ch16./path.m => /Users/stevekochan/ch16/path.m
```

Функция `NSTemporaryDirectory` возвращает имя пути в системе для папки, в которой могут сохраняться временные файлы. Создавая временные файлы в этой папке, не забывайте удалять их по окончании использования. Кроме того, имена файлов должны быть уникальными, особенно в тех случаях, когда несколько экземпляров вашего приложения работает одновременно. (См. упражнение 5 в конце этой главы.) Это может произойти, если несколько пользователей выполнят вход в вашу систему и будут выполнять одно приложение.

Метод `lastPathComponent` извлекает последний файл, указанный в пути. Это полезно, если нужно получить из абсолютного имени пути только базовое имя файла.

Метод `stringByAppendingPathComponent:` позволяет присоединять имя файла в конце пути. Если имя пути, указанное как получатель, не заканчивается символом «слэш», метод вставляет этот символ в имя пути, чтобы отделить его от присоединяемого имени файла. Сочетая метод `currentDirectory` с методом `stringByAppendingPathComponent:`, можно создавать полный путь к файлу в текущей папке. Этот способ показан в программе 16.5.

Метод `pathExtension` дает расширение имени файла для указанного имени пути. Например, расширением для имени файла `path.m` является `m`, и данный метод возвращает это расширение. Если имя файла не содержит расширения, то метод просто возвращает пустую строку.

Функция `NSHomeDirectory` возвращает домашнюю папку для текущего пользователя. Чтобы получить домашнюю папку для определенного пользователя, можно использовать вместо этого функцию `NSHomeDirectoryForUser`, указав имя пользователя как аргумент для этой функции.

Метод `pathComponents` возвращает массив, содержащий каждый из компонентов указанного пути. В программе 16.5 последовательно извлекается каждый элемент возвращаемого массива, и каждый компонент отображается в отдельной строке вывода.

И, наконец, иногда имена пути содержат символы «тильда» (~), о чем мы уже говорили выше. Методы `FileManager` воспринимают ~ как сокращение представление домашней папки пользователя, или ~user для домашней папки указанного пользователя. Если ваши имена путей могут содержать символы «тильда», то раскрыть их позволяет метод `stringByStandardizingPath`. Этот метод возвращает путь в стандартизованном виде, то есть с удалением специальных символов. Метод `stringByExpandingTildeInPath` раскрывает только символ «тильда», если он присутствует в имени пути.

## Наиболее распространенные методы для работы с путями

В таблице 16.3 приводятся наиболее распространенные методы для работы с путями. В данной таблице *components* – это объект `NSArray`, содержащий строковые объекты для каждого компонента пути; *path* – это строковый объект, указывающий путь к файлу; *ext* – это строковый объект, указывающий расширение имени файла (например, @"mp4").

**Табл. 16.3.** Наиболее распространенные методы для работы с путями

| Метод                                                                      | Описание                                                                                                                              |
|----------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>+ (NSString *) pathWithComponents:</code><br><i>components</i>       | Создает допустимый путь из элементов <i>components</i> .                                                                              |
| <code>- (NSArray *) pathComponents</code>                                  | Разделяет путь на его составляющие компоненты.                                                                                        |
| <code>- (NSString *) lastPathComponent</code>                              | Извлекает последний компонент пути.                                                                                                   |
| <code>- (NSString *) pathExtension</code>                                  | Извлекает расширение имени из последнего компонента пути.                                                                             |
| <code>- (NSString *) stringByAppendingPathComponent:</code><br><i>path</i> | Добавляет путь <i>path</i> в конец существующего пути.                                                                                |
| <code>- (NSString *) stringByAppendingPathExtension:</code> <i>ext</i>     | Добавляет расширение имени <i>ext</i> к последнему компоненту пути.                                                                   |
| <code>- (NSString *) stringByDeletingLastPathComponent</code>              | Удаляет последний компонент пути.                                                                                                     |
| <code>- (NSString *) stringByDeletingPathExtension</code>                  | Удаляет расширение имени из последнего компонента пути.                                                                               |
| <code>- (NSString *) stringByExpandingTildeInPath</code>                   | Раскрывает символы «тильда» в пути как домашнюю папку текущего пользователя (~) или домашнюю папку указанного пользователя (@@~user). |

**Табл. 16.3. Наиболее распространенные методы для работы с путями (окончание)**

| Метод                                                          | Описание                                                                                          |
|----------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>-	NSString *(<br/>stringByResolvingSymlinksInPath</code> | Пытается разрешить (раскрыть) символические ссылки в пути.                                        |
| <code>-	NSString *(<br/>stringByStandardizingPath</code>       | Стандартизует путь, раскрывая ~, ..(родительская папка), .(текущая папка) и символические ссылки. |

В таблице 16.4 представлены имеющиеся *функции* для получения информации о пользователе, его домашней папке и его папке для хранения временных файлов.

**Табл. 16.4. Наиболее распространенные функции для работы с путями**

| Функция                                                        | Описание                                                                           |
|----------------------------------------------------------------|------------------------------------------------------------------------------------|
| <code>NSString *NSUserName (void)</code>                       | Возвращает имя входа ( <i>login</i> ) текущего пользователя.                       |
| <code>NSString *NSFullUserName (void)</code>                   | Возвращает полное пользовательское имя текущего пользователя.                      |
| <code>NSString *NSHomeDirectory (void)</code>                  | Возвращает путь к домашней папке текущего пользователя.                            |
| <code>NSString *NSHomeDirectoryForUser (NSString *user)</code> | Возвращает домашнюю папку пользователя <i>user</i> .                               |
| <code>NSString *NSTemporaryDirectory (void)</code>             | Возвращает путь к папке, которую можно использовать для создания временного файла. |

Возможно, вам потребуется также функция Foundation `NSSearchPathForDirectoriesInDomains` для обнаружения в системе специальных папок, таких как Application.

## Копирование файлов и использование класса NSProcessInfo

В программе 16.6 показано средство командной строки для реализации простой операции копирования файлов. Эта команда применяется следующим образом.

```
copy from-file (bc[j]lysq afqk) to-file (afqk-rjgbz)
```

В отличие от метода `NSFileManager copyPath:toPath:handler:`, это средство командной строки позволяет также использовать *to-file* как имя папки. Тогда эта папка копируется в папку *to-file* под именем *from-file*. Еще одно отличие от указанного метода: если *to-file* уже существует, его содержимое перезаписывается. Это в большей степени согласуется со стандартной командой копирования UNIX (*cp*).

Для получения имен файлов из командной строки можно использовать аргументы `argc` и `argv`, передаваемые в `main`. В качестве этих аргументов в командной строке указываются, соответственно, целый ряд типов аргументов (включая имя команды) и указатель на массив символьных С-строк.

Вместо обработки С-строк, что приходится делать при работе с `argv`, используйте класс `Foundation NSProcessInfo`. `NSProcessInfo` содержит методы, позволяющие задавать и считывать различные типы информации о выполняемом приложении (то есть вашем *процессе*). Эти методы приводятся в таблице 16.5.

**Табл. 16.5. Методы класса `NSProcessInfo`**

| Метод                                                    | Описание                                                                                                                                                                                   |
|----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>+ (NSProcessInfo *) processInfo</code>             | Возвращает информацию о текущем процессе.                                                                                                                                                  |
| <code>- (NSArray *) arguments</code>                     | Возвращает аргументы для текущего процесса в виде массива объектов <code>NSString</code> .                                                                                                 |
| <code>- (NSDictionary *) environment</code>              | Возвращает словарь, состоящий из пар «переменная/значение», представляющих текущие переменные среды (например, <code>PATH</code> и <code>HOME</code> ) вместе с их значениями.             |
| <code>- (int) processIdentifier</code>                   | Возвращает идентификатор процесса, то есть уникальный номер, назначаемый операционной системой для идентификации каждого выполняемого процесса.                                            |
| <code>- (NSString *) processName</code>                  | Возвращает имя текущего выполняемого процесса.                                                                                                                                             |
| <code>- (NSString *) globallyUniqueString</code>         | Возвращает при каждом вызове новую уникальную строку. Это можно использовать для генерации уникальных имен временных файлов (см. упражнение 5).                                            |
| <code>- (NSString *) hostName</code>                     | Возвращает имя хост-системы (возвращает <code>Steve-Kochans-Computer.local</code> в моей системе Mac OS X).                                                                                |
| <code>- (NSUInteger) operatingSystem</code>              | Возвращает число, обозначающее операционную систему (возвращает значение 5 на моем Mac).                                                                                                   |
| <code>- (NSString *) operatingSystemName</code>          | Возвращает имя операционной системы (возвращает константу <code>NSMACHOperatingSystem</code> на моем Mac, где возможные возвращаемые значения определены в <code>NSProcessInfo.h</code> ). |
| <code>- (NSString *) operatingSystemVersionString</code> | Возвращает текущую версию операционной системы (возвращает <code>Version 10.5.4 (Build 9E17)</code> в моей системе Mac OS X).                                                              |

Табл. 16.5. Методы класса NSProcessInfo (окончание)

| Метод                                         | Описание                                                                                                                                                                                                                |
|-----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - (void) setProcessName:<br>(NSString *) name | Задает имя <i>name</i> для текущего процесса. Следует использовать с осторожностью, поскольку необходимо учитывать некоторые предположения об имени вашего процесса (например, в настройках пользователя по умолчанию). |

**Программа 16.6**

```
// Реализация простой утилиты копирования

#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSPathUtilities.h>
#import <Foundation/NSProcessInfo.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSFileManager *fm;
 NSString *source, *dest;
 BOOL isDir;
 NSProcessInfo *proc = [NSProcessInfo processInfo];
 NSArray *args = [proc arguments];

 fm = [NSFileManager defaultManager];

 // Проверка того, что в командной строке заданы два аргумента

 if ([args count] != 3) {
 NSLog (@"Usage: %@ src dest", [proc processName]);
 return 1;
 }

 source = [args objectAtIndex: 1];
 dest = [args objectAtIndex: 2];

 // Проверка того, что исходный файл можно читать

 if ([fm isReadableFileAtPath: source] == NO) {
 NSLog (@"Can't read %@", source);
 return 2;
 }
}
```

```

// Является ли целевой файл (dest) папкой?
// Если да, то исходный файл (source) добавляется в конец целевого пути

[fm fileExistsAtPath: dest isDirectory: &isDir];

if (isDir == YES)
 dest = [dest stringByAppendingPathComponent:
 [source lastPathComponent]];

// Удаление целевого файла, если он уже существует

[fm removeFileAtPath: dest handler: nil];

// Все сделано, можно выполнять копирование

if ([fm copyPath: source toPath: dest handler: nil] == NO) {
 NSLog (@"Copy failed!"); return 3;
}

NSLog (@"Copy of %@ to %@ succeeded!", source, dest);

[pool drain];
return 0;
}

```

**Выход программы 16.6**

```

$ ls -l смотрим, какие файлы у нас есть
total 96
-rwxr-xr-x 1 stevekoc staff 19956 Jul 24 14:33 copy
-rw-r--r-- 1 stevekoc staff 1484 Jul 24 14:32 copy.m
-rw-r--r-- 1 stevekoc staff 1403 Jul 24 13:00 file1.m
drwxr-xr-x 2 stevekoc staff 68 Jul 24 14:40 newdir
-rw-r--r-- 1 stevekoc staff 1567 Jul 24 14:12 path1.m
-rw-r--r-- 1 stevekoc staff 84 Jul 24 13:22 testfile
$ copy попытка команды без аргументов
Usage: copy src dest (Использование: ...)
$ copy foo copy2
Can't read foo (Невозможно прочитать foo)
$ copy copy.m backup.m
Copy of copy.m to backup.m succeeded! (Копирование copy.m в backup.m выполнено!)
$ diff copy.m backup.m сравнение файлов
$ copy copy.m newdir попытка копирования в папку
Copy of copy.m to newdir/copy.m succeeded! (Копирование copy.m в newdir/copy.m
выполнено!)
$ ls -l newdir
total 8
-rw-r--r-- 1 stevekoc staff 1484 Jul 24 14:44 copy.m
$

```

Метод `NSProcessInfo arguments` возвращает массив строковых объектов. Первый элемент этого массива – имя процесса, остальные элементы содержат аргументы, которые вводятся в командной строке.

Сначала проверяем, что в командной строке введено два аргумента. Для этого проверяем размер массива `args`, возвращаемого методом `arguments`. Если проверка дает правильный результат, то программа затем извлекает имена исходного и целевого файлов из массива `args`, присваивая их значения переменным `source` и `dest` соответственно.

Затем программа проверяет, может ли читаться исходный файл, выдает сообщение об ошибке и завершает работу, если файл не читается.

#### Оператор

```
[fm fileExistsAtPath: dest.isDirectory: &isDir];
```

проверяет файл, который указывается переменной `dest`, чтобы определить, не является ли он папкой. Как вы видели выше, ответ (`YES` или `NO`) сохраняется в переменной `isDir`.

Если `dest` является папкой, то нужно добавить в конец имени этой папки имя исходного файла в качестве последнего компонента пути. Для этого используется метод работы с папками `stringByAppendingPathComponent:`. Например, если значением переменной `source` является строка `ch16/copy1.m`, а значением переменной `dest` является `/Users/stevekochan/progs`, и это папка, то нужно изменить значение `dest` на `/Users/stevekochan/progs/copy1.m`.

Метод `copyPath:ToPath:handler:` не допускает перезаписи файлов. Таким образом, чтобы избежать ошибки, программа пытается удалить сначала целевой файл с помощью метода `removeFileAtPath:handler:`. На самом деле не имеет значения, выполнит ли эту работу данный метод, поскольку он в любом случае выдаст ошибку, если целевой файл не существует.

Если достигнут конец программы, то предполагается, что задача выполнена, и выдается соответствующее сообщение.

## Основные файловые операции: `NSFileHandle`

Методы из класса `NSFileHandle` позволяют выполнять более сложную работу с файлами. В начале этой главы мы перечислили некоторые возможности этих методов.

В общем случае при работе с файлом нужно выполнить следующие шаги.

1. Открыть файл и получить объект класса `NSFileHandle` для ссылки на этот файл в последующих операциях ввода-вывода.
2. Выполнить необходимые операции ввода-вывода с открытым файлом.
3. Закрыть файл.

В таблице 16.6 приводятся некоторые наиболее распространенные методы `NSFileHandle`. В этой таблице `fh` – это объект `NSFileHandle`, `data` – это объект `NSData`, `path` – это объект `NSString`, `offset` – это значение типа `unsigned long long`.

Табл. 16.6. Наиболее распространенные методы NSFileHandle

| Метод                                                             | Описание                                                                                                 |
|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>+ (NSFileHandle *) fileHandleForReadingAtPath: path</code>  | Открывает файл для чтения.                                                                               |
| <code>+ (NSFileHandle *) fileHandleForWritingAtPath: path</code>  | Открывает файл для записи.                                                                               |
| <code>+ (NSFileHandle *) fileHandleForUpdatingAtPath: path</code> | Открывает файл для изменения (чтение и запись).                                                          |
| <code>- (NSData *) availableData</code>                           | Возвращает данные, доступные для чтения, из устройства или канала.                                       |
| <code>- (NSData *) readDataToEndOfFile</code>                     | Читает оставшиеся данные до конца файла порциями не более ( <code>UINT_MAX</code> ) байт.                |
| <code>- (NSData *) readDataOfLength: (NSUInteger) bytes</code>    | Читает заданное количество байтов ( <code>bytes</code> ) из файла.                                       |
| <code>- (void) writeData: data</code>                             | Записывает данные ( <code>data</code> ) в файл.                                                          |
| <code>- (unsigned long long) offsetInFile</code>                  | Получает текущее смещение в файле.                                                                       |
| <code>- (void) seekToFileOffset: offset</code>                    | Задает текущее смещение ( <code>offset</code> ) в файле.                                                 |
| <code>- (unsigned long long) seekToEndOfFile</code>               | Помещает текущее смещение в конец файла.                                                                 |
| <code>- (void) truncateFileAtOffset: offset</code>                | Задает размер файла, равный <code>offset</code> байт (с заполнением при необходимости свободного места). |
| <code>- (void) closeFile</code>                                   | Закрывает файл.                                                                                          |

Здесь не приводятся методы `NSFileHandle` для стандартного ввода, стандартного вывода, стандартных ошибок и `null`-устройства. Они имеют форму `fileHandleWithDevice`, где обозначение `Device` может быть представлено как `StandardInput`, `StandardOutput`, `StandardError` или `NullDevice`.

Здесь также не приводятся методы для чтения и записи данных в фоновом режиме, то есть асинхронно.

Следует отметить, что класс `FileHandle` не предусматривает создание файлов. Это нужно делать с помощью методов `FileManager`, которые были описаны выше. Например, в методах `fileHandleForWritingAtPath:` и `fileHandleForUpdatingAtPath:` предполагается, что файл существует, и они возвращают значение `nil`, если его нет. В обоих случаях смещение задастся в начале файла, поэтому запись (или чтение в режиме `update`) начинается с начала файла. Кроме того, если вы работали с UNIX, то, видимо, обратили внимание, что открытие файла для записи не вызывает усечения файла. Вы должны сделать это самостоятельно, если это вам нужно.

Программа 16.7 открывает исходный файл `testfile`, который мы создали в начале главы, читает его содержимое и копирует его в файл с именем `testout`.

### Программа 16.7

```
// Основные операции обработки файлов
// Предполагается, что существует файл с именем "testfile"
// в текущей рабочей папке
```

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSFileHandle.h>
#import <Foundation/NSFileManager.h>

#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSFileHandle *inFile, *outFile;
 NSData *buffer;

 // Открытие файла testfile для чтения

 inFile = [NSFileHandle fileHandleForReadingAtPath: @"testfile"];
 if (inFile == nil) {
 NSLog (@"Open of testfile for reading failed");
 // (Невозможно открыть testfile для чтения)
 return 1;
 }

 // Создание выходного файла, если это нужно

 [[NSFileManager defaultManager] createFileAtPath: @"testout"
 contents: nil attributes: nil];

 // Теперь открытие выходного файла для записи

 outFile = [NSFileHandle fileHandleForWritingAtPath: @"testout"];

 if (outFile == nil) {
 NSLog (@"Open of testout for writing failed");
 // (Невозможно открыть testfile для записи)
 return 2;
 }

 // Усечение выходного файла, поскольку он может содержать данные

 [outFile truncateFileAtOffset: 0];

 // Чтение данных из inFile и запись в outFile

 buffer = [inFile readDataToEndOfFile];

 [outFile writeData: buffer];
```

```
// Закрытие обоих файлов

[inFile closeFile];
[outFile closeFile];

// Проверка содержимого файла

NSLog(@"%@", [NSString stringWithFormat:@testout encoding:
NSUTF8StringEncoding error: nil]);

[pool drain];
return 0;
}
```

### Вывод программы 16.7

This is a test file with some data in it. (Это тестовый файл с некоторыми данными.)  
Here's another line of data. (Это еще одна строка данных.)  
And a third. (И третья.)

Метод `readDataToEndOfFile`: читает данные порциями не более `UINT_MAX` байт, что определено в `<limits.h>` и равно  $FFFF_{16}$ . Этого вполне достаточно для приложения, которое требуется написать. Мы можем также разбить эту работу для чтения и записи небольшими порциями и задать цикл для единовременной пересылки полного буфера между файлами с помощью метода `readDataOfLength`: Например, размер буфера можно задать равным 8192 (8 кбайт) или 131072 (128 кбайт). Как правило, размер равен степени 2, поскольку базовая операционная система обычно выполняет свои операции ввода-вывода именно такими порциями данных. Вы можете опробовать различные значения, чтобы определить, что лучше в вашей системе.

Если метод чтения обнаруживает конец файла без чтения каких-либо данных, он возвращает пустой объект `NSData` (то есть буфер, не содержащий никаких данных). Мы можем применять к буферу метод `length` и проверять его на равенство нулю, чтобы определить, остались ли какие-то данные для чтения из файла.

Если мы открываем файл для изменения, смещение устанавливается на начало файла. Это смещение можно изменить путем поиска внутри файла и затем выполнить в файле операции чтения или записи. Например, для поиска 10-го байта в файле с описателем `databaseHandle` можно написать в сообщении следующее выражение.

```
[databaseHandle seekToFileOffset: 10];
```

Для указания относительных позиций в файле нужно получить текущее смещение в файле и затем добавлять или вычитать из него относительное смещение. Например, чтобы пропустить следующие 128 байтов в файле, нужно написать следующее.

```
[databaseHandle seekToFileOffset:
[databaseHandle offsetInFile] + 128];
```

Для смещения на пять целых данных в файле можно написать следующее.

```
[databaseHandle seekToFileOffset:
 [databaseHandle offsetInFile] - 5 * sizeof (int)];
```

Программа 16.8 добавляет содержимое одного файла ко второму. Второй файл открывается для записи, выполняется поиск конца этого файла и затем содержимое первого файла записывается во второй.

### Программа 16.8

```
// Добавление содержимого файла "fileA" в конец файла "fileB"

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>

#import <Foundation/NSFileHandle.h>
#import <Foundation/NSFileManager.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSData.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSFileHandle *inFile, *outFile;
 NSData *buffer;

 // Открытие файла fileA для чтения

 inFile = [NSFileHandle fileHandleForReadingAtPath: @"fileA"];
 if (inFile == nil) {
 NSLog (@"Open of fileA for reading failed");
 return 1;
 }

 // Открытие файла fileB для изменения

 outFile = [NSFileHandle fileHandleForWritingAtPath: @"fileB"];
 if (outFile == nil) {
 NSLog (@"Open of fileB for writing failed");
 return 2;
 }

 // Поиск конца outFile

 [outFile seekToEndOfFile];

 // Чтение inFile и запись его содержимого в outFile
```

```
buffer = [inFile readDataToEndOfFile];
[outFile writeData: buffer];

// Закрытие обоих файлов

[inFile closeFile];
[outFile closeFile];

[pool drain];
return 0;
}
```

*Содержимое FileA перед запуском программы 16.8*

This is line 1 in the first file. (Это строка 1 в первом файле)  
This is line 2 in the first file. (Это строка 2 в первом файле)

*Содержимое FileB перед запуском программы 16.8*

This is line 1 in the second file. (Это строка 1 во втором файле)  
This is line 2 in the second file. (Это строка 2 во втором файле)

### Вывод программы 16.8

Contents of fileB (Содержимое fileB)

This is line 1 in the second file.  
This is line 2 in the second file.  
This is line 1 in the first file.  
This is line 2 in the first file.

Вывод программы показывает, что содержимое первого файла успешно добавлено в конец второго файла. `seekToEndOfFile` возвращает текущее смещение файла после выполнения поиска. В данном случае мы игнорируем эту информацию, но она позволяет получить размер файла.

## Упражнения

1. Внесите изменения в программу копирования (программа 16.6), чтобы она допускала более одного исходного файла для копирования в папку, аналогично стандартной команде UNIX `cp`. Например, команда

**\$ copy copy1.m file1.m file2.m progs**

должна копировать три файла (`copy1.m`, `file1.m` и `file2.m`) в папку `progs`. Предусмотрите, что если указано более одного файла, то последним аргументом на самом деле является существующая папка.

2. Напишите средство командной строки с именем `myfind`, которое принимает два аргумента. Первый из них указывает начальную папку для поиска, а второй – имя файла, который нужно найти. Например, командная строка

```
$ myfind /Users proposal.doc
/Users/stevekochan/MyDocuments/proposals/proposal.doc
$
```

начинает поиск в файловой системе с `/Users`, чтобы найти файл `proposal.doc`. Выведите полный путь к файлу, если он найден (в том виде, как показано выше), или соответствующее сообщение, если он не найден.

3. Напишите вашу собственную версию стандартных средств UNIX `basename` и `dirname`.
4. Используя класс `NSProcessInfo`, напишите программу, которая выводит всю информацию, возвращаемую каждым из ее методов-получателей (`getter`).
5. Используя функцию `NSTemporaryDirectory` из `NSPathUtilities.h` и метод `NSProcessInfo globallyUniqueString`, описанные в этой главе, добавьте в `NSString` категорию с именем `TempFiles` и определите в ней метод с именем `temporaryFileName`, который возвращает при каждом вызове новое уникальное имя файла.
6. Внесите изменения в программу `I6.7`, чтобы в файле выполнялись чтение и запись `kBufSize` байт, причем `kBufSize` нужно определить в начале вашей программы. Обязательно проверьте эту программу на больших файлах (размером больше `kBufSize` байт).
7. Откройте файл, считывайте его содержимое по 128 байт и выводите эти данные на терминал. Используйте метод `FileHandle fileHandleWithStandardOutput`, чтобы получить описатель для вывода на терминал.

## Глава 17

# Управление памятью

На протяжении этой книги мы неоднократно затрагивали тему управления памятью. К этому моменту вам должно быть понятно, в каких случаях вы должны сами освобождать память, занятую объектами, и в каких случаях вы не обязаны это делать. Рассматривая даже небольшие примеры, мы постоянно подчеркивали, насколько важно уделять внимание управлению памятью для освоения практики надежного программирования и разработки программ, не допускающих утечки памяти.

Продуманное использование памяти может оказаться критически важным для работы приложения. Например, при создании интерактивного приложения для рисования многих объектов необходимо следить, чтобы по мере выполнения программы количество потребляемых ресурсов памяти не увеличивалось. В таких случаях вы обязаны аккуратно управлять этими ресурсами и освобождать их, когда они становятся не нужны. Это означает, что ресурсы следует освобождать во время выполнения программы, не дожидаясь ее окончания.

В этой главе мы обсудим стратегию выделения памяти в Foundation, пул автоматического высвобождения (*autorelease pool*) и идею «удержания» (*retain*) объектов. Вы узнаете также о счетчике ссылок (*reference count*) объекта и о сборке мусора (*garbage collection*), которая облегчает задачу удержания и последующего высвобождения объектов. Однако, как вы увидите, сборку мусора нельзя использовать для приложений iPhone, поэтому вы должны знать способы управления памятью, которые описываются в этой книге.

## Автоматически высвобождаемый пул

Вы уже знакомы с автоматически высвобождаемым пулом (пулом автоматического освобождения памяти) из примеров во второй части книги. При работе с программами Foundation вы должны создавать этот пул для работы с объектами Foundation. Именно в этом пуле программа следит за объектами для их дальнейшего высвобождения. Как уже говорилось, пул в приложении можно задать с помощью следующего вызова.

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

После создания этого пула Foundation автоматически добавляет в него определенные массивы, строки, словари и другие объекты. Закончив использование пула, можно освободить память, которую он использует, отправив сообщение *drain*.

```
[pool drain];
```

Автоматически высвобождаемый пул называется так потому, что любые объекты, которые помечаются как автоматически высвобождаемые (*autorelease*) и поэтому добавляются в этот пул, автоматически высвобождаются, когда высвобождается сам пул. В программе можно иметь несколько *autorelease*-пулов, и они могут быть также вложенными.

Если программа создает много временных объектов (что часто происходит при выполнении кода в цикле), может потребоваться создание нескольких *autorelease*-пулов. Например, в следующем фрагменте кода показано, как создавать *autorelease*-пулы для высвобождения временных объектов, создаваемых на каждом шаге цикла *for*.

```
NSAutoreleasePool *tempPool;
...
for (i = 0; i < n; ++i) {
 tempPool = [[NSAutoReleasePool alloc] init];
 ... // здесь выполняется много работы с временными объектами
 [tempPool drain];
}
```

Отметим, что при опустошении пула (*pool drain*) *autorelease*-пул содержит не сами объекты, а только ссылку на объекты. Чтобы добавить объект в текущий *autorelease*-пул для последующего высвобождения, нужно отправить сообщение *autorelease*.

```
[myFraction autorelease];
```

Система добавит *myFraction* в *autorelease*-пул для автоматического высвобождения. Как будет показано ниже, метод *autorelease* полезен, чтобы помечать объекты внутри метода для их устранения.

## Подсчет ссылок

Описывая базовый класс объектов Objective-C *NSObject*, мы говорили, что память выделяется с помощью метода *alloc*, и ее можно в дальнейшем освободить с помощью сообщения *release*. К сожалению, это не всегда так просто. Выполняемое приложение может ссылаться на объект, который может быть создан в нескольких местах; объект может быть также сохранен в массиве или, например, к нему может быть обращение с помощью переменной экземпляра. Мы не можем освободить память, занимаемую объектом, пока не будем уверены, что все закончили использовать этот объект.

К счастью, Foundation framework включает удобное решение для отслеживания числа ссылок на объект. Это довольно простой способ, который называется *подсчетом ссылок* (*reference count*). Он состоит в следующем. При создании объекта его счетчик ссылок устанавливается равным 1. Каждый раз, когда нужно учесть объект, мы увеличиваем его счетчик ссылок на 1, отправляя сообщение *retain*, как в следующей строке.

```
[myFraction retain];
```

Некоторые методы в Foundation framework тоже наращивают этот счетчик ссылок, например, когда объект добавляется в массив.

Когда объект уже не нужен, мы уменьшаем на 1 его счетчик ссылок, отправляя сообщение `release`, как в следующей строке.

```
[myFraction release];
```

Когда счетчик ссылок объект становится равным 0, система «понимает», что этот объект больше не нужен (поскольку на него нет больше ссылок), и поэтому она *освобождает (deallocate)* его память. Для этого объекту отправляется сообщение `dealloc`.

Успешное осуществление этой стратегии требует аккуратности от программиста, чтобы счетчик ссылок правильно наращивался и уменьшался во время выполнения программы. Как вы увидите ниже, система выполняет только часть этой работы.

Рассмотрим подсчет ссылок несколько подробнее. Объекту можно отправить сообщение `retainCount`, чтобы получить его счетчик ссылок (или *удержаний, retain*). Обычно вы не будете использовать этот метод, но здесь мы рассмотрим его в иллюстративных целях (см. программу 17.1). Отметим, что он возвращает целое без знака (`unsigned int`) типа `NSUInteger`.

### Программа 17.1

```
// Знакомство с подсчетом ссылок

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSNumber.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSNumber *myInt = [NSNumber numberWithInt: 100];
 NSNumber *myInt2;
 NSMutableArray *myArr = [NSMutableArray array];

 NSLog (@"myInt retain count = %lx",
 (unsigned long) [myInt retainCount]);

 [myArr addObject: myInt];
 NSLog (@"after adding to array = %lx",
 (unsigned long) [myInt retainCount]);

 myInt2 = myInt;
 NSLog (@"after asssgnment to myInt2 = %lx",
 (unsigned long) [myInt retainCount]);

 [myInt retain];
 NSLog (@"myInt after retain = %lx",
```

```

(unsigned long) [myInt retainCount];

 NSLog (@"myInt2 after retain = %lx",
(unsigned long) [myInt2 retainCount]);

[myInt release];
 NSLog (@"after release = %lx",
(unsigned long) [myInt retainCount]);

[myArr removeObjectAtIndex: 0];
 NSLog (@"after removal from array = %lx",
(unsigned long) [myInt retainCount]);

[pool drain];
return 0;
}

```

### Вывод программы 17.1

```

myInt retain count = 1 (счетчик удержаний myInt)
after adding to array = 2 (после добавления в массив)
after assignment to myInt2 = 2 (после присваивания myInt2)
myInt after retain = 3 (myInt после удержания)
myInt2 after retain = 3 (myInt2 после удержания)
after release = 2 (после release)
after removal from array = 1 (после удаления из массива)

```

Объекту `NSNumber myInt` присваивается целое значение 100, и вывод показывает, что начальное число его удержаний равно 1. Затем этот объект добавляется в массив `myArr` с помощью метода  `addObject:`. Обратите внимание, что после этого его счетчик ссылок равен 2. Метод  `addObject:` делает это автоматически; в документации по  `addObject:` описан этот факт. Добавление объекта в любой тип коллекции увеличивает его счетчик ссылок. Это означает, что когда мы высвобождаем (`release`) добавленный ранее объект, на него можно будет по-прежнему ссылаться из массива, и он не будет высвобожден.

Затем мы присваиваем `myInt` переменной `myInt2`. Отметим, что это не приводит к наращиванию счетчика ссылок, что может вызвать в дальнейшем потенциальные проблемы. Например, если счетчик ссылок для `myInt` уменьшится до 0 и его память будет освобождена, `myInt2` будет содержать неверную ссылку на объект (напомним, что присваивание `myInt` переменной `myInt2` не приводит к копированию самого объекта, а только к созданию указателя на место в памяти, где находится сам объект).

Поскольку `myInt` теперь имеет еще одну ссылку (через `myInt2`), мы наращиваем его счетчик ссылок, отправляя ему сообщение `retain`. Это происходит в следующей строке программы 17.1. Как мы видим, после отправки сообщения `retain` счетчик ссылок становится равным 3. Первая ссылка — это сам объект, вторая ссылка делается из массива и третья — во время присваивания. Сохранение элемента в массиве вызывает автоматическое наращивание счетчика ссылок, а присваивание другому элементу — нет, поэтому мы должны сделать это сами. Отметим, что

при выводе счетчик ссылок `myInt` и на `myInt2` дает одинаковое значение 3; дело в том, что в обоих случаях это ссылка на один и тот же объект в памяти.

Предположим, что мы прекратили использовать объект `myInt` в программе. Это можно сообщить системе, отправив объекту сообщение `release`. Как мы можем видеть, его счетчик ссылок в результате уменьшается с 3 до 2. Счетчик не равен 0; это означает, что продолжают действовать другие ссылки (из массива и через `myInt2`). Система не освобождает память, используемую этим объектом, поскольку счетчик ссылок не равен нулю.

После удаления первого элемента из массива `myArr` с помощью метода `removeObjectAtIndex`: мы видим, что счетчик ссылок автоматически уменьшился до 1. В общем случае удаление объекта из любой коллекции сопровождается уменьшением на 1 его счетчика ссылок. Поэтому следующая последовательность кода может вызвать проблемы.

```
myInt = [myArr objectAtIndex: 0];
...
[myArr removeObjectAtIndex: 0]
...
```

Дело в том, что в данном случае объект, на который ссылается `myInt`, может стать недействительным после вызова метода `removeObjectAtIndex`; если его счетчик ссылок уменьшился до 0. Конечно, для решения этой проблемы нужно удержать (`retain`) `myInt` после считывания из массива, чтобы на его ссылку не повлияло то, что происходит в других местах.

## Подсчет ссылок и строки

В программе 17.2 показано, как действует подсчет ссылок для строковых объектов.

### Программа 17.2

```
// Подсчет ссылок в случае строковых объектов.

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *myStr1 = @"Constant string";
 NSString *myStr2 = [NSString stringWithString: @"string 2"];
 NSMutableString *myStr3 = [NSMutableString stringWithString: @"string 3"];
 NSMutableArray *myArr = [NSMutableArray array];

 NSLog (@"Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx",
 (unsigned long) [myStr1 retainCount],
 (unsigned long) [myStr2 retainCount],
```

```

(unsigned long) [myStr3 retainCount]);

[myArr addObject: myStr1];
[myArr addObject: myStr2];
[myArr addObject: myStr3];

NSLog (@"Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx",
 (unsigned long) [myStr1 retainCount],
 (unsigned long) [myStr2retainCount],
 (unsigned long) [myStr3 retainCount]);

[myArr addObject: myStr1];
[myArr addObject: myStr2];
[myArr addObject: myStr3];
NSLog (@"Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx",
 (unsigned long) [myStr1 retainCount],
 (unsigned long) [myStr2retainCount],
 (unsigned long) [myStr3 retainCount]);

[myStr1 retain];
[myStr2 retain];
[myStr3 retain];

NSLog (@"Retain count: myStr1: %lx, myStr2: %lx, myStr3: %lx",
 (unsigned long) [myStr1 retainCount],
 (unsigned long) [myStr2 retainCount],
 (unsigned long) [myStr3 retainCount]);

// Уменьшение счетчика ссылок myStr3 снова до 2
[myStr3 release];

[pool drain];
return 0;
}

```

**Вывод программы 17.2**

```

Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 1 (Счетчик ссылок:)
Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 2
Retain count: myStr1: ffffffff, myStr2: ffffffff, myStr3: 3

```

Объекту NSString myStr1 присваивается строка NSConstantString @'Constant string' (Константная строка). Выделение места в памяти для константных строк отличается от других объектов. Константные строки не имеют механизма подсчета ссылок, поскольку их нельзя высвободить. Именно поэтому при отправке сообщения retainCount переменной myStr1 счетчик возвращает значение 0xffffffff. (Это на самом деле максимально возможное целое значение без знака, то есть `UINT_MAX` в стандартном header-файле `<limits.h>`.)

**Примечание.** Очевидно, что в некоторых системах счетчик ссылок, возвращаемый для константных строк в программе 17.2, дает значение 0x7fffffff (а не 0xffffffff), что является максимально возможным целым значением со знаком, то есть INT\_MAX.

Отметим, что то же самое относится к немутабельному строковому объекту, который инициализируется с константной строкой: он тоже не имеет счетчика ссылок, что подтверждается счетчиком ссылок, выведенным для myStr2.

**Примечание.** В данном случае система уже достаточно «сообразительна», поэтому она определила, что немутабельный строковый объект инициализируется с помощью константного строкового объекта. До выпуска Leopard такая оптимизация не выполнялась, и поэтому для myStr2 действовал счетчик удержаний.

### В операторе

```
NSMutableString *myStr3 = [NSMutableString stringWithString: @"string 3"];
```

переменной myStr3 присваивается строка, полученная из копии константной символьной строки @"string 3". Мы создали копию этой строки, поскольку классу NSMutableString было передано сообщение `stringWithString:`, указывающее, что содержимое строки может быть изменено в ходе выполнения программы. А поскольку содержимое константных символьных строк нельзя изменить, система не может сделать так, чтобы переменная myStr3 только указывала на константную строку @"string 3", как это было сделано в случае myStr2.

Поэтому строковый объект myStr3 действительно имеет счетчик ссылок, что подтверждается результатами вывода. Счетчик ссылок можно изменить путем добавления этой строки к массиву или передачи ему сообщения `retain`, что подтверждается результатами вывода с помощью последних двух вызовов  `NSLog`. Метод Foundation `stringWithString:` добавил этот объект в autorelease-пул при его создании. Метод Foundation `array` также добавил массив myArr в этот пул.

Прежде чем высвободить сам autorelease-пул, высвобождается myStr3. В результате его счетчик ссылок уменьшается до 2. Высвобождение autorelease-пула уменьшает счетчик ссылок этого объекта до 0, что приводит к освобождению занятой им памяти. Как это происходит? При высвобождении autorelease-пула каждый из объектов этого пула получает сообщение `release`, и это сообщение передается объекту столько раз, сколько было передано сообщений `autorelease`. Поскольку строковый объект myStr3 был добавлен в autorelease-пул при создании этого объекта с помощью метода `stringWithString:`, ему передается сообщение `release`. Это уменьшает его счетчик ссылок до 1. При высвобождении массива в autorelease-пуле также происходит высвобождение каждого из его элементов. Поэтому при высвобождении массива myArr из пула каждому из его элементов (включая myStr3) передается сообщение `release`. Это уменьшает его счетчик ссылок до 0, в результате чего его память должна быть освобождена.

Следите, чтобы не было лишних высвобождений объекта. Если в программе 17.2 сделать счетчик ссылок myStr3 меньше 2 до высвобождения самого пула, то пул будет содержать ссылку на неверный объект. Затем при высвобождении самого пула ссылка на неверный объект вызовет, скорее всего, аварийное завершение программы с ошибкой неверной сегментации (`segmentation fault`).

## Подсчет ссылок и переменные экземпляра

Счетчикам ссылок необходимо уделять внимание при работе с переменными экземпляра. Вспомним метод `setName:` из класса `AddressCard`.

```
-{void) setName: (NSString *) theName
{
 [name release];
 name = [[NSString alloc] initWithString: theName];
}
```

Предположим, что вместо этого мы определили `setName:` следующим образом и он не получил владения своим объектом `name`.

```
-{void) setName: (NSString *) theName
{
 name = theName;
}
```

Эта версия метода получает строку, представляющую имя человека, и сохраняет ее в переменной экземпляра `name`. Казалось бы, здесь все очевидно, но рассмотрим следующий вызов метода.

```
NSString *newName;
...
[myCard setName: newName];
```

Предположим, что `newName` — это пространство временного хранения для имени человека, которого добавили в адресную карточку, и что в дальнейшем это пространство нужно освободить. Как вы думаете, что произойдет с переменной экземпляра `name` в `myCard`? Ее поле `name` будет недействительным, поскольку будет ссылаться на объект, который был ликвидирован. Именно поэтому нужно, чтобы наши классы имели свои собственные объекты-члены: эти объекты могут быть неожиданно высвобождены или модифицированы.

В следующих примерах этот вопрос обсуждается более подробно. Начнем с определения нового класса `ClassA`, содержащего одну переменную экземпляра: строковый объект с именем `str`. Напишем метод-установщик и метод-получатель (`setter` и `getter`) для этой переменной. Мы не будем синтезировать эти методы, а напишем их сами, чтобы было ясно, что происходит.

### Программа 17.3

```
// Знакомство с подсчетом ссылок

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>

@interface ClassA: NSObject
{
```

```
NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
@end

@implementation ClassA
-(void) setStr: (NSString *) s
{
 str = s;
}

-(NSString *) str
{
 return str;
}
@end

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSMutableString *myStr = [NSMutableString stringWithString: @"A string"];
 ClassA *myA = [[ClassA alloc] init];

 NSLog (@"myStr retain count: %x", [myStr retainCount]);

 [myA setStr: myStr];
 NSLog (@"myStr retain count: %x", [myStr retainCount]);

 [myA release];
 [pool drain];
 return 0;
}
```

### Выход программы 17.3

```
myStr retain count: 1 (счетчик ссылок myStr)
myStr retain count: 1
```

Программа просто выделяет память (`alloc`) для объекта класса `ClassA` с именем `myA` и затем вызывает метод-установщик (`setStr`), чтобы присвоить ему объект `NSString`, указанный `myStr`. Счетчик ссылок для `myStr` равен 1 как до, так и после вызова метода `setStr` (как и следовало ожидать), поскольку этот метод просто сохраняет значение своего аргумента в своей переменной `str`. И здесь снова, если программа высвободит `myStr` после вызова метода `setStr`, значение, сохраненное внутри переменной экземпляра `str`, будет неверным, поскольку ее счетчик ссылок будет уменьшен до 0 и пространство памяти, занятое объектом, на который она ссылается, будет освобождено.

Это происходит в программе 17.3 при высвобождении autorelease-пула. Мы не добавляли строковый объект myStr в этот пул явным образом, но он был добавлен в autorelease-пул при его создании с помощью метода `stringWithString:`. При высвобождении пула произошло также высвобождение myStr. Поэтому любая попытка доступа к этому объекту после высвобождения пула будет неверной.

В программе 17.4 внесены изменения в метод `setStr:`, чтобы удержать (`retain`) значение str. Это защитит от возможности случайно высвободить ссылки на объект str.

#### Программа 17.4

```
// Удержание объектов

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>

@interface ClassA: NSObject
{
 NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
@end

@implementation ClassA
-(void) setStr: (NSString *) s
{
 str = s;
 [str retain];
}

-(NSString *) str
{
 return str;
}
@end

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *myStr = [NSMutableString stringWithString: @"A string"];
 ClassA *myA = [[ClassA alloc] init];

 NSLog(@"%@", [myStr retainCount]);
```

```
[myA setStr: myStr];
NSLog (@"myStr retain count: %x", [myStr retainCount]);

[myStr release];
NSLog (@"myStr retain count: %x", [myStr retainCount]);

[myA release];
[pool drain];
return 0;
}
```

#### Вывод программы 17.4

```
myStr retain count: 1
myStr retain count: 2
myStr retain count: 1
```

Мы видим, что после вызова метода `setStr:` счетчик ссылок для `myStr` увеличился до 2, что позволило решить эту проблему. Последующее высвобождение `myStr` в этой программе оставляет допустимой ссылку на нее через переменную экземпляра, поскольку ее счетчик ссылок пока равен 1.

Поскольку мы выделили память для `myA` с помощью `alloc`, мы по-прежнему обязаны высвободить ее сами. Но вместо этого мы могли бы добавить ее в `autorelease`-пул, отправив ей сообщение `autorelease`.

```
[myA autorelease];
```

Это можно сделать сразу после выделения памяти для объекта. Напомним, что добавление объекта в `autorelease`-пул не высвобождает его и не делает его недействительным; он просто помечается для дальнейшего высвобождения. Мы можем продолжать использовать этот объект, пока не будет освобождена занимаемая им память, что происходит при высвобождении пула, если счетчик ссылок этого объекта на этот момент стал равным 0.

У нас все еще остаются некоторые потенциальные проблемы, которые вы, возможно, видите. Метод `setStr:` выполняет необходимую работу по удержанию (`retain`) строкового объекта, который он получает в качестве своего аргумента, но когда высвобождается *этот* строковый объект? И что происходит со старым значением переменной экземпляра `str`, которое мы перезаписываем? Нужно ли высвобождать это значение, чтобы освободить занимаемую им память? В программе 17.5 содержится решение этой проблемы.

#### Программа 17.5

```
// Знакомство с подсчетом ссылок

#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSArray.h>
```

```
@interface ClassA: NSObject
{
 NSString *str;
}

-(void) setStr: (NSString *) s;
-(NSString *) str;
-(void) dealloc;
@end

@implementation ClassA
-(void) setStr: (NSString *) s
{
 // высвобождение старого объекта, поскольку мы закончили работать с ним
 [str autorelease];

 // удержание (retain) аргумента на тот случай, если кто-то высвободит его
 str = [s retain];
}

-(NSString *) str
{
 return str;
}

-(void) dealloc {
 NSLog(@"ClassA dealloc");
 [str release];
 [super dealloc];
}
@end

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSString *myStr = [NSMutableString stringWithString: @"A string"];
 ClassA *myA = [[ClassA alloc] init];

 NSLog(@"myStr retain count: %x", [myStr retainCount]);
 [myA autorelease];

 [myA setStr: myStr];
 NSLog(@"myStr retain count: %x", [myStr retainCount]);

 [pool drain];
 return 0;
}
```

### Вывод программы 17.5

```
myStr retain count: 1
myStr retain count: 2
ClassA dealloc
```

Метод `setStr:` берет то, что сохранено на данный момент в переменной экземпляра `str`, и применяет к нему `autorelease`, то есть делает его доступным для дальнейшего высвобождения. Это важно, если метод вызывается много раз, чтобы присваивать одному и тому же полю различные значения. Каждый раз перед сохранением нового значения старое значение должно быть помечено для высвобождения. После высвобождения старого значения новое значение удерживается (`retain`) и сохраняется в поле `str`. В выражении с сообщением

```
str = [s retain];
```

используется тот факт, что метод `retain` возвращает своего получателя.

---

**Примечание.** Если переменная `str` имеет значение `nil`, это не представляет проблемы. Среда выполнения Objective-C инициализирует все переменные экземпляра, присваивая им `nil`, и вполне допустимо передать сообщение `nil`.

Метод `dealloc` уже встречался в главе 15 при работе с классами `AddressBook` и `AddressCard`. Замещающий метод `dealloc` – это удобный способ избавиться от последнего объекта, на который ссылается наша переменная экземпляра `str` при освобождении ее памяти (то есть когда ее счетчик ссылок стал равным 0). В таком случае система вызывает метод `dealloc`, который наследуется из `NSObject` и его обычно не требуется замещать. Если внутри методов происходит удержание (`retain`) объектов, выделение для них памяти (с помощью `alloc`) или их копирование (с помощью методов копирования, описанных в следующей главе), может потребоваться замещение `dealloc`, чтобы иметь возможность их высвобождения. Операторы

```
[str release];
[super dealloc];
```

сначала высвобождают переменную экземпляра `str` и затем вызывают родительский метод `dealloc`, чтобы закончить работу.

Вызов `NSLog` был помещен внутри метода `dealloc`, чтобы выводить сообщение, когда вызывается этот метод. Мы сделали это, чтобы подтвердить, что объект `ClassA` правильно высвобожден после высвобождения `autorelease`-пула.

Возможно, вы увидели один последний недочет в методе-установщике `setStr`. Рассмотрим еще раз программу 17.5. Предположим, что `myStr` является мутабельной строкой (а не немутабельной) и один или несколько символов в `myStr` были изменены после вызова `setStr`. Изменения в строке, на которую ссылается `myStr`, повлияют также на строку, на которую ссылается переменная экземпляра, поскольку они ссылаются на один и тот же объект. Перечитайте последнее предложение, чтобы убедиться, что вы понимаете его. Вы должны также понять, что присваивание `myStr` совершенно нового строкового объекта не вызовет этой проблемы. Проблема возникает только в том случае, если будут изменены каким-либо способом один или несколько символов строки.

Решением этой проблемы является создание новой копии строки внутри метода-установщика, если вы хотите защитить ее и сделать совершенно независимой от аргумента этого метода. Именно поэтому здесь выбрано создание копии компонентов name и email в методах setName: и setEmail: класса AddressCard (см. главу 15).

## Пример автоматического высвобождения

Рассмотрим последний пример из этой главы, чтобы убедиться, что вы действительно понимаете, как действует подсчет ссылок, удержание (retain) и высвобождение/автоматическое высвобождение (release/autorelease) объектов. Рассмотрим программу 17.6, которая определяет фиктивный класс с именем Foo с одной переменной экземпляра и только наследуемыми методами.

### Программа 17.6

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

@interface Foo: NSObject
{
 int x;
}
@end

@implementation Foo
@end

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

 NSLog (@"myFoo retain count = %x", [myFoo retainCount]);

 [pool drain];
 NSLog (@"after pool drain = %x", [myFoo retainCount]);

 pool = [[NSAutoreleasePool alloc] init];
 [myFoo autorelease];
 NSLog (@"after autorelease = %x", [myFoo retainCount]);

 [myFoo retain];
 NSLog (@"after retain = %x", [myFoo retainCount]);

 [pool drain];
 NSLog (@"after second pool drain = %x", [myFoo retainCount]);
```

```
[myFoo release];
return 0;
}
```

### Выход программы 17.6

myFoo retain count = 1 (счетчик ссылок myFoo)

```
after pool drain = 1 (после pool drain)
after autorelease = 1 (после autorelease)
after retain = 2 (после retain)
after second pool drain = 1 (после второго pool drain)
```

Программа выделяет память для нового объекта Foo и присваивает его переменной myFoo. Как вы уже видите, начальное значение ее счетчика удержаний (ссылок) равно 1. Этот объект еще не является частью autorelease-пула, поэтому высвобождение этого пула не делает объект недействительным. Затем выделяется новый пул, и myFoo добавляется в этот пул путем отправки сообщения autorelease. Снова отметим, что счетчик ссылок этой переменной не изменяется, так как добавление объекта в autorelease-пул не влияет на его счетчик ссылок, а только помечает его для дальнейшего высвобождения.

Затем мы передаем myFoo сообщение retain. Это изменяет ее счетчик ссылок на 2. При последующем высвобождении пула во второй раз счетчик ссылок для myFoo уменьшается на 1, так как ранее ей было передано сообщение autorelease и, следовательно, ей передается сообщение release, когда происходит высвобождение пула.

Поскольку для myFoo было выполнено удержание (retain) до высвобождения пула, ее счетчик ссылок после уменьшения будет все еще больше 0. Поэтому myFoo остается после pool drain и все еще является действительным объектом. Конечно, вы должны теперь высвободить и ее, что мы и делаем в программе 17.6 для очистки и ликвидации утечек памяти.

Если вам понятна программа 17.6, значит, вы хорошо понимаете, что такое autorelease-пул и как он действует.

## Сводка правил по управлению памятью

Подытожим сведения по управлению памятью, которые изложены в этой главе.

- Высвобождение объекта может освобождать его память, что может требовать определенных усилий, если вы создаете много объектов во время выполнения программы. Основным правилом является высвобождение созданных или удержанных объектов, когда вы прекращаете работать с ними.
- Передача сообщения release не обязательно означает ликвидацию объекта. Если счетчик ссылок объекта уменьшается до 0, то объект ликвидируется. Система осуществляет это, передавая объекту сообщение dealloc для освобождения его памяти.
- Автоматически высвобождаемый пул (autorelease-пул) предусматривает автоматическое высвобождение объектов, когда высвобождается сам пул. Система делает это, передавая сообщение release каждому объекту столько же

раз, сколько было передано сообщений `autorelease`. Каждому объекту в `autorelease`-пуле, счетчик ссылок которого уменьшился до 0, передается сообщение `dealloc` для ликвидации самого объекта.

- Если вам больше не нужен объект в методе, но нужно вернуть его, передайте ему сообщение `autorelease`, чтобы пометить его для дальнейшего высвобождения. Сообщение `autorelease` не влияет на счетчик ссылок объекта. Это позволяет отправителю сообщения использовать объект, но при этом объект будет высвобожден позже, когда будет высвобожден сам `autorelease`-пул.
- Когда завершается работа приложения, освобождается вся память, занимаемая объектами, независимо от их включения в `autorelease`-пул.
- Если вы разрабатываете более сложные приложения (например, приложения Cocoa), `autorelease`-пулы можно создавать и ликвидировать во время выполнения программы (для приложений Cocoa это происходит каждый раз при возникновении события). В таких случаях можно сделать так, чтобы объект остался действительным после автоматического высвобождения (когда высвобождается сам `autorelease`-пул); для этого нужно явно удержать его (`retain`). Все объекты, счетчик ссылок которых больше числа переданных им сообщений `autorelease`, продолжают действовать после высвобождения пула.
- В случае непосредственного создания объекта с помощью метода `alloc` или `copy` (либо метода `allocWithZone:`, `copyWithZone:` или `mutableCopy`) вы обязаны сами высвободить его. Каждый раз при удержании объекта (с помощью `retain`) вы должны применить `release` или `autorelease` к этому объекту.
- Вам не нужно заботиться о высвобождении объектов, возвращаемых методами, которые не упомянуты в предыдущем правиле. Такие методы будут сами обеспечивать автоматическое высвобождение объектов. Именно поэтому вам нужно в первую очередь создавать `autorelease`-пул в своих программах. Такие методы, как `stringWithString:`, автоматически добавляют в этот пул новые создаваемые строковые объекты, передавая им сообщения `autorelease`. Если не создать пул, то вы получите сообщение, что пытаетесь автоматически высвободить (`autorelease`) объект при отсутствии пула.

## Сборка мусора

До настоящего момента мы создавали программы для выполнения в среде runtime с управлением памятью (*memory-managed environment*). Правила управления памятью, изложенные в предыдущих разделах, применимы к такой среде, где используются `autorelease`-пулы и возникают вопросы высвобождения и удержания объектов, а также владения объектами.

В Objective-C 2.0 стала доступна альтернативная форма управления памятью, которая называется *сборкой мусора* (*garbage collection*). Используя сборку мусора, можно не думать об удержании и высвобождении объектов, `autorelease`-пулах или счетчиках удержаний (ссылок). Система автоматически следит, какие объекты владеют другими объектами, автоматически высвобождает (то есть включает в сборку мусора) объекты, к которым больше нет обращений, по мере того, как возникает потребность в памяти при выполнении программы.

Если все так просто, то почему мы не использовали сборку мусора на протяжении всей этой книги, опустив все вопросы управления памятью? Здесь есть три причины. Во-первых, даже в среде, которая поддерживает сборку мусора, будет лучше, если мы знаем владельцев наших объектов, чтобы определять момент, когда они больше не требуются. Это заставит вас быть более аккуратными при написании кода, поскольку вы будете понимать связи объектов друг с другом и знать продолжительность их действия в программе.

Во-вторых, как уже говорилось выше, среда runtime для iPhone не поддерживает сборку мусора, поэтому у вас нет выбора при разработке программ для этой платформы.

И третья причина относится к вам, если вы планируете писать библиотечные процедуры, встраиваемые модули (plug-in) или код для совместного доступа. Этот код может быть загружен в процесс как со сборкой мусора, так и без сборки мусора, поэтому его следует писать для работы в обеих средах. Это означает, что вы должны использовать методы управления памятью, описанные в этой книге. Это также означает, что вы должны тестировать свой код с отключением и включением сборки мусора.

Если вы решили использовать сборку мусора, то должны включить ее при создании программ с помощью Xcode. Это можно делать через меню Project (Проект), Edit Project Settings (Изменение настроек проекта). В настройках «GCC 4.0-Code Generation» вы увидите пункт Objective-C Garbage Collection (Сборка мусора Objective-C). Измените ее со значения по умолчанию Unsupported (Не поддерживается) на Required (Требуется), чтобы ваша программа создавалась с включением автоматической сборки мусора (см. рис. 17.1).

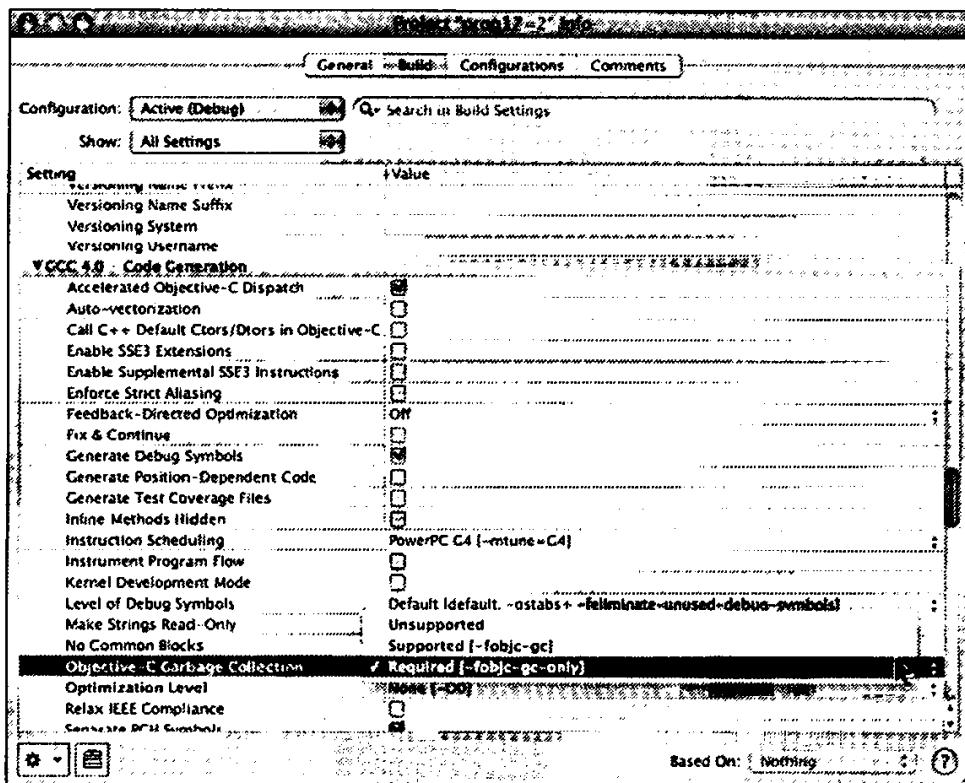


Рис. 17.1. Включение сборки мусора

С включенной сборкой мусора в программе могут по-прежнему использоваться вызовы методов `retain`, `autorelease`, `release` и `dealloc`. Однако все они будут игнорироваться. Это позволит вам разработать программу, которую можно будет выполнять в обеих средах — с управлением памятью или со сборкой мусора. Однако при выполнении в обеих средах вы не сможете работать с собственным методом `dealloc`: как уже говорилось, вызовы `dealloc` игнорируются, если включена сборка мусора.

---

**Примечание.** Способы управления памятью, описанные в этой главе, достаточны для большинства приложений. Однако в более сложных случаях, например, при написании многопотоковых приложений, могут потребоваться дополнительные сведения. О вопросах, относящихся к сборке мусора, см. приложение D.

---

## Упражнения

1. Напишите программу, чтобы проверить, как влияет добавление и удаление записей в словарь на счетчик ссылок объектов, которые вы добавляете и удаляете.
2. Как, по вашему мнению, будет влиять метод `NSArray replaceObjectAtIndex:withObject:` на счетчик ссылок объекта, который заменяется в массиве? Напишите соответствующую программу для проверки. Затем ознакомьтесь с документацией по этому методу, чтобы проверить результаты.
3. Вернемся к классу `Fraction`, с которым мы работали на протяжении части I этой книги. Для вашего удобства он приводится в приложении D. Внесите изменения в этот класс, чтобы работать в Foundation framework. Затем добавьте нужные сообщения в различные методы категории `MathOps`, чтобы добавлять в `autorelease`-пул дроби (`fraction`), получаемые в результате каждой операции. Сможете ли вы использовать после этого следующий оператор без утечки памяти?

`[[fractionA add: fractionB] print];`

Объясните ответ.

4. Вернемся к примерам `AddressBook` и `AddressCard` из главы 15. Внесите изменения в каждый метод `dealloc`, чтобы выводить сообщение при вызове метода. Затем выполните некоторые примеры программ, где используются эти классы, чтобы убедиться, что сообщение `dealloc` передается каждому объекту `AddressBook` и `AddressCard`, прежде чем будет достигнут конец `main`.
5. Выберите любые две программы из этой книги и выполните их в Xcode с включенной сборкой мусора. Убедитесь, что в этом случае игнорируются вызовы таких методов, как `retain`, `autorelease` и `release`.

## Глава 18

# Копирование объектов

В этой главе рассматриваются некоторые особенности копирования объектов. Мы рассмотрим понятия поверхностного (shallow) и глубокого (deep) копирования и опишем, как создавать копии в Foundation framework.

В главе 8 было показано, что происходит, если мы присваиваем один объект другому с помощью простого оператора присваивания, например

```
origin = pt;
```

В этом примере `origin` и `pt` – это объекты класса `XYPoint`, которые определены следующим образом.

```
@interface XYPoint: NSObject
{
 int x;
 int y;
};

...
@end
```

Напомним, что присваивание – это просто копирование адреса объекта `pt` в `origin`. После операции присваивания обе переменные указывают одно и то же место в памяти. Внесение изменений в переменные экземпляра с помощью сообщения

```
[origin setX: 100 andY: 200];
```

приводит к изменению координат `x,y` объекта класса `XYPoint`, на который ссылаются обе переменные (`origin` и `pt`), поскольку они указывают на один и тот же объект в памяти.

То же самое относится к объектам Foundation: присваивание одной переменной другой вызывает создание еще одной ссылки на объект (но приводит к наращиванию счетчика ссылок, см. главу 17). Например, если `dataArray` и `dataArray2` – объекты класса `NSMutableArray`, то следующие операторы удаляют первый элемент из одного и того же массива, на который ссылаются обе эти переменные.

```
dataArray2 = dataArray;
[dataArray2 removeObjectAtIndex: 0];
```

## Методы copy и mutableCopy

В классах Foundation реализованы методы `copy` и `mutableCopy`, предназначенные для создания копии объекта. Для создания копий методы реализуются в соответствии с протоколом `<NSCopying>`. Если для вашего класса требуется различать создание мутабельных и немутабельных копий объекта, то методы реализуются согласно протоколу `<NSMutableCopying>`.

Вернемся к методам копирования в классах Foundation для двух объектов `dataArray2` и `dataArray` класса `NSMutableArray`. С помощью оператора

```
dataArray2 = [dataArray mutableCopy];
```

создается новая копия массива `dataArray` в памяти с дублированием всех его элементов. Поэтому оператор

```
[dataArray2 removeObjectAtIndex: 0];
```

удаляет первый элемент из массива `dataArray2`, но не из массива `dataArray`.

Это показано в программе 18.1.

### Программа 18.1

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
 @"one", @"two", @"three", @"four", nil];
 NSMutableArray *dataArray2;

 // простое присваивание

 dataArray2 = dataArray;
 [dataArray2 removeObjectAtIndex: 0];

 NSLog(@"%@", dataArray);
 for (NSString *elem in dataArray)
 NSLog(@"%@", elem);

 NSLog(@"%@", dataArray2);

 for (NSString *elem in dataArray2)
 NSLog(@"%@", elem);

 // копирование, затем удаление первого элемента из копии
```

```
dataArray2 = [dataArray mutableCopy];
[dataArray2 removeObjectAtIndex: 0];

NSLog(@"%@", dataArray);

for (NSString *elem in dataArray)
 NSLog(@"%@", elem);

NSLog(@"%@", dataArray2);

for (NSString *elem in dataArray2)
 NSLog(@"%@", elem);

[dataArray2 release];
[pool drain];
return 0;
}
```

### Вывод программы 18.1

```
dataArray:
two
three
four
dataArray2:
two
three
four
dataArray:
two
three
four
dataArray2:
three
four
```

В программе определяется объект мутабельного массива `dataArray`, и его элементам присваиваются строковые объекты `@"one"`, `@"two"`, `@"three"`, `@"four"` соответственно.

Как говорилось выше, оператор присваивания

```
dataArray2 = dataArray;
```

просто создает еще одну ссылку на тот же объект массива в памяти. Поэтому после удаления первого объекта из `dataArray2` и вывода элементов объектов-массивов первый элемент (строка `@"one"`) исчезает из обеих ссылок на этот объект-массив.

Затем мы создаем мутабельную копию `dataArray` и присваиваем полученную копию массиву `dataArray2`. В результате получаются два отдельных мутабельных

массива в памяти, каждый из которых содержит три элемента. Теперь удаление первого элемента из dataArray2 не оказывает влияния на содержимое массива dataArray, что подтверждается последними двумя строками вывода программы.

Отметим, что для создания мутабельной копии объекта копируемый объект не обязан быть мутабельным. То же относится и к немутабельным копиям: можно сделать немутабельную копию мутабельного объекта.

При создании копии массива операция копирования автоматически наращивает счетчик ссылок (удержаний) для каждого элемента массива. Поэтому после создания копии массива и последующего высвобождения (`release`) исходного массива элементы копии продолжают действовать.

Но поскольку копия массива dataArray была создана в этой программе с помощью метода `mutableCopy`, вы обязаны сами освободить его память. Как рассказывалось в предыдущей главе, вы обязаны сами высвобождать объекты, которые создали с помощью одного из методов копирования, поэтому в конце программы 18.1 вставлена строка

```
[dataArray2 release];
```

## Поверхностное и глубокое копирование

В программе 18.1 элементы массива dataArray заполняются немутабельными строками (напомним, что константные строковые объекты являются немутабельными). В программе 18.2 мы будем заполнять его мутабельными строками, чтобы можно было изменить одну из строк в этом массиве. Просмотрите программу 18.2 и постарайтесь понять ее вывод.

### Программа 18.2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
 [NSMutableString stringWithString: @"one"],
 [NSMutableString stringWithString: @"two"],
 [NSMutableString stringWithString: @"three"],
 nil
];
 NSMutableArray *dataArray2;
 NSMutableString *mStr;

 NSLog(@"dataArray: ");
 for (NSString *elem in dataArray)
```

```
 NSLog(@"%@", elem);

// создание копии, затем изменение одной из строк

dataArray2 = [dataArray mutableCopy];

mStr = [dataArray objectAtIndex: 0];
[mStr appendString: @"ONE"];

NSLog(@"dataArray: ");
for (NSString *elem in dataArray)
 NSLog(@"%@", elem);

NSLog(@"dataArray2: ");
for (NSString *elem in dataArray2)
 NSLog(@"%@", elem);

[dataArray2 release];
[pool drain];
return 0;
}
```

### Вывод программы 18.2

```
dataArray:
one
two
three
dataArray:
oneONE
two
three
dataArray2:
oneONE
two
three
```

Мы считываем первый элемент массива dataArray с помощью оператора

```
mStr = [dataArray objectAtIndex: 0];
```

и добавляем в него строку @"ONE" с помощью оператора

```
[mStr appendString: @"ONE"];
```

Обратите внимание на значение первого элемента исходного массива и его копии: в обоих массивах оно было изменено. Понятно, почему был изменен первый элемент dataArray, но почему была изменена и его копия? Получая элемент из коллекции, мы получаем новую ссылку на этот элемент, но не новую копию. Поэтому при вызове метода objectAtIndex: для dataArray возвращаемый объект

указывает на тот же объект в памяти, что и первый элемент в `dataArray`. Последующее изменение строкового объекта `mStr` также сопровождается изменением первого элемента массива `dataArray`, что подтверждается результатами вывода.

Но почему изменился первый элемент созданной копии? По умолчанию копии являются *поверхностными* (*shallow*) копиями. Когда массив был скопирован с помощью метода `mutableCopy`, в памяти было выделено пространство для нового объекта-массива, и элементы были скопированы в новый массив. Но копирование каждого элемента массива из исходного места в новое означает только копирование ссылки из одного элемента массива в другой. В результате элементы обоих массивов ссылаются на одни и те же строки в памяти, что не отличается от присваивания одного объекта другому, о котором мы говорили в начале этой главы.

Чтобы создать другие копии каждого элемента массива, необходимо выполнить *глубокое* (*deep*) копирование, при котором создаются копии содержимого каждого объекта в массиве, а не копируются ссылки на объекты (подумайте, что это означает, если элемент массива сам является объектом-массивом). Но глубокое копирование не выполняется по умолчанию, если мы используем методы `copy` или `mutableCopy` с классами Foundation. В главе 19 мы покажем возможности архивации Foundation для создания глубокой копии объекта.

Копируя массив, словарь или набор, мы получаем новую копию этих коллекций. Создание копий отдельных элементов может потребоваться, например, если нужно внести изменения в коллекцию, но не в ее копию. Например, если в программе 18.2 нужно было бы изменить первый элемент массива `dataArray2`, но не `dataArray`, вы могли бы создать новую строку (например, с помощью метода `stringWithString:`) и сохранить ее в первом элементе `dataArray2` с помощью оператора

```
mStr = [NSMutableString stringWithFormat: [dataArray2 objectAtIndex: 0]];
```

Затем можно было бы внести изменения в переменную `mStr` и добавить ее в этот массив с помощью метода `replaceObjectAtIndex:withObject:`:

```
[mStr appendString:@"ONE"]; [dataArray2 replaceObjectAtIndex: 0 withObject: mStr];
```

Даже после замены объекта `mStr` и первый элемент `dataArray2` ссылаются на один и тот же объект в памяти. Поэтому последующее изменение в `mStr` вызовет также изменение первого элемента этого массива. Чтобы избежать этого, высвободите (`release`) `mStr` и выделите память (`alloc`) для нового экземпляра, поскольку метод `replaceObjectAtIndex:withObject:` автоматически удерживает объект.

## Реализация протокола <NSCopying>

Если применить метод `copy` к одному из ваших собственных классов, например, к вашей адресной книге (*address book*), как в строке

```
NewBook = [myBook mutableCopy];
```

то будет выдано сообщение об ошибке:

```
*** -[AddressBook copyWithZone:]: selector not recognized (селектор не распознан)
*** Uncaught exception: (Невыявленное исключение)
*** -[AddressBook copyWithZone:]: selector not recognized
```

Как уже говорилось, для реализации копирования с вашими собственными классами необходимо реализовать один или два метода согласно протоколу <NSCopying>.

Теперь покажем, как добавить метод `copy` в класс *Fraction*, который много использовался в части I. Эти способы вполне применимы для ваших собственных классов. Если эти классы являются подклассами любого из классов Foundation, то потребуется реализация более сложной стратегии копирования, поскольку в суперклассе может быть уже реализована его собственная стратегия копирования.

Напомним, что класс *Fraction* содержит две целые переменные экземпляра: *numerator* (числитель) и *denominator* (знаменатель). Чтобы создать копию одного из этих объектов, необходимо выделить пространство для новой дроби (*fraction*) и затем скопировать значения этих переменных в эту новую дробь.

При реализации протокола <NSCopying> ваш класс должен реализовать метод `copyWithZone:`, чтобы реагировать на сообщение `copy`. (Сообщение `copy` просто передает сообщение `copyWithZone:` в ваш класс с аргументом `nil`.) Если вам нужно отличать мутабельные и немутабельные копии, то потребуется также реализовать метод `mutableCopyWithZone:` согласно протоколу <NSMutableCopying>. Если вы реализуете оба метода, то `copyWithZone:` будет возвращать немутабельную копию, а `mutableCopyWithZone:` будет возвращать мутабельную копию. Создание мутабельной копии объекта не требует, чтобы копируемый объект был тоже мутабельным, и наоборот; вполне возможно, что может требоваться мутабельная копия немутабельного объекта (например, строкового объекта).

Директива `@interface` должна выглядеть следующим образом.

```
@interface Fraction: NSObject <NSCopying>
```

*Fraction* – это подкласс *NSObject*, полчинающийся протоколу *NSCopying*.

В файле секции `implementation Fraction.m` добавьте следующее определение для нового метода.

```
- (id) copyWithZone: (NSZone *) zone
{
 Fraction *newFract = [[Fraction allocWithZone: zone] init];

 [newFract setTo: numerator over: denominator];
 return newFract;
}
```

Аргумент `zone` применяется для разных зон памяти, которые вы можете выделить для работы с программой. Это требуется только в том случае, если ваши приложения занимают много памяти, и вы хотите оптимизировать выделение памяти, группируя ее по зонам. Вы можете брать значение, передаваемое методу `copyWithZone:`, и передавать его методу выделения памяти `allocWithZone:`. Этот метод выделяет память в указанной зоне.

После выделения памяти для нового объекта класса `Fraction` в него копируются переменные получателя `numerator` и `denominator`. Предполагается, что метод `copyWithZone:` будет возвращать новую копию объекта, которую вы создаете в своем методе.

Этот новый метод проверяется в программе 18.3.

### Программа 18.3

```
// Копирование дробей

#import "Fraction.h"
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Fraction *f1 = [[Fraction alloc] init];
 Fraction *f2;

 [f1 setTo: 2 over: 5];
 f2 = [f1 copy];

 [f2 setTo: 1 over: 3];

 [f1 print];
 [f2 print];

 [f1 release];
 [f2 release];
 [pool drain];
 return 0;
}
```

**Вывод программы 18.3**

2/5

1/3

Эта программа создает объект класса Fraction с именем f1 и присваивает ему значение 2/5. Затем вызывается метод copy для создания копии, который передает сообщение copyWithZone: этому объекту. Этот метод создает новый объект класса Fraction, копирует в него значения из f1 и возвращает результат. После возврата в main этот результат присваивается f2. Последующее присваивание f2 дроби 1/3 подтверждает, что это не оказывает влияния на исходную дробь f1. Измените строку программы

`f2 = [f1 copy];`

на

`f2 = f1;`

и удалите высвобождение (release) f2 в конце программы, после чего увидите другие результаты.

Если ваш класс может быть подклассом, то метод copyWithZone: будет наследоваться. В таком случае вы должны изменить строку этого метода

`Fraction *newFract = [[Fraction allocWithZone: zone] init];`

на строку

`Fraction *newFract = [[[self class] allocWithZone: zone] init];`

Это позволяет выделить память для нового объекта из данного класса, который является получателем копии. (Например, если это подкласс с именем NewFraction, то нужно выделить в наследуемом методе память для нового объекта NewFraction вместо объекта Fraction.)

Если вы пишете метод copyWithZone: для класса, суперкласс которого тоже реализует протокол <NSCopying>, то должны сначала вызвать метод copy в этом суперклассе для копирования наследуемых переменных экземпляра и затем включить ваш собственный код для копирования любых дополнительных переменных экземпляра, которые вы могли добавить в этот класс.

Вы должны решить, какое копирование нужно реализовать в вашем классе: поверхностное или глубокое. Задокументируйте это для других пользователей вашего класса.

## Копирование объектов в методах-установщиках и методах-получателях

Каждый раз, реализуя метод-установщик (*setter*) или метод-получатель (*getter*), вы должны продумать, что будет сохраняться в переменных экземпляра, что будет считываться и нужно ли защитить эти значения. Рассмотрим оператор, в котором мы задаем имя одного из объектов *AddressCard* с помощью метода *setName*:

```
[newCard setName: newName];
```

Предположим, что *newName* — это строковый объект, содержащий имя новой карточки. Предположим также, что внутри процедуры установщика мы просто присваиваем параметр соответствующей переменной экземпляра.

```
-(void) setName: (NSString *) theName
{
 name = theName;
}
```

Если затем в программе будут изменены некоторые символы, содержащиеся в *newName*, то одновременно будет изменено соответствующее поле адресной карточки, поскольку это ссылка на один и тот же строковый объект.

Чтобы исключить этот побочный эффект, следует создать копию объекта в процедуре установщика. С помощью метода *alloc* мы создали новый строковый объект и затем с помощью метода *initWithString:* присвоили ему значение параметра, переданного методу.

Можно также написать версию метода *setName:*, чтобы использовать *copy*, например

```
- (void) setName: (NSString *) theName { name = [theName copy]; }
```

Чтобы управлять памятью в процедуре установщика было удобно, необходимо сначала автоматически высвободить (*autorelease*) старое значение, как показано ниже.

```
- (void) setName: (NSString *) theName
{
 [name autorelease];
 name = [theName copy];
}
```

Если задать атрибут *copy* в объявлении свойств (*property*) для переменной экземпляра, то в синтезируемом методе будет использоваться метод класса *copy* (написанный вами или унаследованный). Поэтому объявление

```
@property (nonatomic, copy) NSString *name;
```

приведет к созданию синтезируемого метода, который действует следующим образом.

```
- (void) setName: (NSString *) theName
{
```

```
if (theName != name) {
 [name release]
 name = [theName copy];
}
}
```

Атрибут `nonatomic` указывает системе, что в данном случае не нужно защищать методы доступа с помощью блокировки `mutex` (*mutually exclusive* – взаимоисключение). При написании кода с защищкой потоков используются блокировки `mutex`, чтобы исключить одновременное выполнение одного кода двумя потоками (ситуация, которая часто вызывает ужасные проблемы). Но эти блокировки могут замедлять выполнение программ, поэтому вы можете отказаться от них, если знаете, что этот код будет всегда выполняться только в одном потоке.

Если атрибут `nonatomic` не указан или вместо него указан атрибут `atomic` (это атрибут по умолчанию), то переменная экземпляра будет защищена блокировкой `mutex`. Кроме того, синтезируемый метод-получатель будет удерживать (`retain`) и автоматически высвобождать (`autorelease`) переменную экземпляра перед тем, как будет возвращено ее значение. В среде без сборки мусора это защищает переменную экземпляра от возможной перезаписи методом-установщиком, который высвобождает старое значение переменной экземпляра, прежде чем установить новое значение. Использование `retain` в методе-получателе гарантирует, чтобы память для старого значения не будет освобождена.

---

**Примечание.** Проблема высвобождения и автоматического высвобождения (`retain/autorelease`) не актуальна в среде со сборкой мусора, в которой вызовы этих методов игнорируются, но это не относится к блокировке `mutex`. Если ваш код будет выполняться в многопотоковой среде, предусмотрите использование методов доступа с атрибутом `atomic`.

---

То же самое можно сказать о защите значения переменных экземпляра в процедурах получателей. Если в такой процедуре возвращается объект, то вы должны обеспечить, чтобы изменения, вносимые в возвращаемое значение, не повлияли на значение ваших переменных экземпляра. В таких случаях можно создать копию переменной экземпляра и возвращать эту копию.

Вернемся к реализации метода `copy`. Если мы копируем переменные экземпляра, которые содержат немутабельные объекты (например, немутабельные структурные объекты), то создание новой копии содержимого объекта, возможно, не потребуется. Достаточно создать новую ссылку на объект путем его удержания (`retain`). Например, если мы реализуем метод `copy` для класса `AddressCard` с членами `name` и `email`, то достаточно написать следующую реализацию для `copyWithZone`:

```
- (AddressCard *) copyWithZone: (NSZone *) zone
{
 AddressCard *newCard = [[AddressCard allocWithZone: zone] init];
 [newCard retainName: name andEmail: email];
 return newCard;
}
```

```
- (void) retainName: (NSString *) theName andEmail: (NSString *) theEmail
{
 name = [theName retain];
 email = [theEmail retain];
}
```

Здесь для копирования переменных экземпляра не используется метод `setName:andEmail:`, поскольку он создает новые копии своих аргументов. Вместо этого мы просто удерживаем (`retain`) две переменные с помощью нового метода `retainName:andEmail:`. (Мы могли бы задавать эти две переменные экземпляра в `newCard` непосредственно внутри метода `copyWithZone:`, но для этого нужны операции с указателями, без которых мы обходились до настоящего момента. Операции с указателями были бы более эффективны и не показывали бы пользователю этого класса метод `[retainName:andEmail:]`, не предназначенный для общего пользования, поэтому в какой-то момент вам, возможно, придется этому научиться — но не сейчас!)

В данном случае достаточно использовать удержание (`retain`) для переменных экземпляра вместо создания их полных копий, поскольку владелец скопированной карточки не может повлиять на компоненты `name` и `email` исходной карточки. Убедитесь сами, что это действительно так (подсказка: он должен работать с методами-установщиками).

## Упражнения

1. Реализуйте метод `copy` для класса `AddressBook` согласно протоколу `NSCopying`. Имеет ли смысл реализовать также метод `mutableCopy`? Почему?
2. Внесите изменения в классы `Rectangle` и `XYPoint`, определенные в главе 8, чтобы они подчинялись протоколу `<NSCopying>`. Добавьте в оба класса метод `copyWithZone:`. Сделайте так, чтобы в `Rectangle` выполнялось копирование его члена `XYPoint origin` с помощью метода `XYPoint copy`. Имеет ли смысл реализовать как мутабельную, так и немутабельную копию для этих классов? Объясните.
3. Создайте объект-словарь `NSDictionary` и заполните его несколькимиарами ключ/объект. Затем создайте мутабельную и немутабельную копии. Это глубокие или поверхностные копии? Проверьте свой ответ.
4. Кто обязан освобождать память, выделяемую для новой адресной карточки (`AddressCard`) в методе `copyWithZone:`, если выполнена реализация, как в этой главе? Почему?

## Глава 19

# Архивация

В терминологии Objective-C архивация – это процесс сохранения одного или нескольких объектов в формате, позволяющем восстановить их в дальнейшем. Часто при этом объекты записываются в файл, чтобы их можно было прочитать. Мы рассмотрим в этой главе два метода архивации данных: *списки свойств* (*property list*) и *кодирование с ключами* (*key-valued coding*).

## Архивация со списками свойств XML

В приложениях Mac OS X используются списки свойств XML (*propertylist* или *plists*) для сохранения такой информации, как настройки по умолчанию, настройки приложений и данные конфигурации, поэтому вам будет полезно узнать, как их создавать и считывать. Однако их использование в целях архивации ограничено, поскольку при создании списка свойств для структуры данных конкретные классы объектов не удерживаются, информация о нескольких ссылках на один объект и мутабельность объекта не сохраняются.

---

**Примечание.** Формат в списках свойств в так называемом «старом стиле» отличается от формата списков свойств XML. По возможности старайтесь придерживаться списков свойств XML.

---

При записи данных в файл для объектов типа `NSString`, `NSDictionary`, `NSArray`, `NSDate`, `NSData` или `NSNumber` можно использовать реализованный в этих классах метод `writeToFile:atomically:`. При записи словаря или массива этот метод записывает данные в файл в формате списка свойств XML. В программе 19.1 показано, как записать в файл в виде списка свойств словарь, созданный в главе 15.

### Программа 19.1

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSAutoreleasePool.h>
int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSDictionary *glossary =
 [NSDictionary dictionaryWithObjectsAndKeys:
```

```

 @"A class defined so other classes can inherit from it.", @"abstract class",
 @"To implement all the methods defined in a protocol", @"adopt",
 @"Storing an object for later use.", @"archiving",
 nil
];
if ([glossary writeToFile: @"glossary" atomically: YES
 encoding: NSUTF8StringEncoding error: nil] == NO)
 NSLog(@"Save to file failed!");
[pool drain];
return 0;
}

```

Сообщение `writeToFile:atomically:encoding:error:` передается объекту-словарю `glossary`, что вызывает запись этого словаря в файл `glossary` в виде списка свойств. Параметру `atomically` присваивается значение `YES`, указывая, что операцию записи нужно выполнять сначала во временный резервный файл; если эта запись выполнена успешно, данные окончательно перемещаются в указанный файл с именем `glossary`. Эта мера защищает файл от повреждения, например, при сбое системы во время записи. В этом случае прежний файл `glossary` (если он уже существовал) не будет поврежден.

При просмотре содержимого файла `glossary`, созданного программой 19.1, мы увидим следующее.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>abstract class</key>
 <string>A class defined so other classes can inherit from it.</string>
 <key>adopt</key>
 <string>To implement all the methods defined in a protocol</string>
 <key>archiving</key>
 <string>Storing an object for later use. </string>
</dict>
</plist>

```

XML-файл, созданный для этого словаря, содержит набор из пар ключей (`<key>...</key>`) и значений (`<string>...</string>`).

При создании списка свойств из словаря все ключи в этом словаре должны быть объектами `NSString`. Элементами массива или значениями в словаре могут быть объекты типа `NSString`, `NSArray`, `NSDictionary`, `NSData`, `NSDate` или `NSNumber`.

Для считывания данных используйте метод `dataWithContentsOfFile::`; для считывания строковых объектов используйте метод `stringWithContentsOfFile::`. В программе 19.2 выполняется считывание словаря, записанного в программе 19.1, и последующий вывод его содержимого.

### Программа 19.2

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSEnumerator.h>
#import <Foundation/NSAutoreleasePool.h>

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

 glossary = [NSDictionary dictionaryWithContentsOfFile: @"glossary"];

 for (NSString *key in glossary)
 NSLog(@"%@", key, [glossary objectForKey: key]);
 [pool drain];
 return 0;
}
```

### Вывод программы 19.2

archiving: Storing an object for later use

(архивация: сохранение объекта для дальнейшего использования)

abstract class: A class defined so other classes can inherit from it

(абстрактный класс: Класс, определенный таким образом, чтобы другие классы могли наследовать из него)

adopt: To implement all the methods defined in a protocol

(принять: для реализации всех методов, определенных в протоколе)

Ваши списки свойств не обязательно должны создаваться из программы на Objective-C; список свойств может поступать из любого источника. Вы можете создавать свои собственные списки свойств с помощью простого текстового редактора или программы Property List Editor (Редактор списков свойств), которая находится в /Developer/Applications/Utilities на компьютерах с системой Mac OS X.

## Архивация с помощью NSKeyedArchiver

В файле можно сохранять объекты любого типа, а не только строки, массивы и словари. Для этого необходимо создать архив с *ключами* (*keyed archive*) с помощью класса NSKeyedArchiver.

Mac OS X поддерживает архивы с ключами, начиная с версии 10.2. В ранних версиях с помощью класса NSArchiver создавались *последовательные* архивы (*sequential archives*). Данные последовательных архивов должны считываться точно в том же порядке, в каком они записывались.

В архиве с ключами каждое поле архива имеет имя. При архивации объекта мы присваиваем ему имя, или *ключ*. При считывании объекта из архива мы используем тот же ключ. Это позволяет записывать объекты в архив и считывать

их в любом порядке. Кроме того, если в классе добавляются или удаляются новые переменные экземпляра, это можно предусмотреть в программе.

Отметим, что `NSArchiver` недоступен в iPhone SDK. Если нужна архивация в iPhone, вы должны использовать `NSKeyedArchiver`.

Чтобы использовать архивы с ключами, нужно импортировать `<Foundation/NSKeyedArchiver.h>`.

В программе 19.3 показано, как сохранять файл на диске с помощью метода `archiveRootObject:toFile:` из класса `NSKeyedArchiver`. Чтобы использовать этот класс, нужно включить в программу соответствующий файл с помощью оператора

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

### Программа 19.3

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSDictionary *glossary =
 [NSDictionary dictionaryWithObjectsAndKeys:
 @{@"A class defined so other classes can inherit from it",
 @"abstract class",
 @"To implement all the methods defined in a protocol",
 @"adopt",
 @"Storing an object for later use",
 @"archiving",
 nil
 }];
 [NSKeyedArchiver archiveRootObject: glossary toFile: @"glossary.archive"];

 [pool release];
 return 0;
}
```

Программа 19.3 не выводит никаких данных на терминал. Однако оператор

```
[NSKeyedArchiver archiveRootObject: glossary toFile: @"glossary.archive"];
```

записывает словарь `glossary` в файл `glossary.archive`. Для этого файла можно задать любой путь. В данном случае файл записывается в текущую папку.

Этот архивный файл можно в дальнейшем читать в программу с помощью метода NSKeyedUnarchiver unArchiveObjectWithFile:, как показано в программе 19.4.

#### Программа 19.4

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSEnumerator.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>

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

 glossary = [NSKeyedUnarchiver unarchiveObjectWithFile:
 @"glossary.archive"];

 for (NSString *key in glossary)
 NSLog (@"%@: %@", key, [glossary objectForKey: key]);

 [pool drain];
 return 0;
}
```

#### Вывод программы 19.4

abstract class: A class defined so other classes can inherit from it.

adopt: To implement all the methods defined in a protocol

archiving: Storing an object for later use.

#### Оператор

```
glossary = [NSKeyedUnarchiver unarchiveObjectWithFile: @"glossary.archive"];
```

вызывает открытие указанного файла и считывание его содержимого. Файл должен быть создан в результате архивации, для него можно указывать полное или относительное имя, как в данном примере.

После восстановления словаря программа просто выполняет перебор его содержимого, чтобы убедиться, что восстановление было успешным.

## Написание методов кодирования и декодирования

Объекты базовых классов Objective-C, таких как `NSString`, `NSArray`, `NSDictionary`, `NSSet`, `NSDate`, `NSNumber` и `NSData`, можно архивировать и восстанавливать, как описано в предыдущем разделе. Это относится и к объектам с вложенностью, например, массивам, содержащим строки или другие объекты-массивы.

Мы не можем непосредственно архивировать нашу адресную книгу (`AddressBook`) с помощью этих средств, поскольку система Objective-C «не знает», как архивировать объект класса `AddressBook`. Если попытаться архивировать его в программе с помощью строки

```
[NSKeyedArchiver archiveRootObject: myAddressBook toFile: @"addrbook.arch"];
```

то при выполнении программы будет выведено сообщение

```
*** -[AddressBook encodeWithCoder:]: selector not recognized (селектор не распознан)
*** Uncaught exception: <NSInvalidArgumentException> (Невыявленное исключение)
***-[AddressBook encodeWithCoder:]: selector not recognized
archiveTest: received signal: Trace/BPT trap
```

Из этих сообщений об ошибках мы видим, что система пыталась найти метод с именем `encodeWithCoder:` в классе `AddressBook`, но мы нигде не определяли такой метод.

Чтобы архивировать объекты, отличные от списка в начале раздела, мы должны указать системе, как эти объекты архивировать, или *кодировать* (*encode*), а также как их разархивировать, или *декодировать* (*decode*). Для этого нужно добавить в определения класса методы `encodeWithCoder:` и `initWithCoder:` согласно протоколу `<NSCoding>`. В примере с адресной книгой нужно добавить эти методы в два класса: `AddressBook` и `AddressCard`.

Метод `encodeWithCoder:` вызывается каждый раз, когда архиватору нужно кодировать объект из указанного класса, и этот метод указывает ему, как это сделать. Аналогичным образом, метод `initWithCoder:` вызывается каждый раз, когда нужно декодировать объект указанного класса.

В общем случае метод кодирования должен указывать, как архивировать каждую переменную экземпляра в объекте, который нужно сохранить. Для этого есть вспомогательные средства. Для описанных выше базовых классов Objective-C можно использовать метод `encodeObject:forKey:`. Для базовых типов данных C (например, целых и с плавающей точкой) используется один из методов, приведенных в таблице 19.1. Метод декодирования (`decoder`), `initWithCoder:`, действует в обратном порядке: мы используем `decodeObjectForKey:` для декодирования базовых классов Objective-C и подходящий метод декодирования из таблицы 19.1 для базовых типов данных C.

**Таблица 19.1.** Кодирование и декодирование базовых типов данных в архивах с ключами

| Метод кодирования    | Метод декодирования  |
|----------------------|----------------------|
| encodeBool:forKey:   | decodeBool:forKey:   |
| encodeInt:forKey:    | decodeInt:forKey:    |
| encodeInt32:forKey:  | decodeInt32:forKey:  |
| encodeInt64: forKey: | decodeInt64:forKey:  |
| encodeFloat:forKey:  | decodeFloat:forKey:  |
| encodeDouble:forKey: | decodeDouble:forKey: |

В программе 19.5 в классы AddressCard и AddressBook добавлены методы кодирования и декодирования.

#### Программа 19.5. Файл секции interface Addresscard.h

```
#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>

@interface AddressCard: NSObject <NSCoding, NSCopying>
{
 NSString *name;
 NSString *email;
}

@property (copy, nonatomic) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;
-(void) print;

// Дополнительные методы для протокола NSCopying

-(AddressCard *) copyWithZone: (NSZone *) zone;
-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail;

@end
```

Следующие два метода, которые используются для класса AddressCard, должны быть добавлены в файл секции implementation.

```
- (void) encodeWithCoder: (NSCoder *) encoder
{
 [encoder encodeObject: name forKey: @"AddressCardName"];
 [encoder encodeObject: email forKey: @"AddressCardEmail"];
}
```

```

-(id) initWithCoder: (NSCoder *) decoder
{
 name = [[decoder decodeObjectForKey: @"AddressCardName"] retain];
 email = [[decoder decodeObjectForKey: @"AddressCardEmail"] retain];

 return self;
}

```

Метод кодирования `encodeWithCoder:` передается объекту  `NSCoder` в качестве его аргумента. Поскольку класс `AddressCard` наследует непосредственно из `NSObject`, нам не нужно заботиться о кодировании наследуемых переменных экземпляра. Если суперкласс вашего класса согласуется с протоколом `NSCoding`, то, чтобы обеспечить кодирование своих наследуемых переменных экземпляра, вы должны начать метод кодирования со строки

```
[super encodeWithCoder: encoder];
```

Наша адресная книга содержит две переменные экземпляра с именами `name` и `email`. Поскольку это объекты класса `NSString`, мы можем использовать метод `encodeObject:forKey:` для кодирования каждой из них по порядку. Затем эти переменные экземпляра добавляются в архив.

Метод `encodeObject:forKey:` кодирует объект и сохраняет его с указанным ключом для последующего считывания с помощью этого ключа. Имена ключей можно задавать произвольно, для считывания (декодирования) данных нужно использовать тот же ключ, который использовался для их архивации (кодирования). Конфликт может возникнуть только в том случае, если тот же ключ используется для подкласса кодируемого объекта. Чтобы не возникла эта ситуация, можно вставить имя класса перед именем переменной экземпляра, когда вы составляете ключ для архива, как это сделано в программе 19.5.

Отметим, что `encodeObject:forKey:` можно использовать для любого объекта, в классе которого имеется соответствующий реализованный метод `encodeWithCoder:`.

Процесс декодирования действует в обратном порядке. Аргумент, передаваемый `initWithCoder:`, тоже является объектом  `NSCoder`. Вам не нужно заботиться об этом аргументе; он получает сообщения для каждого объекта, который вы хотите извлечь из архива.

Поскольку в данном случае класс `AddressCard` наследует непосредственно из `NSObject`, вам не нужно заботиться о декодировании наследуемых переменных экземпляра. Достаточно вставить следующую строку в начало вашего метода декодирования (`decoder`), если ваш класс согласуется с протоколом `NSCoding`.

```
self = [super initWithCoder: decoder];
```

Каждая переменная экземпляра затем декодируется путем вызова метода `decodeObjectForKey:` и передачи того же ключа, который использовался для кодирования этой переменной.

Аналогично классу `AddressCard`, мы добавляем методы кодирования и декодирования в класс `AddressBook`. В файле секции `interface` нужно только изменить строку с директивой `@interface`, чтобы объявить, что теперь с протоколом `NSCoding` согласуется класс `AddressBook`. Это изменение может выглядеть следующим образом.

```
@interface AddressBook: NSObject <NSCoding, NSCopying>
```

Ниже определяются методы для включения в файл секции **implementation**.

```
- (void) encodeWithCoder: (NSCoder *) encoder
{
 [encoder encodeObject: bookName forKey: @"AddressBookBookName"];
 [encoder encodeObject: book forKey: @"AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
 bookName = [[decoder decodeObjectForKey: @"AddressBookBookName"] retain];
 book = [[decoder decodeObjectForKey: @"AddressBookBook"] retain];

 return self;
}
```

Программа 19.6 – это тестовая программа.

#### Программа 19.6. Тестовая программа

```
#import «AddressBook.h»
#import <Foundation/NSAutoreleasePool.h>

int main (int argc, char *argv[])
{
 NSString *aName = @"Julia Kochan";
 NSString *aEmail = @"jewls337@axlc.com";
 NSString *bName = @"Tony Iannino";
 NSString *bEmail = @"tony.iannino@techfitness.com";
 NSString *cName = @"Stephen Kochan";
 NSString *cEmail = @"steve@steve_kochan.com";
 NSString *dName = @"Jamie Baker";
 NSString *dEmail = @"jbaker@hotmail.com";

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

 AddressCard *card1 = [[AddressCard alloc] init];
 AddressCard *card2 = [[AddressCard alloc] init];
 AddressCard *card3 = [[AddressCard alloc] init];
 AddressCard *card4 = [[AddressCard alloc] init];

 AddressBook *myBook = [AddressBook alloc];

 // Сначала задаем четыре адресные карточки

 [card1 setName: aName andEmail: aEmail];
 [card2 setName: bName andEmail: bEmail];
 [card3 setName: cName andEmail: cEmail];
```

```

[card4 setName: dName andEmail: dEmail];

myBook = [myBook initWithName: @"Steve's Address Book"];

// Добавляем несколько карточек в адресную книгу

[myBook addCard: card1];
[myBook addCard: card2];
[myBook addCard: card3];
[myBook addCard: card4];
[myBook sort];

if ([NSKeyedArchiver archiveRootObject: myBook toFile: @"addrbook.arch"] == NO)
 NSLog(@"archiving failed");

[card1 release];
[card2 release];
[card3 release];
[card4 release];
[myBook release];

[pool drain];
return 0;
}

```

Эта программа создает адресную книгу и затем архивирует ее в файле `addrbook.arch`. При создании архивного файла метод кодирования вызываются из обоих классов: `AddressBook` и `AddressCard`. Если вы хотите проверить это, добавьте в методы несколько вызовов `NSLog`.

В программе 19.7 показано, как считывать архив в память для создания адресной книги из файла.

### Программа 19.7

```

#import "AddressBook.h"
#import <Foundation/NSAutoreleasePool.h>

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

 myBook = [NSKeyedUnarchiver unarchiveObjectWithFile: @"addrbook.arch"];

 [myBook list];

 [pool drain];
 return 0;
}

```

### Вывод программы 19.7

```
===== Contents of: Steve's Address Book =====
Jamie Baker jbaker@hitmail.com
Julia Kochan jewls337@axlc.com
Stephen Kochan steve@steve_kochan.com
Tony Iannino tony.iannino@techfitness.com
=====
```

При разархивации этой адресной книги автоматически вызываются методы декодирования, добавленные в эти классы. Чтение в программу адресной книги выполняется предельно просто.

Метод `encodeObject:forKey:` используется применительно к встроенным классам и классам, для которых вы пишете свои методы кодирования и декодирования согласно протоколу `NSCoding`. Если ваши переменные экземпляра содержат некоторые базовые типы данных, например, `int` или `float`, вы должны знать, как кодировать и декодировать их (см. таблицу 19.1).

Ниже приводится простое определение для класса с именем `Foo`, который содержит три переменные экземпляра: типа `NSString`, типа `int` и типа `float`. Этот класс содержит один метод-установщик, три метода-получателя и два метода кодирования/декодирования, используемых для архивации.

```
@interface Foo: NSObject <NSCoding>
{
 NSString *strVal;
 int intValue;
 float floatValue;
}

@property (copy, nonatomic) NSString *strVal;
@property int intValue;
@property float floatValue;
@end
```

Затем следует файл секции `implementation`.

```
@implementation Foo

@synthesize strVal, intValue, floatValue;

-(void) encodeWithCoder: (NSCoder *) encoder
{
 [encoder encodeObject: strVal forKey: @"FoostrVal"];
 [encoder encodeInt: intValue forKey: @"FoointVal"];
 [encoder encodeFloat: floatValue forKey: @"FoofloatVal"];
}

-(id) initWithCoder: (NSCoder *) decoder
```

```

{
 strVal = [[decoder decodeObjectForKey: @"FoostrVal"] retain];
 intValue = [decoder decodeIntForKey: @"FointVal"];
 floatValue = [decoder decodeFloatForKey: @"FoofloatVal"];

 return self;
}
@end

```

В этой процедуре кодирования сначала кодируется строковое значение strVal с помощью метода encodeObject:forKey:, как показано ранее.

В программе 19.8 создается объект Foo, выполняется его архивация в файл, разархивация и последующий вывод.

### Программа 19.8. Тестовая программа

```

#import <Foundation/NSObject.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSAutoreleasePool.h>
#import "Foo.h" // Definition for our Foo class

int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Foo *myFoo1 = [[Foo alloc] init];
 Foo *myFoo2;

 [myFoo1 setStrVal: @>This is the string<];
 [myFoo1 setIntVal: 12345];
 [myFoo1 setFloatVal: 98.6];

 [NSKeyedArchiver archiveRootObject: myFoo1 toFile: @"foo.arch"];

 myFoo2 = [NSKeyedUnarchiver unarchiveObjectWithFile: @"foo.arch"];
 NSLog (@"%@%i\n%g", [myFoo2 strVal], [myFoo2 intValue], [myFoo2 floatValue]);
 [myFoo1 release];
 [pool drain];
 return 0;
}

```

### Выход программы 19.8

```

This is the string (Это строка)
12345
98.6

```

Архивация трех п\_6бых экземпляра из объекта выполняется с помощью следующих сообщений.

```
[encoder encodeObject: strVal forKey: @"FoostrVal"];
[encoder encodeInt: intVal forKey: @"FointVal"];
[encoder encodeFloat: floatVal forKey: @"FoofloatVal"];
```

Некоторые из базовых типов данных, такие как `char`, `short`, `long` и `long long`, не включены в таблицу 19.1; для них необходимо определить размер объекта данных и использовать соответствующую процедуру. Например, тип `short int` обычно имеет размер 16 битов, `int` и `long` – 32 или 64 бита, и `long long` – 64 бита. (Размер любого типа данных определяется с помощью оператора `sizeof`, см. главу 13.) Например, данные типа `short int` нужно сохранить их сначала как тип `int` и затем архивировать с помощью `encodeInt:forKey:`. Для декодирования нужно использовать обратный процесс: применить `decodeInt:forKey:` и затем присвоить результат переменной типа `short int`.

## Использование `NSData` для создания нестандартных архивов

Возможно, вы не хотите записывать объект непосредственно в файл с помощью метода `archiveRootObject:ToFile:`, как в предыдущих примерах. Например, вам нужно собрать некоторые объекты и сохранить их в одном архивном файле. Это можно сделать с помощью обобщенного класса объектов потока данных (data stream) `NSData`.

Как говорилось в главе 16, объект класса `NSData` можно использовать для резервирования области памяти для сохранения данных. Эту область памяти можно использовать, например, для временного хранения данных, которые будут последовательно записываться в файл, или для хранения содержимого файла, считанного с диска. Проще всего создать мутабельную область данных с помощью метода `data`.

```
dataArea = [NSMutableData data];
```

В результате создается пустое буферное пространство, размер которого расширяется по мере выполнения программы.

В качестве простого примера предположим, что нужно архивировать в одном файле адресную книгу и один из объектов класса `Foo`. Предположим также, что мы добавили методы архивации с ключами в классы `AddressBook` и `AddressCard` (см. программу 19.9).

### Программа 19.9

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSCoder.h>
#import <Foundation/NSData.h>
#import "AddressBook.h"
#import "Foo.h"
```

```
int main (int argc, char *argv[])
{
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 Foo *myFoo1 = [[Foo alloc] init];
 Foo *myFoo2;
 NSMutableData *dataArea;
 NSKeyedArchiver *archiver;
 AddressBook *myBook;

 // Вставьте здесь код из программы 19.7 для создания в myBook
 // адресной книги, содержащей четыре адресные карточки

 [myFoo1 setStrVal: @"This is the string"];
 [myFoo1 setIntVal: 12345];
 [myFoo1 setFloatVal: 98.6];

 // Создание области данных и ее присоединение к объекту NSKeyedArchiver
 dataArea = [NSMutableData data];

 archiver = [[NSKeyedArchiver alloc]
 initForWritingWithMutableData: dataArea];
 // Теперь можно начать архивацию объектов
 [archiver encodeObject: myBook forKey: @"myaddrbook"];
 [archiver encodeObject: myFoo1 forKey: @"myfoo1"];
 [archiver finishEncoding];

 // Запись архивируемой области данных в файл
 if ([dataArea writeToFile: @"myArchive" atomically: YES
 encoding: NSUTF8StringEncoding error: nil] == NO)
 NSLog(@"Archiving failed!"); // Архивация не выполнена

 [archiver release];

 [myFoo1 release];
 [pool drain];
 return 0;
}
```

После выделения памяти для объекта NSKeyedArchiver передается сообщение `initForWritingWithMutableData:`, чтобы указать область, в которую будут записываться архивируемые данные; это область `NSMutableData` с именем `dataArea`, которая была создана выше. Сохраненному в архиваторе объекту `NSKeyedArchiver` можно передавать теперь сообщения кодирования для архивации объектов в программе. На самом деле, архивация и сохранение все кодируемых сообщений выполняется в указанной области данных, пока не получено сообщение `finishEncoding`.

В данном случае кодируются два объекта: наша адресная книга и объект класса `Foo`. Для этих объектов можно использовать `encodeObject:forKey:`, поскольку мы

ранее реализовали методы кодирования (*encoder*) и декодирования (*decoder*) для классов *AddressBook*, *AddressCard* и *Foo*.

Закончив архивацию этих объектов, мы передаем объекту *archiver* сообщение *finishEncoding*. После этого нельзя кодировать никакие объекты, и мы должны передать это сообщение, чтобы завершить процесс архивации.

Область с именем *dataArea* теперь содержит наши архивированные объекты в форме, которую можно записать в файл. В выражении с сообщением

```
[dataArea writeToFile: @"myArchive" atomically: YES
encoding: NSUTF8StringEncoding error: nil]
```

передается сообщение *writeToFile:atomically:encoding:error:* потоку данных для записи его данных в указанный файл с именем *myArchive*.

Как видно из части с оператором *if*, метод *writeToFile:atomically:encoding:error:* возвращает значение YES типа BOOL, если операция записи успешно выполнена, и значение NO, если ее не удалось выполнить (например, был указан неверный путь к файлу или переполнена файловая система).

Восстановление данных из архивного файла осуществляется просто: нужно выполнить все действия в обратном порядке. Во-первых, нужно выделить, как и раньше, область данных, затем в эту область данных прочитать архивный файл. После этого мы создаем объект *NSKeyedUnarchiver* и сообщаем ему, что требуется декодировать данные из указанной области. Для извлечения и декодирования архивированных объектов нужно вызвать методы декодирования, а по окончании — передать сообщение *finishDecoding* объекту *NSKeyedUnarchiver*. Все это выполняется в программе 19.10.

### Программа 19.10

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSCoder.h>
#import <Foundation/NSData.h>
#import "AddressBook.h"
#import "Foo.h"

int main (int argc, char *argv[]){
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
 NSData *dataArea;
 NSKeyedUnarchiver *unarchiver;
 Foo *myFoo1;
 AddressBook *myBook;
 // Чтение архива и присоединение к нему
 // объекта NSKeyedUnarchiver

 dataArea = [NSData dataWithContentsOfFile: @"myArchive"];
```

```
if (! dataArea) {
 NSLog (@"Can't read back archive file!");
 Return (1);
}

unarchiver = [[NSKeyedUnarchiver alloc]
 initForReadingWithData: dataArea];

// Декодирования объектов, которые мы сохранили ранее в этом архиве
myBook = [unarchiver decodeObjectForKey: @"myaddrbook"];
myFoo1 = [unarchiver decodeObjectForKey: @"myfoo1"];

[unarchiver finishDecoding];

[unarchiver release];

// Проверка того, что восстановление успешно выполнено
[myBook list];
NSLog ("%@\\n%@\\n%g", [myFoo1 strVal],
 [myFoo1 intValue], [myFoo1 floatValue]);

[pool release];
return 0;
}
```

#### Вывод программы 19.10

```
===== Contents of: Steve's Address Book =====
Jamie Baker jbaker@hitmail.com
Julia Kochan jewls337@axlc.com
Stephen Kochan steve@steve_kochan.com
Tony Iannino tony.iannino@techfitness.com
=====
```

```
This is the string
12345
98.6
```

Адрессная книга и объект Foo были успешно восстановлены из архивного файла.

## Использование архиватора для копирования объектов

В программе 18.2 мы пытались создать копию массива, содержащего мутабельные строковые элементы, и создали поверхностную (shallow) копию этого массива — копировались не сами строки, а только ссылки на них.

Возможности архивации Foundation позволяют создать глубокую (deep) копию объекта. Например, в программе 19.11 выполняется копирование dataArray в dataArray2 путем архивации dataArray в буфер и его последующей деархивации с присваиванием результата массиву dataArray2. Нам не нужно задействовать файл для этого процесса; архивацию и деархивацию можно выполнять в памяти.

### Программа 19.11

```
#import <Foundation/NSObject.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <Foundation/NSKeyedArchiver.h>
#import <Foundation/NSArray.h>

int main (int argc, char *argv[])
{
 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
 NSData *data;
 NSMutableArray *dataArray = [NSMutableArray arrayWithObjects:
 [NSMutableString stringWithString: @"one"],
 [NSMutableString stringWithString: @"two"],
 [NSMutableString stringWithString: @"three"],
 nil
];
 NSMutableArray *dataArray2;
 NSMutableString *mStr;

 // Создание глубокой копии с помощью архиватора

 data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
 dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];

 mStr = [dataArray2 objectAtIndex: 0];
 [mStr appendString: @"ONE"];

 NSLog(@"%@", dataArray);
 for (NSString *elem in dataArray)
 NSLog(@"%@", elem);

 NSLog(@"%@", dataArray2);
 for (NSString *elem in dataArray2)
 NSLog(@"%@", elem);

 [pool drain];
 return 0;
}
```

### Вывод программы 19.11

```
dataArray:
```

```
one
```

```
two
```

```
three
```

```
dataArray2:
```

```
oneONE
```

```
two
```

```
three
```

Вывод подтверждает, что изменение первого элемента dataArray2 не оказывает влияния на первый элемент dataArray, поскольку новая копия этой строки была получена в ходе архивации/деархивации.

Операция копирования в программе 19.11 выполняется с помощью следующих двух строк.

```
data = [NSKeyedArchiver archivedDataWithRootObject: dataArray];
dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData: data];
```

Мы можем избежать промежуточного присваивания и выполнить копирование с помощью одного оператора.

```
dataArray2 = [NSKeyedUnarchiver unarchiveObjectWithData:
 [NSKeyedArchiver archivedDataWithRootObject: dataArray]];
```

Этот подход полезен для создания глубокой копии объекта или объекта, который не поддерживает протокол NSCopying.

## Упражнения

1. В главе 15 в программе 15.7 была создана таблица простых чисел. Внесите изменения в эту программу, чтобы записать результирующий массив в виде списка свойств XML в файл primes.plist. Проверьте содержимое этого файла.
2. Напишите программу для чтения списка свойств XML, созданного в упражнении 1, и сохраните эти свойства в объекте-массиве. Выполните вывод всех элементов массива, чтобы убедиться, что операция восстановления прошла успешно.
3. Внесите изменения в программу 19.2, чтобы вывести содержимое одного из списков свойств XML (файлы .plist), сохраненного в папке /Library/Preferences.
4. Напишите программу чтения архивированной адресной книги (AddressBook) и выполните поиск записи в соответствии с именем, указанным в командной строке, например

```
$ lookup gregory
```

## Глава 20

# Введение в Сосоа

На протяжении этой книги мы разрабатывали программы, которые имеют довольно простой пользовательский интерфейс. Для вывода сообщений на консоль мы использовали процедуру `@@ NSLog`. Это действительно полезная процедура, но ее возможности ограничены. Другие программы, которые мы используем на своих Маках, имеют намного более удобный интерфейс. Репутация компьютеров Macintosh во многом основывается на дружественных пользовательских окнах и простоте использования. В своих приложениях мы можем использовать XCode вместе с приложением Interface Builder. Это сочетание образует мощную среду для разработки программ со средствами редактирования и отладки и удобным доступом к online-документации и позволяет легко разрабатывать сложные графические пользовательские интерфейсы (GUI).

Фреймворки для поддержки приложений, обеспечивающие удобный пользовательский интерфейс, называются Сосоа. Это два фреймворка: Foundation framework, с которым вы уже знакомы, и фреймворк Application Kit (или AppKit). Второй фреймворк содержит классы, связанные с окнами, кнопками, списками и т.д.

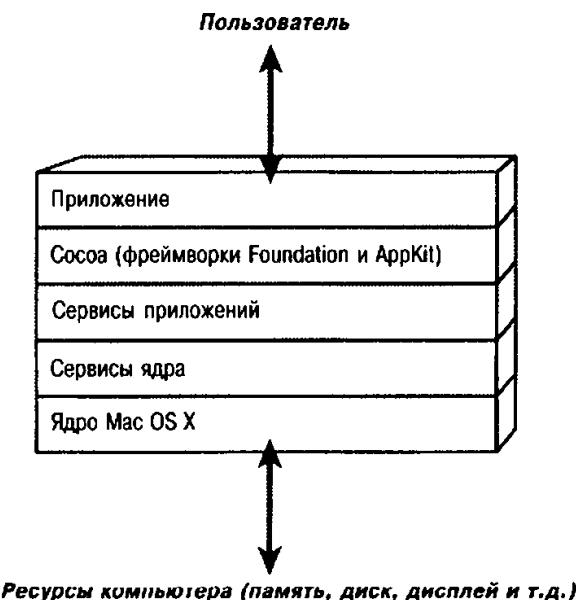
## Уровни фреймворков

Чтобы показать уровни, которые отделяют приложение от оборудования, часто используют схемы. Одна из таких схем показана на рис. 20.1.

*Ядро системы* обеспечивает низкоуровневую связь с оборудованием в форме *драйверов устройств*. Оно управляет ресурсами системы, такими как программы-планировщики, управление памятью и электропитанием и выполнение базовых операций ввода-вывода.

*Сервисы ядра (Core Services)* обеспечивают поддержку на нижнем уровне (уровне ядра), в отличие от находящихся выше уровней. Здесь обеспечивается поддержка коллекций, сетевого обмена, управления файлами, папок, управления памятью, отладки, потоков, времени и электропитания.

Уровень *Сервисов приложений (Application Services)* включает поддержку печати и воспроизведения графики, включая Quartz, OpenGL и Quicktime.



**Рис. 20.1.** Иерархия уровней для приложения

Непосредственно под приложением находится уровень Cocoa. Как говорилось выше, *Cocoa* содержит фреймворки *Foundation* и *AppKit*. *Foundation* содержит классы для работы с коллекциями, строками, для управления памятью, файловой системой, архивацией и т.д. *AppKit* содержит классы для управления представлениями (*view*), окнами, документами, а также для обширного пользовательского интерфейса, который так хорошо известен пользователям Mac OS X.

Из этого описания возникает ощущение дублирования функций некоторых уровней. Например, коллекции существуют на уровнях *Cocoa* и сервисов ядра. Однако первый уровень основывается на поддержке со стороны второго уровня. Кроме того, в некоторых случаях определенный уровень можно пропускать. Например, некоторые классы *Foundation* для работы с файловой системой целиком основываются на функциях уровня сервисов ядра в обход уровня сервисов приложений. Во многих случаях фреймворк *Foundation* определяет объектно-ориентированное отображение структур данных, определенных на уровне сервисов ядра (написанных преимущественно на процедурном языке C).

## Cocoa Touch

Телефон iPhone содержит компьютер, который работает под управлением упрощенной версии Mac OS X. Некоторые возможности оборудования iPhone, например, акселерометр, уникальны для этого телефона и отсутствуют на других компьютерах под управлением Mac OS X, таких как MacBook или iMac.

---

**Примечание.** На самом деле ноутбуки Мак содержат акселерометр для выполнения парковки жесткого диска в случае падения компьютера, но вы не имеете непосредственного доступа к этому акселерометру.

---

В отличие от фреймворков Cocoa, используемых при разработке приложений для настольных компьютеров и ноутбуков с Mac OS X, фреймворки Cocoa Touch используются для приложений, которые будут работать на iPhone и iPod touch.

Cocoa и Cocoa Touch содержат один общий фреймворк — Foundation. Однако в Cocoa Touch вместо фреймворка AppKit используется фреймворк UIKit, поддерживающий много таких же типов объектов (окна, представления, кнопки, текстовые поля и т.д.). Кроме того, Cocoa Touch содержит классы для работы с акселерометром, триангуляции местоположения с помощью GPS и сигналов WiFi, а также сенсорный интерфейс. Из него исключены ненужные классы, например, поддержка клавиатуры.

На этом заканчивается краткий обзор Cocoa. В следующей главе описывается, как писать приложение для iPhone с использованием имитатора, который является частью комплекта разработки программ (SDK) iPhone.

## Глава 21

# Написание приложений iPhone

В этой главе мы будем разрабатывать два простых приложения iPhone. В первом из них вы познакомитесь с понятиями *делегат* (*delegate*), *outlet*-переменные и *действия* (*action*) и с некоторыми концепциями использования Interface Builder и создания соединений. Во втором приложении мы создадим калькулятор для дробей, объединив то, что вы узнали при разработке первого приложения, со всем материалом этой книги.

## Комплект разработки программ (SDK) для iPhone

Чтобы писать приложения для iPhone, необходимо установить Xcode и комплект iPhone SDK. Этот SDK можно получить бесплатно с веб-сайта Apple. Для загрузки SDK нужно зарегистрироваться в качестве разработчика Apple (Apple Developer). Этот процесс тоже выполняется бесплатно. Чтобы получить соответствующие ссылки, начните с адреса developer.apple.com и перейдите к нужному пункту. Обязательно ознакомьтесь с этим сайтом. В приложении ID содержится несколько прямых ссылок на определенные места этого сайта, которые могут заинтересовать вас.

Материал этой главы основывается на Xcode 3.1.1 и iPhone SDK для iPhone OS 2.1. Следующие версии этих продуктов совместимы с изложенным здесь материалом.

## Ваше первое приложение iPhone

В этом приложении показано, как поместить черно-белое окно на экран iPhone, разрешить пользователю нажать на кнопку и вывести текст в ответ на нажатие этой кнопки.

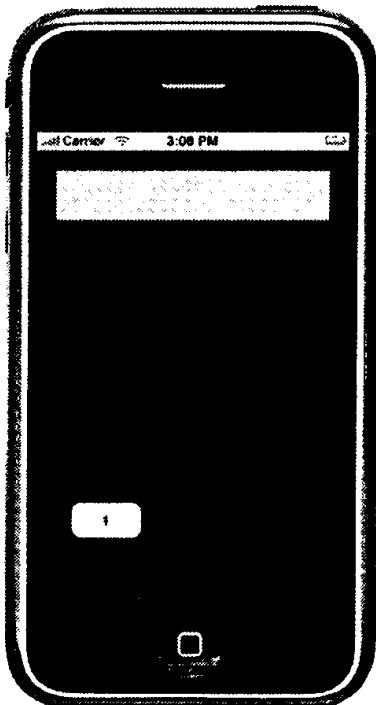
---

**Примечание.** Второе приложение еще интересней! В нем используется опыт первого приложения для создания простого калькулятора, позволяющего выполнять операции с дробями. Мы можем использовать класс Fraction, с которым работали в предыдущих главах, а также модифицированный класс Calculator. На этот раз калькулятор сможет работать с дробями.

---

Приступим к первой программе. Мы опишем самые необходимые шаги, чтобы на их основе вы могли разрабатывать собственные программы Сосоа или iPhone.

На рис. 21.1 показано ваше первое приложение для iPhone, которое будет выполняться па имитаторе iPhone (об этом чуть ниже).

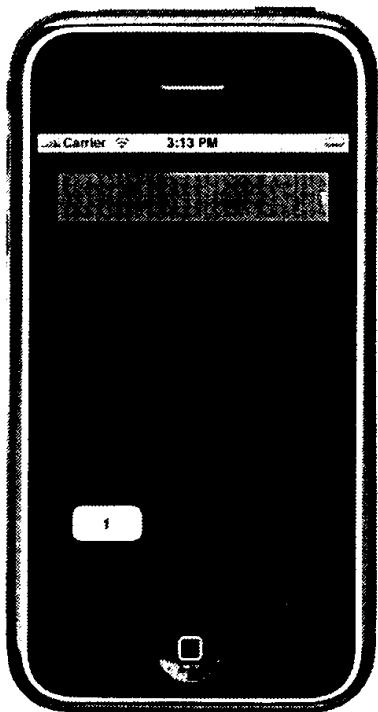


**Рис. 21.1.** Первое приложение для iPhone

В этом приложении при нажатии кнопки «1» на дисплее будет появляться соответствующая цифра (см. рис. 21.2). Оно больше ничего не делает! Это простое приложение является основой для второго приложения, которое представляет калькулятор для работы с дробями.

Мы создадим приложение с помощью Xcode и пользовательский интерфейс с помощью Interface Builder. Если вы работали с Xcode в предыдущих главах, то можете использовать его для ввода и тестирования своих программ. Как говорилось выше, Interface Builder позволяет разрабатывать пользовательский интерфейс, размещая элементы пользовательского интерфейса (UI) – таблицы, метки и кнопки – в окне, которое похоже на элемент iPhone. Работа с Interface Builder, как с любым серьезным средством разработки, требует определенного опыта.

Apple распространяет имитатор (simulator) iPhone в составе SDK iPhone. Этот имитатор повторяет многие элементы среды iPhone, включая домашнюю страницу, веб-браузер Safari, приложение Contacts (Контакты) и т.д. Этот имитатор намного упрощает отладку приложений; вам не нужно каждый раз загружать разрабатываемое приложение на реальное устройство iPhone и затем выполнить отладку на этом устройстве.



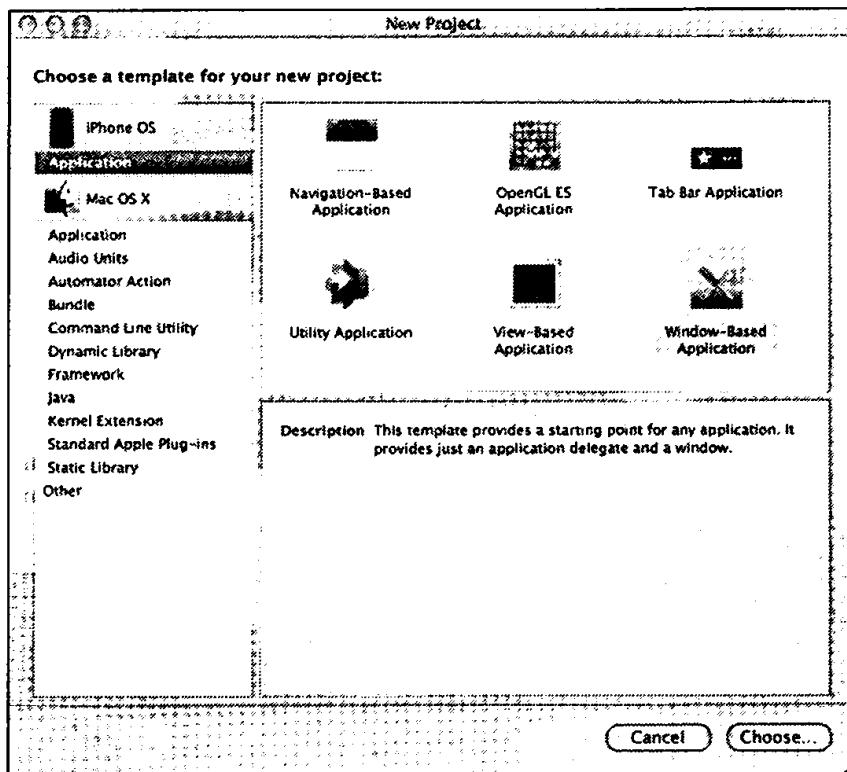
**Рис. 21.2.** Первое приложение iPhone

Чтобы запускать приложения на устройстве iPhone, вам нужно зарегистрироваться как разработчику программ iPhone и заплатить компании Apple 99 долларов (по состоянию на момент написания этой книги). В ответ вы получите код активации, который позволит вам получить сертификат разработки для iPhone (iPhone Development Certificate), чтобы вы могли тестировать и устанавливать приложения на своем iPhone. К сожалению, вы не можете разрабатывать приложения даже для вашего собственного iPhone без прохождения этого процесса. Приложение, которое мы разрабатываем в этой главе, будет загружаться и тестироваться на имитаторе iPhone, а не на устройстве iPhone.

## **Создание нового проекта приложения iPhone**

Вернемся к разработке нашего первого приложения. После установки SDK iPhone запустите приложение Xcode. В меню File (Файл) выберите New Project (Новый проект). Под iPhone OS (если этого пункта нет в левой панели, значит, вы не установили SDK iPhone) щелкните на Application (Приложение). Появится окно, показанное на рис. 21.3.

Здесь мы видим шаблоны, которые можно использовать как отправную точку для различных типов приложений, в соответствии с таблицей 21.1.

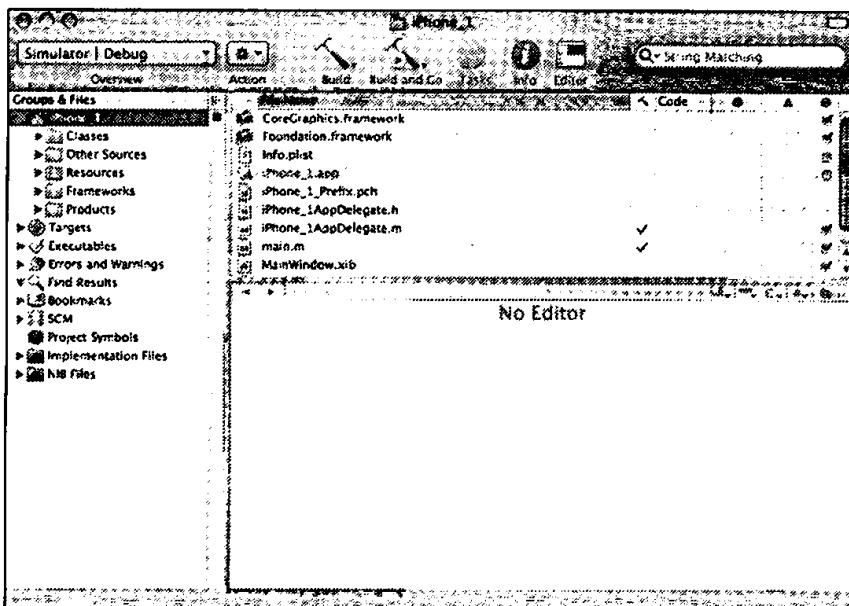


**Рис. 21.3.** Запуск нового проекта для iPhone

**Табл. 21.1.** Шаблоны приложений iPhone

| Тип приложения   | Описание                                                                                                                                  |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Navigation-Based | Для приложения, в котором используется контроллер навигации. <i>Contacts</i> – это пример приложения данного типа.                        |
| OpenGL ES        | Для графических приложений OpenGL, например, игр.                                                                                         |
| Tab Bar          | Для приложений, в которых используется полоса вкладок. Примером может служить приложение <i>iPod</i> .                                    |
| Utility          | Для приложения, где используется представление <i>flipside</i> (обратная сторона). Примером может служить приложение <i>Stock Quote</i> . |
| View-Based       | Для приложения, в котором используется одно представление (view). Вы переходите к этому представлению и затем выводите его в окне.        |
| Window-Based     | Для приложения, которое запускается из главного окна iPhone. Его можно использовать как отправную точку для любого приложения.            |

Вернувшись к окну New Project, выберите в верхней правой панели Window-Based Application и затем щелкните на кнопке Choose (Выбрать). При последующем запросе ввода имени проекта (в поле Save As – Сохранить как) введите текст iPhone\_1 и щелкните на кнопке Save (Сохранить). Это имя станет именем вашего приложения по умолчанию. Как вы уже знаете из предыдущих проектов, созданных с помощью Xcode, будет создан новый проект, содержащий шаблоны для необходимых файлов (рис. 21.4).



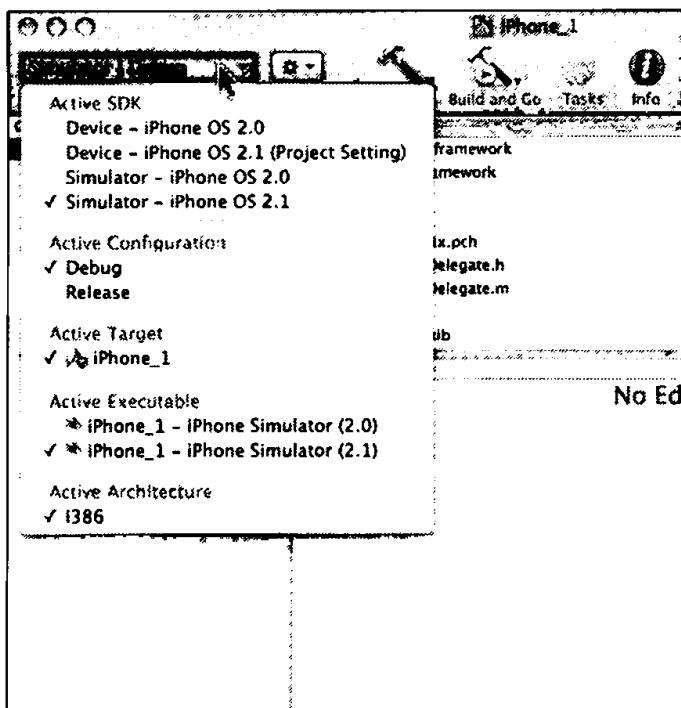
**Рис. 21.4.** Создание нового проекта iPhone 1

В зависимости от ваших настроек и предыдущего использования Xcode ваше окно может несколько отличаться от показанного на рисунке. Вы можете продолжить работу с текущего вида или постараться изменить его так, чтобы он походил на этот рисунок.

В верхнем левом углу окна Xcode мы видим раскрывающийся список, помеченный текущим выбором для SDK и активной конфигурации (Configuration). Поскольку мы не разрабатываем приложение для непосредственного выполнения на iPhone, нужно настроить SDK для работы с имитатором iPhone (Simulator) и для конфигурации задать вариант Debug (Отладка). Если этот раскрывающийся список не помечен как Simulator | Debug, выберите соответствующие опции, как показано на рис. 21.5.

## Ввод кода

Теперь мы можем внести изменения в некоторые файлы проекта. Обратите внимание, что для вас созданы классы `имя_проектаАppDelegate.h` и `имя_проектаАppDelegate.m` (в данном примере `имя_проекта` – это `iPhone_1`). Вся работа по управлению кнопками и метками в типе приложения `Window-based`, которое мы создаем, *делегируется* классу, который называется `имя_проектаАppDelegate` (в данном случае – `iPhone_1AppDelegate`). В этом классе



**Рис. 21.5.** Проект iPhone\_1 с заданными опциями секций SDK и Configuration

мы определим методы, чтобы реагировать на действия, возникающие в окне iPhone, такие как нажатие кнопки или перемещение ползунка. Как вы увидите, конкретная связь между этими управляющими элементами и соответствующими методами задается в приложении Interface Builder.

Этот класс будет также содержать переменные конфигурации, значения которых соответствуют некоторому управляющему элементу в окне iPhone, например, имя на метке или текст, отображаемый в поле изменяемого текста. Эти переменные называют *outlet*-переменными, и, аналогично процедурам действий, мы связываем в Interface Builder переменные экземпляра с конкретным управляющим элементом окна iPhone.

Для нашего первого приложения нужен метод, реагирующий на действие, которое состоит из нажатия кнопки с меткой 1. Нам нужна также *outlet*-переменная, содержащая (среди прочей информации) текст, который должен отображаться на метке, создаваемой вверху окна iPhone.

Внесем изменения в файл iPhone\_1AppDelegate.h, чтобы добавить переменную типа UILabel с именем display и объявить метод для действия с именем click1:, чтобы реагировать на нажатие кнопки. Наш файл секции interface показан в программе 21.1. (Здесь нет строк комментария, которые автоматически вставляются в начале файла.)

#### Программа 21.1. iPhone\_1AppDelegate.h

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

@interface iPhone_1AppDelegate : NSObject <UIApplicationDelegate> {
```

```
UIWindow *window;

UILabel *display;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UILabel *display;

-(IBAction) click1: (id) sender;
@end
```

Отметим, что приложения iPhone импортируют header-файл `<UIKit/UIKit.h>`. Этот файл импортирует, в свою очередь, другие header-файлы UIKit, что аналогично импорту в файле `Foundation.h` других нужных header-файлов, например, `NSString.h` и `NSObject.h`. Чтобы посмотреть содержимое этого файла, нужно пройти достаточно длинный путь. На момент написания этой книги он содержался в папке

```
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator2.1.sdk/System/Library/Frameworks/UIKit.framework/Headers/.
```

Теперь класс `iPhone_1AppDelegate` содержит две переменные экземпляра. Первая — это объект `UIWindow` с именем `window`. Эта переменная экземпляра создается автоматически, когда мы создаем проект, и является ссылкой на главное окно iPhone. Мы добавили еще одну переменную экземпляра с именем `display`, которая принадлежит классу `UILabel`. Это outlet-переменная, которая будет связана с меткой. Когда мы задаем текстовое поле для этой переменной, происходит изменение текста для метки в окне. Другие методы, определенные для класса `UILabel`, позволяют задавать и считывать такие атрибуты метки, как цвет, число строк и размер шрифта.

Для создания интерфейсов вам потребуются и другие классы, но мы не будем описывать их здесь. Имена некоторых классов объясняют их назначение: `UITextField` (Текстовое поле), `UIFont` (Шрифт), `UIView` (Представление), `UITableView` (Табличное представление), `UIImageView` (Представление изображения), `UIImage` (Изображение) и `UIButton` (Кнопка).

Обе переменные — `window` и `display` — являются outlet-переменными. В объявлении свойств для этих переменных используется идентификатор `IBOutlet`. `IBOutlet` определяется как *nothing* (ничто) в header-файле `UINibDeclarations.h` для UIKit. (То есть он фактически не заменяется ничем в исходном файле при обработке предпроцессором.) Однако он необходим, поскольку Interface Builder ищет `IBOutlet`, когда читает header-файл, чтобы определить, какие переменные можно использовать как outlet-переменные.

В этом interface-файле отметим объявление метода с именем `click1:` ему передается один аргумент с именем `sender`. При вызове метода `click1:` в этом аргументе ему будет передаваться информация, связанная с данным событием. Например, если у нас имеется action-процедура для обработки нажатий различных кнопок, то этот аргумент можно запрашивать для определения нажатой кнопки.

Метод `click1:` определен для возврата значения типа `IBAction`. (Оно определяется как `void` в файле `UINibDeclarations.h`.) Аналогично `IBOutlet`, этот идентифи-

тор используется приложением Interface Builder, когда оно просматривает header-файл, чтобы определить, какие методы можно рассматривать как действия (action).

Теперь можно внести изменения в implementation-файл iPhone\_1AppDelegate.m для нашего класса. Здесь мы синтезируем методы доступа (accessor method) для переменной display (методы доступа для window уже синтезированы для вас) и добавляем определение для метода click1:.

Внесите изменения в implementation-файл, чтобы он был похож на файл, показанный в программе 21.1.

#### Программа 21.1. iPhone\_1AppDelegate.m

```
#import ""iPhone_1AppDelegate.h"
@implementation iPhone_1AppDelegate

@synthesize window, display;

-(void) applicationDidFinishLaunching:(UIApplication *)application {

 // Место переопределения для настройки после запуска приложения
 [window makeKeyAndVisible];
}

-(IBAction) click1: (id) sender
{
 [display setText: @"1"];
}

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

@end
```

Метод applicationDidFinishLaunching: автоматически вызывается системой runtime iPhone один раз; как следует из его имени, оно означает, что запуск приложения завершен. Здесь можно инициализировать переменные экземпляра, нарисовать что-то на экране и сделать окно видимым для отображения его содержимого. Последнее действие осуществляется при передаче сообщения makeKeyAndVisible окну (window) в конце этого метода.

Метод click1: задает отображение этой outlet-переменной в строке 1 с помощью метода UILabel setText:. Когда мы свяжем нажатие кнопки с вызовом данного метода, он сможет выполнить нужное действие: поместить 1 на дисплей в окне iPhone. Чтобы задать эту связь, вы должны узнать, как используется Interface Builder. Прежде чем сделать это, нужно собрать программу, чтобы удалить предупреждения компилятора или сообщения об ошибках.

## Проектирование интерфейса

На рис. 21.4 и в вашем главном окне Xcode обратите внимание на файл MainWindow.xib. Файл с расширением xib содержит всю информацию о пользовательском интерфейсе для программы, включая информацию о его окнах, кнопках метках, полосах вкладок (tab bar), текстовых полях и т.д. Конечно, у нас еще нет пользовательского интерфейса! Это следующий шаг.

Дважды щелкните на файле MainWindow.xib. Будет запущено еще одно приложение, Interface Builder. Доступ к этому XIB-файлу можно также выполнить из папки Resources вашего проекта.

При запуске Interface Builder на экране появится последовательность окон (рис. 21.6, 21.7, 21.8). Окна, которые открываются у вас, могут отличаться от изображенных на рисунках.

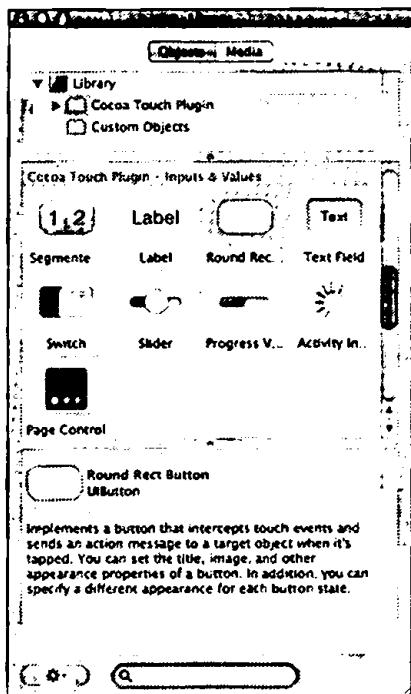
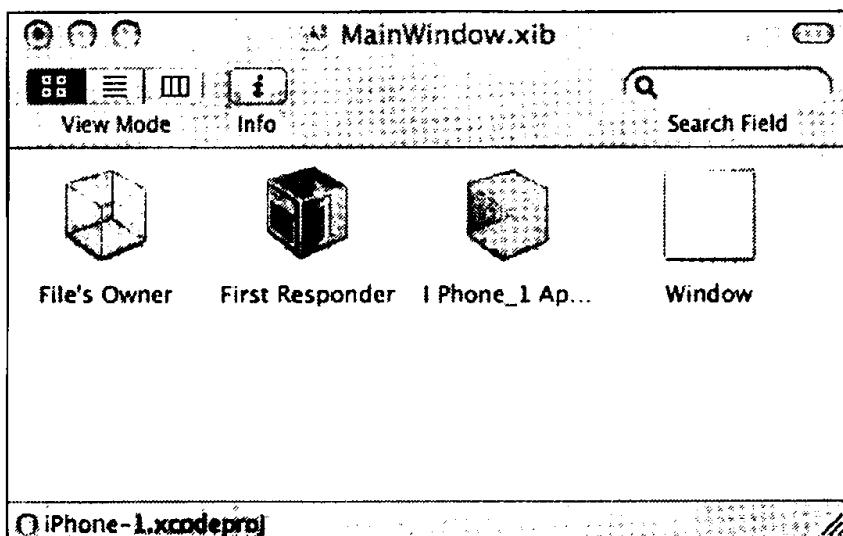


Рис. 21.6. Okno Library приложения Interface Builder

Окно Library (Библиотека) содержит набор управляющих элементов интерфейса. Это окно представлено на рис. 21.6.

Окно MainWindow.xib (рис. 21.7) является управляющим окном для задания связей между кодом приложения и интерфейсом.



**Рис. 21.7.** Окно Interface Builder MainWindow.xib

В окне под заголовком Window показан макет главного окна iPhone. Поскольку мы еще ничего не создали для окна iPhone, оно является пустым (рис. 21.8).

Сначала зададим черный цвет для окна iPhone. Для этого щелкните внутри окна под заголовком Window и выберите пункт Inspector в меню Tools (Сервис). Появится окно Inspector (рис. 21.9).

Убедитесь, что окно Inspector называется Window Attributes (Атрибуты окна), как на рисунке 21.9. Если это не так, щелкните на левой вкладке в полосе вкладок, чтобы отобразить нужное окно.

В секции View (Вид) этого окна имеется атрибут с именем Background (Фон). Дважды щелкните внутри белого прямоугольника рядом с меткой Background. Появится указатель цветов. Выберите черный цвет из этого указателя. Цвет прямоугольника рядом с атрибутом Background изменится на черный (рис. 21.10).

Окно Window, представляющее окно отображения iPhone, тоже будет иметь черный цвет (рис. 21.11).

Теперь можно закрыть окно цветов Colors.

Для создания нового объекта в окне интерфейса iPhone нужно щелкнуть на объекте в окне Library и перетянуть его в ваше окно iPhone. Щелкните и перетяните элемент Label (Метка). Отпустите кнопку мыши, когда метка окажется наверху примерно посередине окна (рис. 21.12).

Во время перемещения метки в окне будут показаны синие направляющие линии. Иногда они нужны для выравнивания объектов относительно других объектов, размещенных ранее в этом окне. Иногда они требуются, чтобы объекты находились на достаточном расстоянии друг от друга и от краев окна в соответствии с рекомендациями Apple.

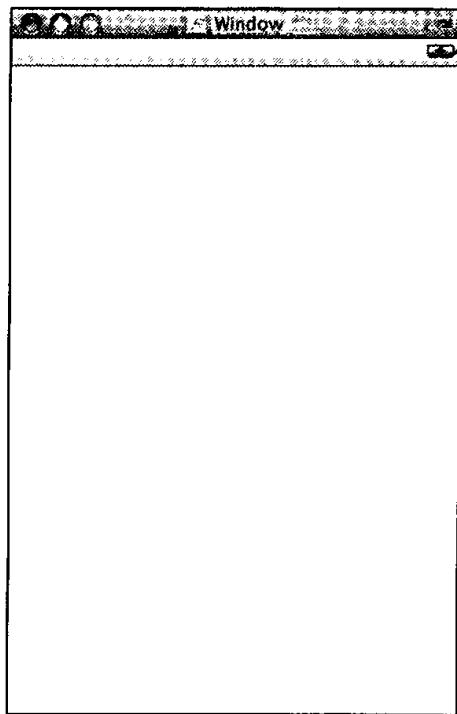


Рис. 21.8. Окно iPhone в Interface Builder

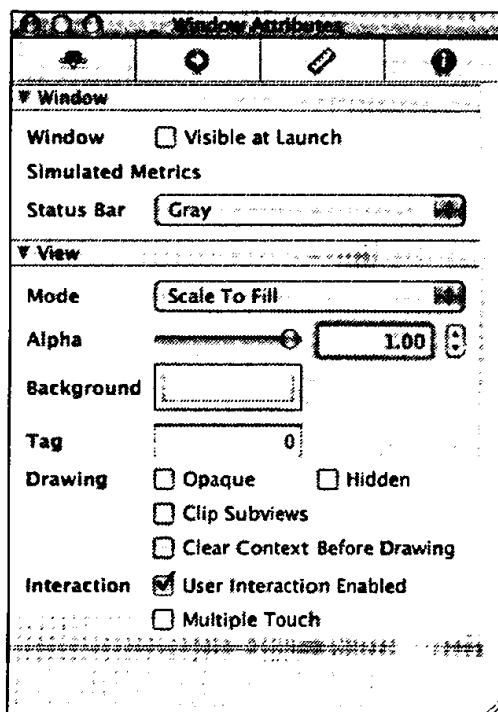


Рис. 21.9. Окно Inspector в Interface Builder

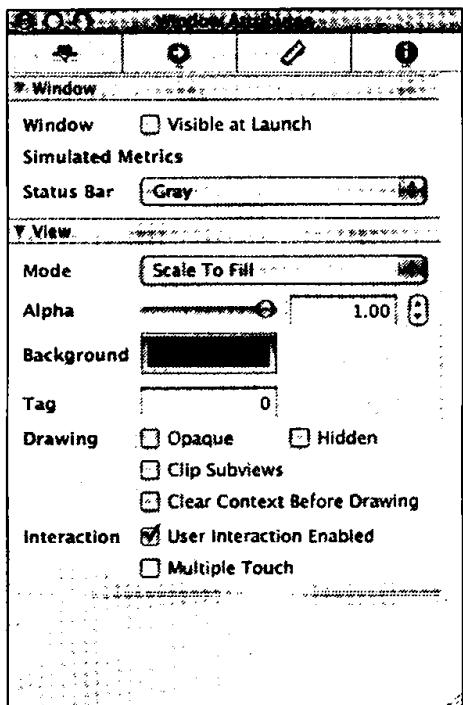


Рис. 21.10. Изменение цвета фона окна

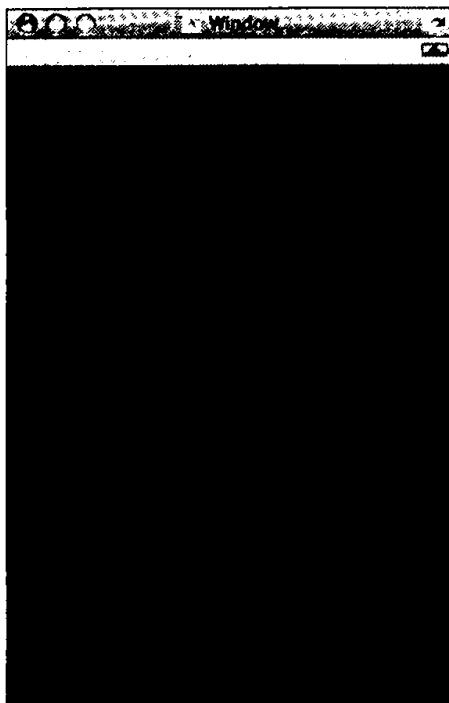
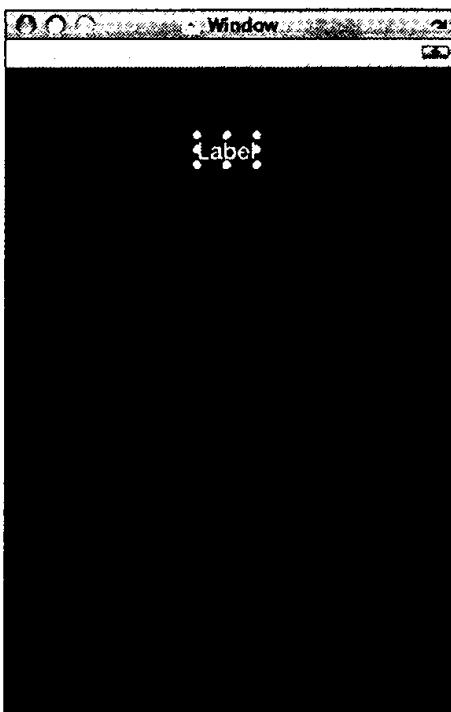


Рис. 21.11. Изменение цвета окна интерфейса на черный цвет



**Рис. 21.12.** Добавление метки

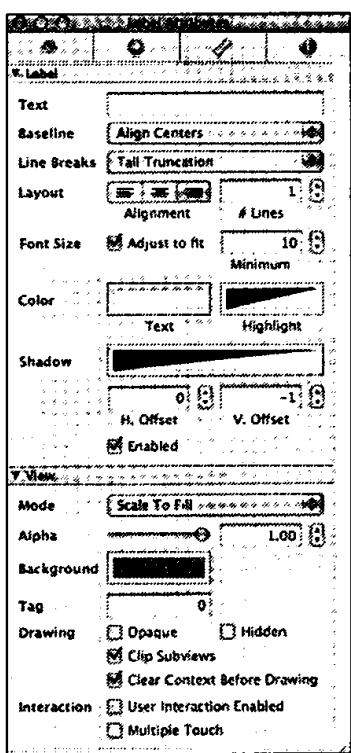
Вы можете в любой момент перетянуть метку в другое место внутри окна.

Теперь зададим некоторые атрибуты для этой метки. Щелкните на метке, которую вы только что создали, чтобы выделить ее (если она еще не выделена). Отметим, что окно Inspector автоматически изменяется, чтобы вы получали информацию о текущем выбранном объекте в вашем окне. Нам не нужно, чтобы текст по умолчанию появился в этой метке, поэтому измените значение Text на пустую строку, то есть удалите текст Label из текстового поля, показанного в окне Inspector.

Для атрибута Layout (Компоновка) выберите Right-justified (Выравнивание справа) среди вариантов выравнивания (Alignment). И, наконец, измените цвет фона (background) для метки на синий так же, как изменили цвет фона окна на черный. Ваше окно Inspector должно быть похоже на рис. 21.13.

Теперь изменим размер метки. Вернемся к окну Window и изменим размер метки, растянув ее углы и стороны. Измените размер и местоположение метки, чтобы она выглядела, как на рисунке 21.14.

Теперь добавим в этот интерфейс кнопку. Из окна Library перетяните объект Round Rect Button в ваше окно интерфейса, поместив его в нижний левый угол окна (рис. 21.15). Метку на этой кнопке можно изменить одним из двух способов: дважды щелкнуть на кнопке и затем ввести нужный текст или ввести текст в поле Title (Заголовок) в окне Inspector. При любом способе сделайте так, чтобы ваше окно было похоже на показанное на рисунке 21.15.



**Рис. 21.13.** Изменение атрибутов метки

Теперь у нас есть метка, которую нужно связать с переменной экземпляра `display`, чтобы при задании этой переменной в программе текст метки был изменен.

У нас также есть кнопка с меткой 1, для которой нужно задать вызов метода `click1:` при каждом нажатии этой кнопки. Этот метод задаст значение 1 для текстового поля на дисплее. А поскольку эта переменная будет связана с данной меткой, метка будет изменяться. Приведем ниже последовательность, которую нужно задать.

1. Пользователь нажимает кнопку с меткой 1.
2. По этому событию вызывается метод `click1:`.
3. Метод `click1:` изменяет текст переменной экземпляра `display` на строку 1.
4. Поскольку объект `UILabel display` связан с меткой в окне iPhone, текст этой метки изменяется на соответствующее текстовое значение, равное 1.

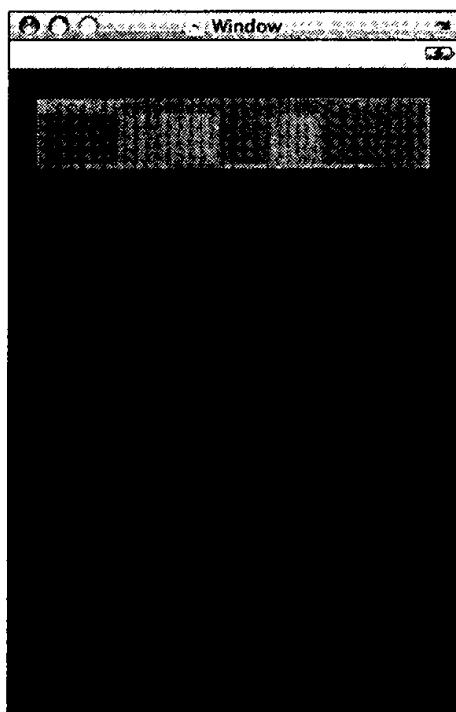


Рис. 21.14. Изменение размера и местоположения метки

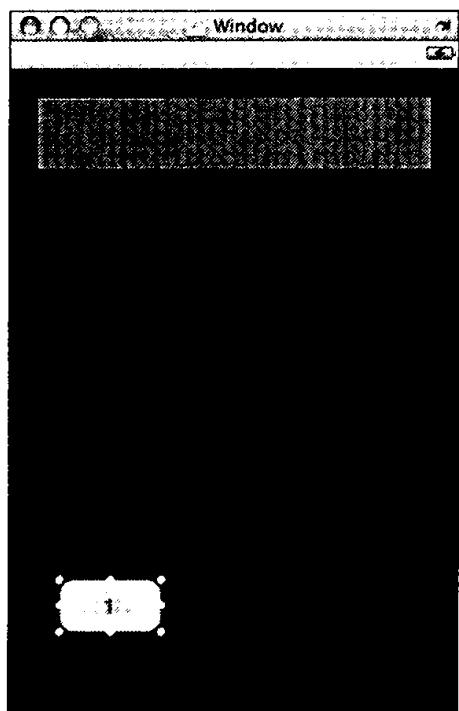


Рис. 21.15. Добавление кнопки в интерфейс

Для выполнения этой последовательности нам требуются только две привязки.

Сначала свяжем кнопку с методом `IBAction click1:`. Для этого удерживайте нажатой клавишу `Control` во время щелчка на этой кнопке и затем протяните синюю линию, которая появится на экране, к кубику делегата приложения (`AppDelegate`) в окне `MainWindow.xib`, как показано рис. 21.16.

Когда вы отпустите кнопку мыши над кубиком делегата приложения, появится раскрывающееся меню, где можно выбрать метод `IBAction` для привязки к кнопке. В данном случае имеется только один такой метод (с именем `click1:`), и он появится в этом раскрывающемся меню. Выберите этот метод, чтобы создать привязку (рис. 21.17).

Теперь выполним привязку переменной `display` к метке. Нажатие кнопки вызывает выполнение метода в приложении (то есть управление передается от интерфейса к делегату приложения), задание значения переменной экземпляра в приложении вызывает изменение текста метки в окне iPhone. (Управление передается от делегата приложения к интерфейсу.) Поэтому, удерживая нажатой клавишу `Control`, щелкните на значке делегата приложения и протяните к метке синюю линию, которая появится в окне `Window` (рис. 21.18).

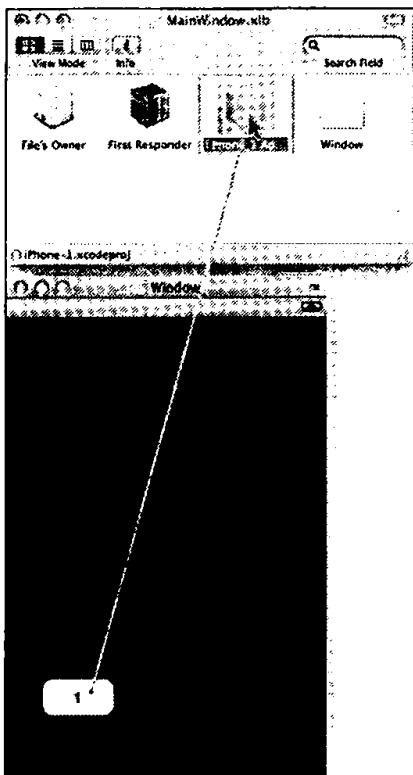


Рис. 21.16. Добавление действия для кнопки

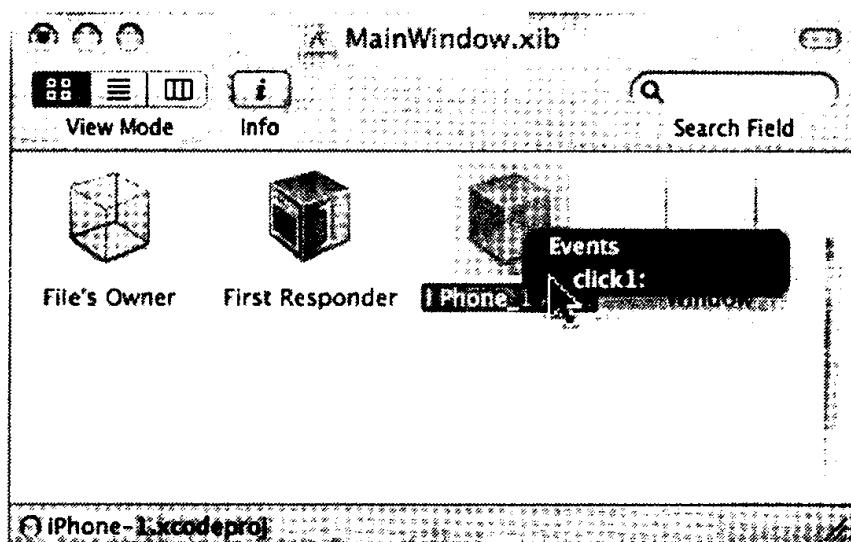


Рис. 21.17. Привязка события к методу

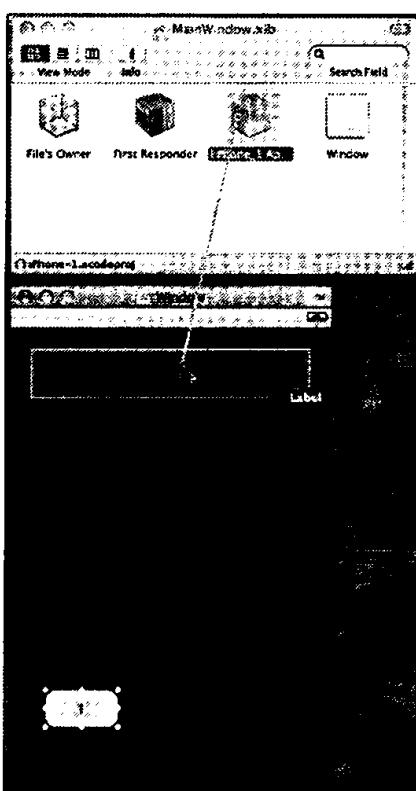


Рис. 21.18. Привязка outlet-переменной

Отпустив кнопку мыши, мы получим список outlet-переменных соответствующего класса как элемент управления (UILabel), среди которых можно сделать выбор. В нашей программе такая переменная одна — с именем `display`. Выберите ее и выполните привязку (рис. 21.19).

Выберите `File->Save` в линейке меню Interface Builder и затем `Build and Go` в Xcode. (Это можно также инициировать из Interface Builder.)

Если все проходит нормально, программа будет успешно собрана и начнет выполняться. При этом она будет сначала загружена в имитатор iPhone, который появится на экране вашего компьютера (см. рис. 21.1). Чтобы имитировать нажатие кнопки в имитаторе, нужно просто щелкнуть на ней. Последовательность шагов и привязок, которая описана выше, будет реализована в виде отображения текстовой строки 1 в метке вверху дисплея, как показано на рис. 21.2.

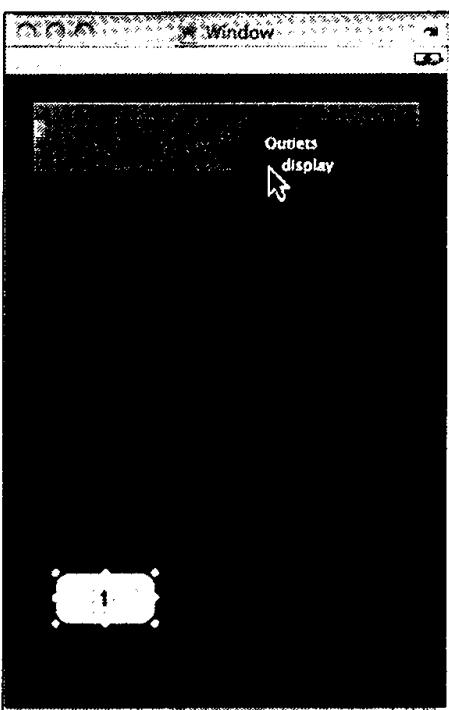


Рис. 21.19. Завершение привязки

## Калькулятор дробей для iPhone

Следующий пример несколько сложнее, но к нему применяются все концепции предыдущего примера. Мы не будем описывать шаги по созданию этого примера полностью, а только приведем сводку этих шагов и дадим обзор методологии разработки. И, конечно, покажем весь код.

Сначала рассмотрим, как работает приложение. Приложение на имитаторе непосредственно после запуска показано на рис. 21.20.

Этот калькулятор позволяет вводить дроби. Сначала вводится числитель (numerator), затем нужно нажать клавишу с меткой Over и ввести знаменатель (denominator). Таким образом, чтобы ввести дробь  $2/5$ , нужно нажать 2, затем Over и 5. В отличие от других калькуляторов, это приложение показывает простую дробь на дисплее, то есть  $2/5$  отображается как  $2/5$ .

После ввода дроби нужно выбрать операцию: сложение, вычитание, умножение или деление – и нажать клавишу +, -,  $\times$  или ?.

После ввода второй дроби нужно завершить операцию, нажав клавишу =, как в стандартном калькуляторе.

---

**Примечание.** Этот калькулятор может выполнять только одну операцию над дробями. В упражнении в конце главы от вас потребуется снять это ограничение.

---

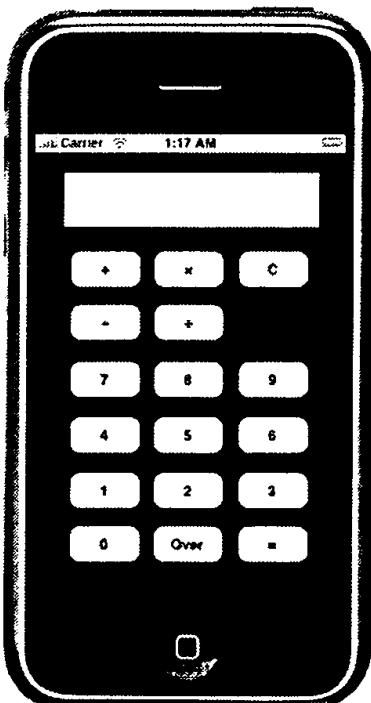


Рис. 21.20. Калькулятор дробей после запуска

По мере нажатия клавиш дисплей непрерывно обновляется. На рис. 21.21 показан дисплей после ввода дроби  $4/6$  и нажатия клавиши умножения.

На рис. 21.22 показан результат умножения дробей  $4/6$  и  $2/8$ . Результат  $1/6$  показывает, что выполняется сокращение результирующей дроби.

### Запуск нового проекта Fraction\_Calculator

Первая программа iPhone в этой главе начиналась с шаблона проекта типа Window-Based. Наша небольшая работа, связанная с пользовательским интерфейсом (UI), была выполнена непосредственно в контроллере приложения (с помощью класса AppDelegate). Такой подход не рекомендуется для разработки приложений с насыщенным пользовательским интерфейсом. Класс AppDelegate обычно используется только для обработки изменений, относящихся к состоянию самого приложения, например, к окончанию запуска приложения или к его завершению.

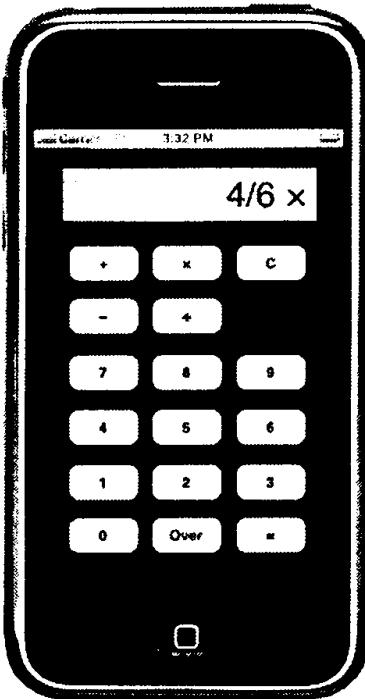


Рис. 21.21 Ввод операции

Контроллер представлений (view controller), реализованный с помощью класса UIViewController, подходит именно для действий, относящихся к UI, таких как отображение текста, реагирование на нажатие кнопки или вывод другого представления на экране iPhone.

Разработку второго примера программы мы начнем с создания нового проекта. На этот раз в окне New Project мы выберем тип View-Based Application и назовем новый проект Fraction\_Calculator.

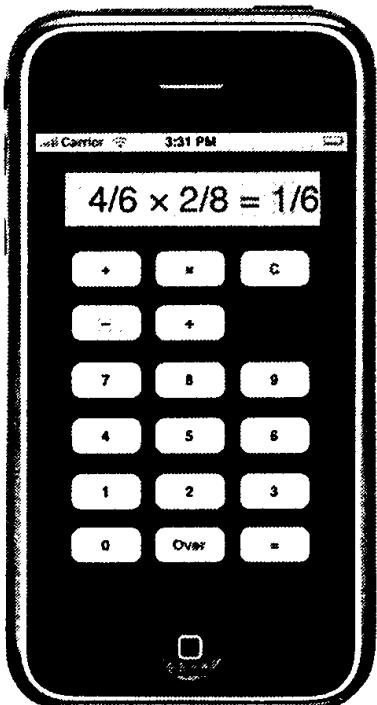


Рис. 21.22. Результат умножения двух дробей

При создании проекта вы увидите, что на этот раз используются два шаблона классов. `Fraction_CalculatorAppDelegate.h` и `Fraction_CalculatorAppDelegate.m` определяют для нашего проекта класс контроллера приложения, а `Fraction_CalculatorViewController.h` и `Fraction_CalculatorViewController.m` определяют класс контроллера представлений. Как говорилось выше, вся наша работа будет выполнена именно во втором классе.

Начнем с класса контроллера приложения. Он содержит две переменные экземпляра: одну для ссылки на окно iPhone и вторую – для контроллера представлений. Обе переменные заданы в Xcode. Вам не потребуется вносить никакие изменения в .h-файл и .m-файл контроллера приложения.

Файл секции `interface Fraction_CalculatorAppDelegate` показан в программе 21.2.

#### Программа 21.2. Interface-файл `Fraction_CalculatorAppDelegate.h`

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

@class Fraction_CalculatorViewController;

@interface Fraction_CalculatorAppDelegate : NSObject <UIApplicationDelegate> {
 IBOutlet UIWindow *window;
 IBOutlet Fraction_CalculatorViewController *viewController;
}
```

```
@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) Fraction_CalculatorViewController *viewController;

@end
```

Переменная экземпляра `UIWindow` `window` предназначена для той же цели, что и в первой программе: она представляет окно iPhone. Переменная экземпляра `Fraction_CalculatorViewController` используется для контроллера представлений, который будет управлять всем взаимодействием с пользователем и выводом на дисплей. В файле секции `implementation` для этого класса мы реализуем всю работу, связанную с этими задачами.

В программе 21.2 показан `implementation`-файл для класса контроллера приложения. Как говорилось выше, мы не будем делать никакой работы в этом файле (в отличие от программы 21.1); вся работа передается контроллеру представлений. Таким образом, этот файл показан так, как он генерируется Xcode при создании нового проекта.

#### Программа 21.2. Implementation-файл `Fraction_CalculatorAppDelegate.m`

```
#import "Fraction_CalculatorAppDelegate.h"
#import "Fraction_CalculatorViewController.h"

@implementation Fraction_CalculatorAppDelegate

@synthesize window;
@synthesize viewController;

-(void)applicationDidFinishLaunching:(UIApplication *)application {
 // Место переопределения для настройки после запуска приложения
 [window addSubview:viewController.view];
 [window makeKeyAndVisible];
}

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

@end
```

### Определение контроллера представлений

Теперь мы напишем код для класса контроллера представлений `Fraction_CalculatorViewController`. Начнем с файла секции `interface`. Он показан в программе 21.2.

#### Программа 21.2. Interface-файл `Fraction_CalculatorViewController.h`

```
#import <UIKit/UIKit.h>
#import "Calculator.h"
```

```
@interface Fraction_CalculatorViewController : UIViewController {
 UILabel *display;
 char op;
 int currentNumber;
 NSMutableString *displayString;
 BOOL firstOperand, isNumerator;
 Calculator *myCalculator;
}

@property (nonatomic, retain) IBOutlet UILabel *display;
@property (nonatomic, retain) NSMutableString *displayString;

-(void) processDigit: (int) digit;
-(void) processOp: (char) op;
-(void) storeFracPart;

// Числовые клавиши

-(IBAction) clickDigit: (id) sender;

// Клавиши арифметических операций

-(IBAction) clickPlus: (id) sender;
-(IBAction) clickMinus: (id) sender;
-(IBAction) clickMultiply: (id) sender;
-(IBAction) clickDivide: (id) sender;

// Другие клавиши

-(IBAction) clickOver: (id) sender;
-(IBAction) clickEquals: (id) sender;
-(IBAction) clickClear: (id) sender;

@end
```

У нас имеются служебные переменные для создания дробей (`currentNumber`, `firstOperand` и `isNumerator`) и для создания выводимой строки (`displayString`). Объект типа `Calculator` (`myCalculator`) выполняет конкретные вычисления между двумя дробями. Мы выполним привязку метода с именем `clickDigit:` для обработки нажатий цифровых клавиш 0-9. И, наконец, мы определяем методы для хранения операции, которая должна быть выполнена (`clickPlus:`, `clickMinus:`, `clickMultiply:`, `clickDivide:`), выполняя сами вычисления при нажатии клавиши `=` (`clickEquals:`), сброс текущей операции (`clickClear:`) и отделение числителя от знаменателя при нажатии клавиши `Over` (`clickOver:`). Несколько методов (`processDigit:`, `processOp:` и `storeFracPart`) определяются для этих задач как вспомогательные средства.

В программе 21.2 показан файл секции `implementation` для этого класса контроллера.

**Программа 21.2.** Implementation-файл Fraction\_CalculatorViewController.m

```
#import <Fraction_CalculatorViewController.h>
@implementation Fraction_CalculatorViewController

@synthesize display, displayString;

-(void) viewDidLoad {
 // Место переопределения для настройки после запуска приложения

 firstOperand = YES;
 isNumerator = YES;
 self.displayString = [NSMutableString stringWithCapacity: 40];
 myCalculator = [[Calculator alloc] init];
}

-(void) processDigit: (int) digit
{
 currentNumber = currentNumber * 10 + digit;

 [displayString appendString: [NSString stringWithFormat: @"%i", digit]];
 [display setText: displayString];
}

-(IBAction) clickDigit:(id)sender
{
 int digit = [sender tag];

 [self processDigit:digit];
}

-(void) processOp: (char) theOp
{
 NSString *opStr;

 op = theOp;

 switch (theOp) {
 case '+':
 opStr = @"+ ";
 break;
 case '-':
 opStr = @"- ";
 break;
 case '*':
 opStr = @"? ";
 break;
 }
}
```

```
case '/':
 opStr = @"?";
 break;
}

[self storeFracPart];
firstOperand = NO;
isNumerator = YES;

[displayString appendString: opStr];
[display setText: displayString];
}

-(void) storeFracPart
{
 if (firstOperand) {
 if (isNumerator) {
 myCalculator.operand1.numerator = currentNumber;
 myCalculator.operand1.denominator = 1; // e.g. 3 * 4/5 =
 }
 else
 myCalculator.operand1.denominator = currentNumber;
 }
 else if (isNumerator) {
 myCalculator.operand2.numerator = currentNumber;
 myCalculator.operand2.denominator = 1; // e.g. 3/2 * 4 =
 }
 else {
 myCalculator.operand2.denominator = currentNumber;
 firstOperand = YES;
 }
}

currentNumber = 0;
}

-(IBAction) clickOver: (id) sender
{
 [self storeFracPart];
 isNumerator = NO;
 [displayString appendString: @"/"];
 [display setText: displayString];
}

// Клавиши арифметических операций

-(IBAction) clickPlus: (id) sender
{
```

```
[self processOp: '+'];
}

-(IBAction) clickMinus: (id) sender
{
 [self processOp: '-'];
}

-(IBAction) clickMultiply: (id) sender
{
 [self processOp: '*'];
}

-(IBAction) clickDivide: (id) sender
{
 [self processOp: '/'];
}

// Другие клавиши

-(IBAction) clickEquals: (id) sender
{
 [self storeFracPart];
 [myCalculator performOperation: op];
 [displayString appendString: @" = "];
 [displayString appendString: [myCalculator.accumulator convertToString]];
 [display setText: displayString];

 currentNumber = 0;
 isNumerator = YES;
 firstOperand = YES;
 [displayString setString: @""];
}

-(IBAction) clickClear: (id) sender
{
 isNumerator = YES;
 firstOperand = YES;
 currentNumber = 0;
 [myCalculator clear];

 [displayString setString: @""];
 [display setText: displayString];
}

-(void)dealloc {
 [myCalculator dealloc];
```

```
[super dealloc];
}

@end
```

Окно калькулятора пока содержит только одну метку, как в предыдущем приложении, и мы по-прежнему называем ее `display`. По мере того, как пользователь вводит число цифра за цифрой, мы должны формировать это число. Переменная `current_Number` содержит накапливаемое число, а BOOL-переменные `firstOperand` и `isNumerator` следят за тем, какой операнд введен (первый или второй), и что представляет этот операнд – числитель или знаменатель.

При нажатии числовой клавиши идентифицирующая информация будет передаваться методу `clickDigit:`, чтобы указать, какая числовая клавиша была нажата. Для этого атрибуту клавиши с именем `tag` (с помощью средства `Inspector` в `Interface Builder`) присваивается соответствующее значение каждой числовой клавиши. В данном случае нам нужно присвоить `tag` соответствующую цифру. Например, для клавиши с меткой 0 `tag` будет присвоено значение 0, для клавиши с меткой 1 – значение 1, и т.д. При последующей отправке сообщения `tag` параметру `sender`, который передается методу `clickDigit:`, мы считываем значение тега клавиши. Это происходит в методе `clickDigit:`, как показано ниже.

```
-(IBAction) clickDigit:(id)sender
{
 int digit = [sender tag];

 [self processDigit:digit];
}
```

В программе 21.2 намного больше клавиш, чем в первом приложении. Наиболее сложной частью в `implementation`-файле контроллера представлений является формирование и отображение дробей. Как говорилось выше, при нажатии числовой клавиши от 0 до 9 выполняется `action`-метод `clickDigit:`. Этот метод вызывает метод `processDigit:`, который помещает цифру в конец числа, формируемого в переменной `currentNumber`. Этот метод также добавляет цифру в текущую строку отображения, которая поддерживается в переменной `displayString`, и обновляет отображение.

```
-(void) processDigit: (int) digit
{
 currentNumber = currentNumber * 10 + digit;

 [displayString appendString: [NSString stringWithFormat: @"%i", digit]];
 [display setText: displayString]; }
```

При нажатии клавиши со знаком равенства (=) для выполнения операции вызывается метод `clickEquals:`. Калькулятор выполняет операцию между двумя дробями, сохраняя результат в накопителе (`accumulator`). Этот накопитель считывается внутри метода `clickEquals:`, и результат выводится на дисплей.

## Класс Fraction

Класс Fraction мало отличается от предыдущих примеров этой книги. В нем имеется новый метод `convertToString`, который добавлен для преобразования дроби в эквивалентное строковое представление. В программе 21.2 показан файл секции `interface` для класса Fraction, после которого следует соответствующий файл секции `implementation`.

### Программа 21.2. Interface-файл Fraction.h

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

@interface Fraction : NSObject {
 int numerator;
 int denominator;
}

@property int numerator, denominator;

-(void) print;
-(void) setTo: (int) n over: (int) d;
-(Fraction *) add: (Fraction *) f;
-(Fraction *) subtract: (Fraction *) f;
-(Fraction *) multiply: (Fraction *) f;
-(Fraction *) divide: (Fraction *) f;
-(void) reduce;
-(double) convertToNum;
-(NSString *) convertToString;

@end
```

### Программа 21.2. Implementation-файл Fraction.m

```
#import "Fraction.h"

@implementation Fraction

@synthesize numerator, denominator;

-(void) setTo: (int) n over: (int) d
{
 numerator = n;
 denominator = d;
}

-(void) print
{
 NSLog(@"%@", numerator, denominator);
}
```

```
- (double) convertToNum
{
 if (denominator != 0)
 return (double) numerator / denominator;
 else
 return 1.0;
}

-(NSString *) convertToString;
{
 if (numerator == denominator)
 if (numerator == 0)
 return @>0";
 else
 return @>1";
 else if (denominator == 1)
 return [NSString stringWithFormat: @"%i", numerator];
 else
 return [NSString stringWithFormat: @"%i/%i",
 numerator, denominator];
}

// Сложение дроби (Fraction) с получателем

-(Fraction *) add: (Fraction *) f
{
 // Сложение двух дробей
 // a/b + c/d = ((a*d) + (b*c)) / (b * d)

 // в result будет сохраняться результат сложения
 Fraction *result = [[Fraction alloc] init];
 int resultNum, resultDenom;

 resultNum = numerator * f.denominator + denominator * f.numerator;
 resultDenom = denominator * f.denominator;

 [result setTo: resultNum over: resultDenom];
 [result reduce];

 return [result autorelease];
}

-(Fraction *) subtract: (Fraction *) f
{
 // Вычитание двух дробей
 // a/b - c/d = ((a*d) - (b*c)) / (b * d)
```

```
Fraction *result = [[Fraction alloc] init];
int resultNum, resultDenom;

resultNum = numerator * f.denominator - denominator * f.numerator;
resultDenom = denominator * f.denominator;

[result setTo: resultNum over: resultDenom];
[result reduce];
return [result autorelease];
}

-(Fraction *) multiply: (Fraction *) f
{
 Fraction *result = [[Fraction alloc] init];

 [result setTo: numerator * f.numerator over: denominator
 * f.denominator];
 [result reduce];

 return [result autorelease];
}

-(Fraction *) divide: (Fraction *) f
{
 Fraction *result = [[Fraction alloc] init];

 [result setTo: numerator * f.denominator over: denominator * f.numerator];
 [result reduce];

 return [result autorelease];
}

-(void) reduce
{
 int u = numerator;
 int v = denominator;
 int temp;

 if (u == 0)
 return;
 else if (u < 0)
 u = -u;

 while (v != 0) {
 temp = u % v;
 u = v;
 v = temp;
 }
}
```

```

 }

 numerator /= u;
 denominator /= u;
}

@end

```

Метод `convertToString`: проверяет числитель и знаменатель дроби, чтобы создать вид числа для внешнего отображения. Если числитель и знаменатель равны (но не равны нулю), возвращается @"1". Если числитель равен нулю, то возвращается строка @"0". Если знаменатель равен 1, результатом является целое число, то есть показывать знаменатель не нужно.

Метод `stringWithFormat:`, который используется внутри `convertToString:`, возвращает строку в соответствии со строкой формата (аналогично `NSLog`) и список аргументов с разделителями-запятыми. Мы передаем аргументы методу, который принимает переменное число параметров, разделяя их запятыми, как при передаче аргументов функции `NSLog`.

## Класс Calculator, который работает с дробями

Теперь рассмотрим класс `Calculator`. Он аналогичен одноименному классу, с которым мы работали раньше, но теперь наш калькулятор должен «знать», как работать с дробями. Ниже приводятся файлы секций `interface` и `implementation` для класса `Calculator`.

### Программа 21.2. Interface-файл Calculator.h

```

#import <UIKit/UIKit.h>
#import "Fraction.h"

@interface Calculator : NSObject {
 Fraction *operand1;
 Fraction *operand2;
 Fraction *accumulator;
}

@property (retain, nonatomic) Fraction *operand1, *operand2, *accumulator;

-(Fraction *) performOperation: (char) op;
-(void) clear;

@end

```

### Программа 21.2. Implementation-файл Calculator.m

```

#import "Calculator.h"

@implementation Calculator

```

```
@synthesize operand1, operand2, accumulator;

-(id) init
{
 self = [super init];

 operand1 = [[Fraction alloc] init];
 operand2 = [[Fraction alloc] init];
 accumulator = [[Fraction alloc] init];

 return self;
}

-(void) clear
{
 if (accumulator) {
 accumulator.numerator = 0;
 accumulator.denominator = 0;
 }
}

-(Fraction *) performOperation: (char) op
{
 Fraction *result;

 switch (op) {
 case '+':
 result = [operand1 add: operand2];
 break;
 case '-':
 result = [operand1 subtract: operand2];
 break;
 case '*':
 result = [operand1 multiply: operand2];
 break;
 case '/':
 result = [operand1 divide: operand2];
 break;
 }

 accumulator.numerator = result.numerator;
 accumulator.denominator = result.denominator;

 return accumulator;
}

-(void) dealloc
```

```

{
 [operand1 release];
 [operand2 release];
 [accumulator release];
 [super dealloc];
}
@end

```

## Разработка пользовательского интерфейса (UI)

В папке Resources этого проекта содержатся два xib-файла: MainWindow.xib и Fraction\_CalculatorViewController.xib. Вам не требуется работать с первым файлом. Откроем второй файл, дважды щелкнув на его имени. При запуске Interface Builder вы увидите значок с именем “View” в окне с именем Fraction\_CalculatorViewController.xib (рис. 21.23).

Если окно View еще не открыто, дважды щелкните на этом значке и откройте его. Внутри окна View мы будем разрабатывать пользовательский интерфейс нашего калькулятора. Мы будем связывать каждую числовую клавишу с методом clickDigit:. Для этого нужно при нажатой клавише Control протянуть мышь к значку File’s Owner по очереди от каждой клавиши в окне Fraction\_CalculatorViewController.xib и выбрать clickDigit: в раскрывающемся меню Events (События). Кроме того, для каждой числовой клавиши нужно задать в окне Inspector значение Tag, соответствующее названию клавиши. Для числовой клавиши с меткой 0 нужно задать значение Tag, равное 0, для клавиши с меткой 1 – значение 1, и т.д.

Протяните мышь от остальных клавиш в окне View и создайте соответствующие привязки. Не забудьте вставить метку для отображения (display) калькулятора и протяните мышь при нажатой клавише Control от File’s Owner к этой метке. Выберите display из появившегося списка Outlets.

Все! Структура интерфейса создана, и приложение с калькулятором дробей готово к работе.

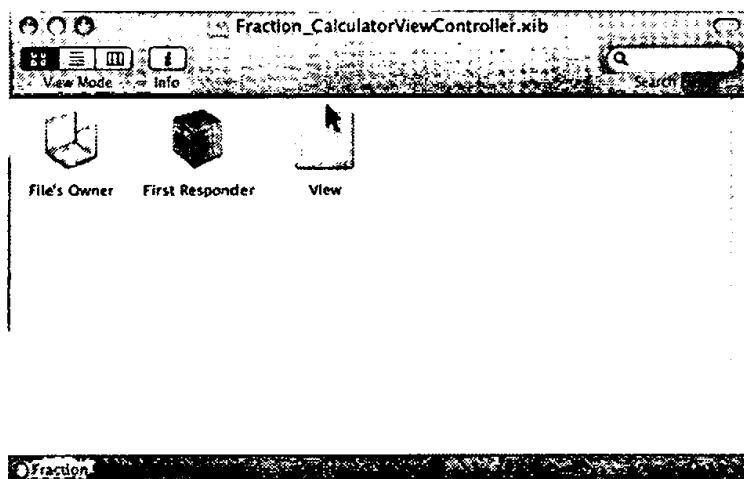


Рис. 21.23. Окно Fraction\_CalculatorViewController.xib

## Сводка шагов

На рис. 21.24 показано окно проекта Xcode со всеми файлами, относящимися к проекту.

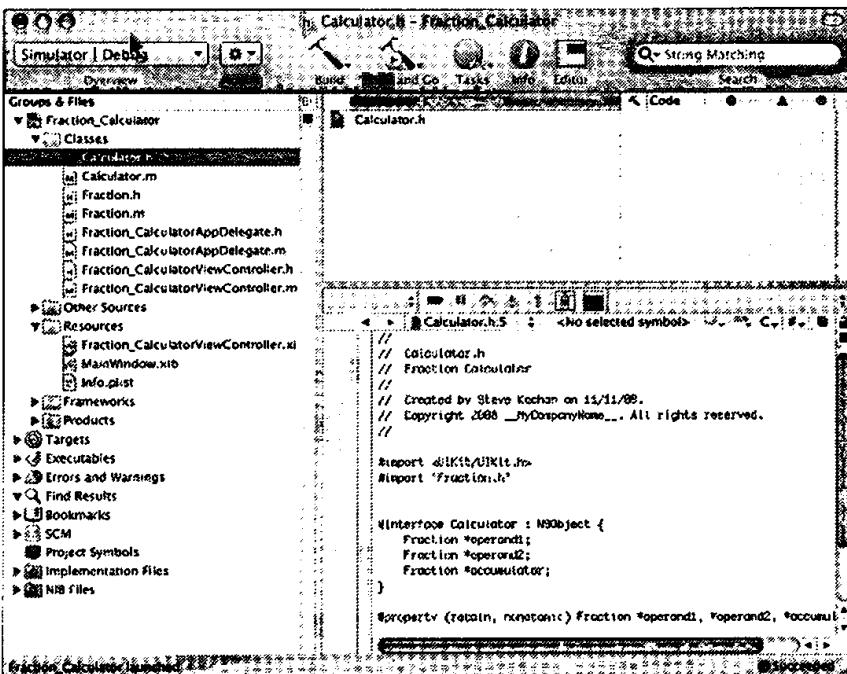


Рис. 21.24. Файлы проекта калькулятора дробей

Ниже приводится сводка шагов по созданию калькулятора дробей для iPhone.

1. Создание нового приложения типа View-based.
2. Ввод UI-кода в файлы **Fraction\_CalculatorViewController** с расширениями .h и .m.
3. Добавление в проект классов **Fraction** и **Calculator**.
4. Открытие окна **Fraction\_CalculatorViewController.xib** в Interface Builder для создания UI.
5. Создание черного фона окна View.
6. Создание метки и клавиш (кнопок), их позиционирование внутри окна View.
7. При нажатой клавише Control протягивание мыши из File's Owner к метке, созданной в окне View, и выбор для нее варианта “display”.
8. При нажатой клавише Control протягивание мыши от каждой клавиши в окне View к File's Owner и привязка к соответствующему action-методу. Для каждой числовой клавиши выбирается метод **clickDigit:**. Кроме того, для каждой числовой клавиши нужно присвоить атрибуту клавиши tag соответствующую цифру от 0 до 9, чтобы метод **clickDigit:** мог определить, какая клавиша была нажата.

Изучение контроллера представлений было бы полезным упражнением, но это потребовало бы куда больше усилий, чем вся разработка проекта в контроллере приложения. Однако, если вам нужно осуществлять в приложении более сложные задачи, например, выполнить анимацию, реагировать на поворот экрана, использовать контроллер навигации или создать интерфейс с вкладками, то контроллер представлений вам просто необходим.

Мы надеемся, что это краткое введение в разработку приложений iPhone поможет вам в разработке ваших собственных приложений iPhone. Как говорилось выше, UIKit предоставляет разработчику множество возможностей.

В описанном выше приложении для калькулятора дробей имеется несколько ограничений. Многие из них вы снимете, выполнив упражнения, которые приводятся ниже.

## Упражнения

1. Добавьте клавишу Convert (Преобразование) в приложение для калькулятора дробей. При нажатии этой клавиши используйте метод convertToNum класса Fraction, чтобы создать десятичное представление для результата, полученного в виде дроби. Преобразуйте это число в строку и выведите его на дисплее калькулятора.
2. Внесите изменения в приложение для калькулятора дробей, чтобы можно было вводить отрицательную дробь (перед вводом числителя нажимается клавиша «-»).
3. Если для знаменателя введено значение «0», нужно вывести строку Етот на дисплее калькулятора дробей.
4. Внесите изменения в приложение для калькулятора дробей, чтобы можно было выполнять цепочку вычислений. Например, нужно разрешить выполнение следующей операции:  
 $1/5 + 2/7 - 3/8 =$
5. Вы можете добавить значок приложения, который будет отображаться на экране iPhone. Для этого можно в папке Resources вашего приложения добавить изображение, которое будет использоваться в качестве значка (.png-файл), и задать этот файл изображения для клавиши «Icon file» в списке информационных свойств (файл Info.plist в вашей папке Resources), как показано на рис. 21.25.

Найдите в Интернете подходящее изображение калькулятора и задайте для калькулятора дробей использование этого изображения как значка приложения.

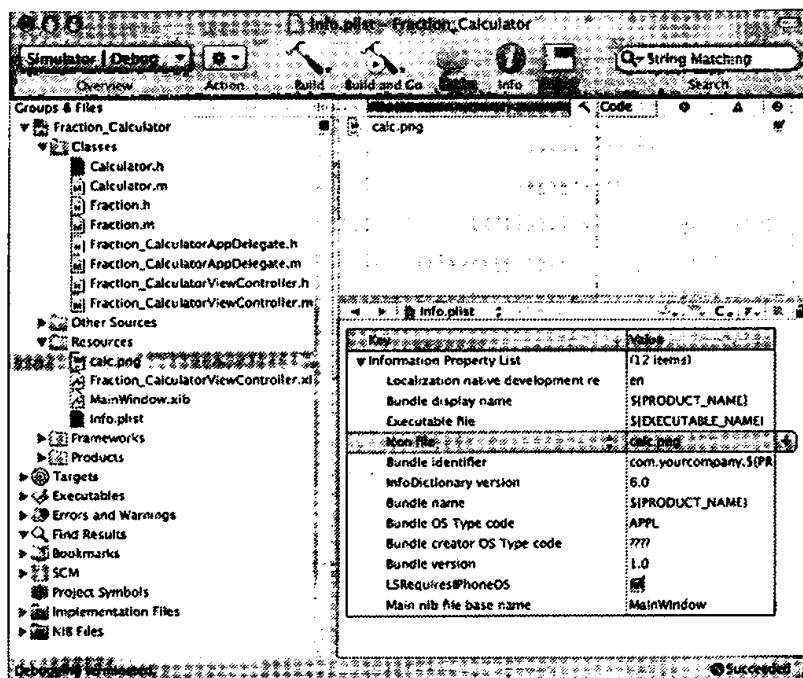


Рис. 21.25. Добавление значка приложения

# Приложение А

## Словарь

Это приложение содержит неформальные определения многих терминов, которые встречаются в книге. Некоторые из них относятся непосредственно к языку Objective-C, а другие связаны с объектно-ориентированным программированием. Для них указан смысл термина в языке Objective-C.

**Application Kit.** Фреймворк для разработки пользовательского интерфейса приложения, который включает такие объекты, как меню, панели инструментов и окна. Входит в Сосоа. Иногда называется AppKit.

**Cocoa Touch.** Среда разработки, состоящая из фреймворков Foundation и UIKit.  
**Cocoa.** Среда разработки, состоящая из фреймворков Foundation и Application Kit.

**extern-переменная.** См. Глобальная переменная.

**Foundation framework.** Коллекция классов, функций и протоколов, которые создают основу для разработки приложений, обеспечивая такие базовые возможности, как управление памятью, доступ к файлам и URL, выполнение задач архивации, работа с коллекциями, строками, а также с числовыми объектами и объектами данных.

**gcc.** Имя компилятора, разработанного организацией Free Software Foundation (FSF). gcc поддерживает многие языки программирования, включая C, Objective-C и C++. gcc – это стандартный компилятор, используемый в Mac OS X для компиляции программ Objective-C.

**gdb.** Стандартное средство отладки для программ, откомпилированных с помощью gcc.

**Header-файл.** См. Заголовочный файл.

**id.** Обобщенный тип объекта, который может содержать указатель на любой тип объекта.

**Interface Builder.** Средство под управлением Mac OS X для создания графического пользовательского интерфейса для приложения.

**isa.** Специальная переменная экземпляра, определенная в корневом объекте и наследуемая всеми объектами. Переменная isa используется во время выполнения (runtime) для идентификации класса, которому принадлежит объект.

**nil.** Объект типа id, который используется для представления недействительного объекта. Его значение определяется как 0. Объекту nil можно передавать сообщения.

**NSObject.** Корневой объект в Foundation framework.

**self.** Переменная, которая используется внутри метода для ссылки на получателя сообщения.

**super.** Ключевое слово, используемое в методе для ссылки на родительский класс получателя.

**UIKit.** Фреймворк, позволяющий разрабатывать приложения для iPhone и iPod touch. Помимо классов для работы с обычными элементами пользовательского интерфейса (UI), такими как окна, кнопки и метки, в нем определены классы для работы со специфическими для устройства средствами, такими как акселерометр и сенсорный интерфейс. UIKit является частью Cocoa Touch.

**Unicode-символ (Unicode character).** Стандарт представления символов из наборов, содержащих до миллионов символов. Классы NSString и NSMutableString работают со строками, содержащими символы Unicode.

**Xcode.** Средство компиляции и отладки для разработки программ в Mac OS X.

**XML.** Сокращение от Extensible Markup Language. Формат по умолчанию для списков свойств, генерируемых в Mac OS X.

**Абстрактный класс (abstract class).** Класс, определенный таким образом, чтобы упростить создание подклассов. Экземпляры создаются из подкласса, а не из абстрактного класса. См. также Конкретный подкласс.

**Автоматическая переменная (automatic variable).** Переменная, для которой автоматически выделяется и затем высвобождается память соответственно при входе и выходе из блока операторов. Автоматические переменные имеют область действия, ограниченную блоком, в котором они определены, и не имеют начального значения по умолчанию. Перед ними можно ставить необязательное ключевое слово `auto`.

**Автоматически высвобождаемый пул (autorelease pool).** Объект, определенный в Foundation framework, который следит за объектами, которые должны быть высвобождены, когда высвобождается сам пул. Объекты добавляются в этот пул при передаче сообщения `autorelease`.

**Архивация (archiving).** Преобразование представления данных объекта в формат, который можно в дальнейшем восстановить (разархивировать).

**Битовое поле (bitfield).** Структура, содержащая одно или несколько целых полей указанной длины в битах. Доступ к битовым полям и работа с ними происходит так же, как с другими элементами структур.

**Блок операторов (statement block).** Один или несколько операторов, заключенных в фигурные скобки. Внутри блока операторов можно объявить локальные переменные, и область их действия ограничивается этим блоком.

**Выражение с сообщением (message expression).** Выражение, заключенное в прямуюгольные скобки, которое указывает объект (получатель) и сообщение, передаваемое этому объекту.

**Глобальная переменная (global variable).** Переменная, определенная вне любого метода или функции в том же исходном файле или в других исходных файлах, где эта переменная объявлена как внешняя (`extern`).

**Делегат (delegate).** Объект, которому поручает выполнить определенное действие другой объект.

**Динамический контроль типов (dynamic typing)**. Определение класса, которому принадлежит объект, на этапе выполнения, а не во время компиляции. См. также Статический контроль типов.

**Динамическое связывание (dynamic binding)**. Определение конкретного метода для вызова с объектом на этапе выполнения, а не во время компиляции.

**Директива (directive)**. В Objective-C – специальная конструкция, которая начинается со знака (@). Примеры директив: @interface, @implementation, @end и @class.

**Заголовочный файл (header file)**. Файл, содержащий общие определения, макросы и объявления переменных, которые включаются в программу с помощью оператора #import или #include.

**Зона (zone)**. Назначаемая область памяти для размещения данных и объектов. Программа может работать с несколькими зонами для более эффективного управления памятью.

**Инкапсуляция (encapsulation)**. См. Инкапсуляция данных.

**Инкапсуляция данных (data encapsulation)**. Концепция, согласно которой данные для объекта сохраняются в его переменных экземпляра и доступ к ним имеют только методы этого объекта. Это позволяет поддерживать целостность данных.

**Интернационализация (internationalization)**. См. Локализация.

**Категория (category)**. Набор методов, сгруппированных под указанным именем. Категории можно использовать как модули определений методов для класса, они могут использоваться для добавления новых методов к существующему классу.

**Класс (class)**. Набор переменных экземпляра и методов, которые имеют доступ к этим переменным. После определения класса можно создавать экземпляры этого класса (то есть объекты).

**Кластер (cluster)**. Абстрактный класс, который группирует набор частных конкретных подклассов, создавая упрощенный интерфейс с пользователем через этот абстрактный класс.

**Коллекция (collection)**. Объект Foundation framework, который является массивом, словарем или набором (множеством). Используется для группирования родственных объектов и работы с ними.

**Конкретный подкласс (concrete subclass)**. Подкласс абстрактного класса. Экземпляры могут создаваться из конкретного подкласса.

**Константная символьная строка (constant character string)**. Последовательность символов, заключенная в кавычки. Если она начинается с символа @, то определяет объект константной символьной строки типа NSConstantString.

**Корневой объект (root object)**. Объект верхнего уровня в иерархии наследования, не имеющий родительского объекта.

**Локализация (localization)**. Процесс подготовки программы для выполнения в определенном географическом регионе, обычно путем перевода сообщений на язык этого региона и адаптации к соответствующим часовым поясам, денежным знакам, форматам даты и т.д. Иногда локализацией называют только пере-

вод на соответствующий язык, а термин *интернационализация* используют для остальной части этого процесса.

**Локальная переменная (local variable)**. Переменная, область действия которой ограничена блоком, в котором она определена. Переменные могут быть локальными в методе, функции или в блоке операторов.

**Массив (array).** Упорядоченный набор значений. Массивы можно определять как базовый тип Objective-C и реализовать как объекты в среде Foundation с помощью классов `NSArray` и `NSMutableArray`.

**Метод (method).** Процедура, которая принадлежит классу и может выполняться путем передачи сообщения объекту-классу или экземплярам из этого класса. См. также Метод класса и Метод экземпляра.

**Метод доступа (accessor method).** Метод, который получает или задает значение переменной экземпляра. Использование методов доступа для получения или задания значений переменных экземпляра согласуется с методологией инкапсуляции данных.

**Метод завода (factory method).** См. Метод класса.

**Метод класса (class method).** Метод (определенный с помощью ведущего знака +), который вызывается для объектов-классов. См. также Метод экземпляра.

**Метод экземпляра (instance method).** Метод, который может быть вызван экземпляром класса. См. также Метод класса.

**Метод-получатель (getter method).** Метод доступа, который считывает значение переменной экземпляра.

**Метод-установщик (setter method).** Метод доступа, который задает значение переменной экземпляра. См. также Метод-получатель.

**Мутабельный объект (mutable object).** Объект, значение которого можно изменять. Foundation framework поддерживает мутабельные и немутабельные массивы, наборы (множества), строки и словари. См. также Немутабельный объект.

**Набор, или множество (set).** Неупорядоченная коллекция уникальных объектов, реализуемая в Foundation с помощью классов `NSSet`, `NSMutableSet` и `NSCountedSet`.

**Назначенный инициализатор (designated initializer).** Метод, который будут вызывать все остальные методы инициализации в данном классе или подклассах (путем передачи сообщений к `super`).

**Наследование (inheritance).** Процесс передачи в подклассы методов и переменных экземпляра из класса, начиная с корневого объекта.

**Немутабельный объект (immutable object).** Объект, значение которого может изменяться. Примеры из Foundation framework: объекты `NSString`, `NSDictionary` и `NSArray`. См. также Мутабельный объект.

**Неформальный протокол (informal protocol).** Набор логически связанных методов, объявленный как категория (часто как категория корневого класса). В отличие от формальных протоколов все методы в неформальном протоколе не обязательно должны быть реализованы. См. также Формальный протокол.

**Нуль-символ (null character).** Символ, значение которого равно 0. Константа нуль-символа обозначается как "\0".

**Объединение (union).** Составной тип данных, например, структура, содержащая элементы, которые сохраняются в одной и той же области памяти. В любой момент времени только один из таких элементов может занимать эту область памяти.

**Объект (object).** Набор переменных и соответствующих методов. Объекту можно передавать сообщения для выполнения одного из его методов.

**Объект-класс (class object).** Объект, который идентифицирует определенный класс. Имя класса может использоваться как получатель сообщения для вызова метода класса. В других случаях метод класса может вызываться применительно к этому классу для создания объекта класса.

**Объектно-ориентированное программирование (object-oriented programming).** Способ программирования, основанный на классах и объектах с выполнением действий над этими объектами.

**Объявление свойств (property declaration).** Способ задания атрибутов для переменных экземпляра, которые позволяют компилятору генерировать для переменных экземпляра методы доступа, не допускающие утечки памяти и конфликта потоков. Объявления свойств можно также использовать, чтобы объявлять атрибуты для методов доступа, которые будут динамически загружаться во время выполнения.

**Оператор (statement).** Одно или несколько выражений, заканчивающихся точкой с запятой.

**Переменная экземпляра (instance variable).** Переменная, которая объявлена в секции `interface` (или унаследована из родительского класса) и содержится в каждом экземпляре объекта. Методы экземпляра имеют непосредственный доступ к своим переменным экземпляра.

**Пересылка (forwarding).** Процесс передачи сообщения и связанных с ним аргументов другому методу для выполнения.

**Подкласс (subclass).** Называют также *дочерним классом*. Подкласс наследует методы и переменные экземпляра из своего родительского класса (суперкласса).

**Подчинение (conform).** Класс подчиняется протоколу (согласуется с ним), если он принимает все обязательные методы в этом протоколе либо путем реализации (`implementation`), либо посредством наследования.

**Полиморфизм (polymorphism).** Способность объектов различных классов принимать одинаковое сообщение.

**Получатель (receiver).** Объект, которому передается сообщение. Получатель может быть указан как `self` изнутри вызываемого метода.

**Препроцессор (preprocessor).** Программа, которая первоначально просматривает исходный код, обрабатывая строки, которые начинаются со знака `#`, то есть предположительно содержат специальные препроцессорные выражения. Обычно применяется для определения макросов с помощью оператора `#define`, включения других исходных файлов с помощью операторов `#import` и `#include`, а также условного включения строк исходного текста с помощью операторов `#if`, `#ifdef` и `#ifndef`.

**Протокол (protocol).** Список методов, которые должны быть реализованы классом для подчинения протоколу или принятия протокола. Протоколы позволяют стандартизовать интерфейс между классами. См. также Формальный протокол и неформальный протокол.

**Процедурный язык программирования (procedural programming language).** Язык, в котором программы определяются с помощью процедур и функций, работающих с набором данных.

**Пустой символ.** См. Нуль-символ.

**Пустой указатель (null pointer).** Значение недействительного указателя, обычно определяемое как 0.

**Распределенные объекты (Distributed Objects).** Способность объектов Foundation в одном приложении взаимодействовать с объектами Foundation в другом приложении, возможно, работающем на другой машине.

**Родительский класс (parent class).** Класс, из которого наследует другой класс. Называется также *суперклассом*.

**Сборка (linking).** Процесс преобразования одного или нескольких объектных файлов в программу, которую можно выполнять.

**Сборка мусора (garbage collection).** Система управления памятью, которая автоматически освобождает память, занятую объектами, ссылки на которые отсутствуют. Сборка мусора не поддерживается в среде runtime iPhone.

**Секция implementation (implementation section).** Секция определения класса, которая содержит конкретный код (то есть реализацию) для методов, объявленных в соответствующей секции interface (или в соответствии с определением протокола).

**Секция interface (interface section).** Секция для объявления класса, его суперкласса, переменных экземпляра и методов. Для каждого метода объявляются также типы аргументов и тип возвращаемого значения. См. также Секция implementation.

**Секция интерфейса.** См. Секция interface.

**Секция реализации.** См. Секция implementation.

**Селектор (selector).** Имя, используемое для выбора метода, который должен быть выполнен для объекта. Компилированные селекторы имеют тип SEL, и они могут генерироваться с помощью директивы @selector.

**Символьная строка (character string).** Последовательность символов, которая заканчивается нуль-символом ("\0").

**Синтезируемый метод (synthesized method).** Метод-установщик (setter) или метод-получатель (getter), который автоматически создается для вас компилятором. Эта возможность была добавлена в язык Objective-C 2.0.

**Словарь (dictionary).** Коллекция пар ключ-значение, реализуемая в Foundation с помощью классов NSDictionary и NSMutableDictionary.

**См. также** Метод-установщик

**Сообщение (message).** Метод и связанные с ним аргументы, которые передаются объекту (получателю сообщения).

**Составной класс (composite class).** Класс, который состоит из объектов других классов; он часто используется как альтернатива подклассов.

**Список свойств (property list).** Представление различных типов объектов в стандартизованном формате. Списки свойств обычно хранятся в формате XML.

**Статическая переменная (static variable).** Переменная, область действия которой ограничена блоком или модулем, в котором она определена. Статические переменные имеют начальное значение по умолчанию, равное 0, и сохраняют свое значение после вызова метода или функции.

**Статическая функция (static function).** Функция, объявляемая с ключевым словом static. Ее могут вызывать только другие функции или методы, определенные в том же исходном файле.

**Статический контроль типов (static typing).** Явное указание класса, которому принадлежит объект, на этапе компиляции. См. также Динамический контроль типов.

**Структура (structure).** Составной тип данных, который может содержать элементы различных типов. Структуры можно присваивать другим структурам, передавать как аргументы функциям и методам, и они могут также возвращаться функциями и методами.

**Суперкласс (super class).** Родительский класс определенного класса. См. также super.

**Счетчик ссылок (reference count).** См. Счетчик удержаний.

**Счетчик удержаний (retain count).** Счетчик числа ссылок на объект. Наращивается путем передачи объекту сообщения retain и уменьшается путем передачи сообщения release.

**Уведомление (notification).** Процесс передачи сообщения объектам, которые зарегистрированы для извещения (уведомления) в случае возникновения определенного события.

**Указатель (pointer).** Значение, которое является ссылкой на другой объект или тип данных. Указатель реализуется как адрес определенного объекта или значения в памяти. Экземпляр класса – это указатель на местонахождение данных объекта в памяти.

**Формальный протокол (formal protocol).** Набор связанных методов, которые сгруппированы под именем, объявленным с помощью директивы @protocol. Различные классы (не обязательно связанные) могут принять формальный протокол путем реализации (или наследования) всех его обязательных методов. См. также Неформальный протокол

**Фреймворк (framework).** Набор классов, функций, протоколов, документации, header-файлов и других ресурсов, которые связаны друг с другом. Например, фреймворк Сосоа используется в разработке интерактивных графических приложений под управлением Mac OS X.

**Функция (function).** Блок операторов, идентифицируемый определенным именем; может принимать один или несколько аргументов, передаваемых в виде значений, и может (не обязательно) возвращать значение. Функции могут быть локальными (статическими) по отношению к файлу, в котором они определены.

ны, или глобальными. Во втором случае их можно вызывать из функций или методов, определенных в других файлах.

**Экземпляр (instance).** Конкретное представление класса. Экземпляры – это объекты, которые обычно создаются путем передачи объекту-классу сообщения alloc или new.

**Этап выполнения (runtime).** Время, когда выполняется программа; runtime – это также механизм, ответственный за выполнение инструкций программы.

**Этап компиляции (compile time).** Этап, во время которого происходит анализ исходного кода и его преобразование в формат более низкого уровня, который называется объектным кодом.

# Приложение В

## Сводка языка Objective-C

В этом приложении содержится краткий справочник по языку Objective-C в удобном формате. Эта не полный справочник, а скорее неформальное описание средств языка. Тщательно проработайте этот материал после того, как закончите работу с основным текстом книги. Это позволит вам не только закрепить изученный материал, но и лучше понять возможности Objective-C.

Описание основывается на стандарте ANSI C99 (ISO/IEC 9899:1999) с расширениями языка Objective-C. На момент написания этой книги в моей системе Mac OS X v10.5.5 последняя версия компилятора GNU gcc имела номер 4.0.1.

### Диграфы и идентификаторы

#### Символы-диграфы

Следующие двухсимвольные последовательности (*диграфы*) эквивалентны указанным односимвольным пунктуаторам.

| Диграф | Значение |
|--------|----------|
| <:     | {        |
| :>     | }        |
| <%     | {        |
| %>     | }        |
| %:     | #        |
| %:%:   | ##       |

## Идентификаторы

*Идентификатор* в Objective-C состоит из последовательности букв (прописных и строчных), имен универсальных символов (см. ниже), цифр и знака подчеркивания. Первый символ идентификатора должен быть буквой, знаком подчеркивания или именем универсального символа. Первые 31 символов идентификатора обязательно будут значащими для внешнего имени, первые 63 символа обязательно будут значащими для внутреннего имени или имени макроса.

### Имена универсальных символов

Имя универсального символа состоит из символов \u, после которых следуют четыре шестнадцатеричных числа, или из символов \U, после которых следуют восемь шестнадцатеричных чисел. Если первый символ идентификатора указан универсальным символом, его значение не может быть символом цифры. Универсальные символы при использовании в именах идентификаторов не могут указывать символ, значение которого меньше A0<sub>16</sub> (в отличие от 24<sub>16</sub>, 40<sub>16</sub> или 60<sub>16</sub>), или символ в диапазоне от D800<sub>16</sub> до DFFF<sub>16</sub> включительно.

Имена универсальных символов можно использовать в именах идентификаторов, символьных константах и символьных строках.

### Ключевые слова

Приведенные ниже идентификаторы являются ключевыми словами, представляющими специальное значение для компилятора Objective-C.

```
_Bool
_Complex
_Imaginary
auto
break
bycopy
byref
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
in
inline
```

---

```

inout
int
long
oneway
out

register
restrict
return
self
short
signed
sizeof
static
struct
super
switch
typedef
union
unsigned
void
volatile
while

```

### Директивы

Директивы компилятора начинаются со знака @ и используются особым образом для работы с классами и обработками, см. таблицу В.1.

**Табл. В.1.** Директивы компилятора

| Директива          | Описание                                                                                             | Пример                                                                                            |
|--------------------|------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| @"символы"         | Определяет константный строковый объект типа NSSTRING. (Для смежных строк выполняется конкатенация.) | NSString *url = @"http://www.kochan-wood.com";                                                    |
| @class c1, c2, ... | Объявляет c1, c2, ... как классы.                                                                    | @class Point, Rectangle;                                                                          |
| @defs (класс)      | Возвращает список структурных переменных для класса.                                                 | struct Fract { @defs(Fraction); } *fractPtr; fractPtr = (struct Fract *) [[Fraction alloc] init]; |
| @dynamic имена     | Методы доступа для имен могут предоставляться динамически.                                           | @dynamic drawRect;                                                                                |
| @encode (тип)      | Строковое кодирование для типа.                                                                      | @encode (int *)                                                                                   |
| @end               | Заканчивает секцию interface, секцию implementation или секцию протокола.                            | @end                                                                                              |

Табл. В.1. Директивы компилятора (окончание)

| Директива                   | Описание                                                                                                                               | Пример                                                          |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| @implementation             | Начинает секцию implementation.                                                                                                        | @implementation Fraction                                        |
| @interface                  | Начинает секцию interface.                                                                                                             | @interface Fraction: NSObject<Copying>                          |
| @private                    | Определяет область действия одной или нескольких переменных экземпляра.                                                                | См. «Переменные экземпляра»                                     |
| @protected                  | Определяет область действия одной или нескольких переменных экземпляра.                                                                |                                                                 |
| @public                     | Определяет область действия одной или нескольких переменных экземпляра.                                                                |                                                                 |
| @property<br>(список) имена | Объявляет свойства в списке для имен.                                                                                                  | property (retain, nonatomic)<br>NSSTRING *name;                 |
| @protocol                   | Создает объект типа Protocol для указанного протокола.                                                                                 | @protocol (Copying)]}{...}<br>if ([myObj conformsTo: (protocol) |
| @protocol имя               | Начинает определение протокола для имени.                                                                                              | @protocol Copying                                               |
| @selector<br>(метод)        | Объект типа SEL для указанного метода.                                                                                                 | if ([myObj respondsToSelector: @selector (allocF)]) {...}       |
| @synchronized<br>(объект)   | Начинает блок, который должен выполняться одним потоком.<br>Объект называется взаимоисключающим (mutually exclusive, mutex) семафором. |                                                                 |
| @synthesize<br>имена        | Генерирует методы доступа для имен, если они не заданы.                                                                                | @synthesize name, email;<br>См. также «Переменные экземпляра»   |
| @try                        | Начинает блок для улавливания исключительных ситуаций (исключений).                                                                    | См. «Обработка исключительных ситуаций»                         |
| @catch<br>(исключение)      | Начинает блок для обработки исключения.                                                                                                |                                                                 |
| @finally                    | Начинает блок, который выполняется в предположении, что предыдущем блоке @try было инициировано» (throw) исключение.                   |                                                                 |
| @throw                      | Инициирует исключение.                                                                                                                 |                                                                 |

### Предопределенные идентификаторы

В таблице В.2 приводится список идентификаторов, которые имеют специальный смысл в программах на Objective-C.

**Табл. В.2. Специальные предопределенные идентификаторы**

| Идентификатор | Описание                                                                                                                       |
|---------------|--------------------------------------------------------------------------------------------------------------------------------|
| _cmd          | Локальная переменная, автоматически определяемая в методе, который содержит селектор для этого метода.                         |
| _func_        | Локальная символьная строковая переменная, автоматически определяемая в функции или в методе; содержит имя функции или метода. |
| BOOL          | Булева переменная, обычно со значениями YES и NO.                                                                              |
| Class         | Тип объекта-класса.                                                                                                            |
| id            | Обобщенный тип объекта.                                                                                                        |
| IMP           | Указатель на метод, возвращающий значение типа id.                                                                             |
| nil           | Пустой объект.                                                                                                                 |
| Nil           | Пустой объект-класс.                                                                                                           |
| NO            | Определяется как (BOOL) 0.                                                                                                     |
| NSObject      | Корневой объект Foundation, определенный в <Foundation/NSObject.h>.                                                            |
| Protocol      | Имя класса для хранения информации о протоколах.                                                                               |
| SEL           | Откомпилированный селектор.                                                                                                    |
| self          | Локальная переменная, автоматически определяемая в методе, которая обозначает получателя сообщения.                            |
| super         | Родительский объект получателя сообщения.                                                                                      |
| YES           | Определяется как (BOOL) 1.                                                                                                     |

## Комментарии

Имеются два способа вставки комментариев в программу. Комментарий может начинаться с двух символов //, и в этом случае любые последующие символы этой строки игнорируются компилятором.

Комментарий может также начинаться с символов /\* и заканчиваться символами \*/. В такой комментарий можно включать любые символы, и он может занимать несколько строк. Комментарий можно использовать в любом месте программы, если для этого есть место. Однако комментарии не допускают вложений. Это означает, что комментарий всегда заканчивается символами \*/, даже если до этого встретилось несколько наборов символов \*/.

## Константы

### Константы целого типа

Константа целого типа (целая константа) – это состоящая из цифр последовательность, перед которой может дополнительно ставиться знак «плюс» или «минус». Если первой цифрой является 0, значит, это восьмеричная константа, и тогда следующие цифры должны быть в диапазоне 0-7. Если первой цифрой является 0, и сразу после нее следует буква x (или X), значит, это шестнадцатеричная константа, и последующие цифры должны быть в диапазоне 0-9, a-f (или A-F).

В конце десятичной целой константы можно добавить суффикс l или L, и тогда она становится константой типа long int, если умещается в этот размер, иначе она интерпретируется как long long int. И, наконец, если она не умещается в long long int, она интерпретируется как константа типа unsigned long long int.

В конце десятичной целой константы можно добавить суффикс ll или LL, чтобы сделать ее long long int. При добавлении этого суффикса в конец восьмеричной или шестнадцатеричной константы, она интерпретируется сначала как long long int, но если не умещается в этот размер, то получает тип unsigned long long int.

В конце целой константы можно добавить суффикс u или U, чтобы сделать ее константой без знака (unsigned). Если она не умещается в размер unsigned int, она интерпретируется как unsigned long int. Если она превышает также этот размер, то интерпретируется как unsigned long long int.

Два суффикса – для unsigned и для long – можно добавить к целой константе, чтобы она имела тип unsigned long int. Если константа не умещается в этот тип, она обрабатывается как unsigned long long int.

Два суффикса – для unsigned и для long long – можно добавить к целой константе, чтобы она имела тип unsigned long long int.

Если десятичная целая константа без суффиксов слишком велика, чтобы уместиться в тип signed int, она интерпретируется как long int. Если она слишком велика, чтобы уместиться в тип long int, она интерпретируется как long long int.

Если восьмеричная или шестнадцатеричная целая константа без суффиксов слишком велика, чтобы уместиться в тип signed int, она интерпретируется как unsigned int. Если она слишком велика, чтобы уместиться в тип unsigned int, она интерпретируется как long int, и если она не умещается в этот тип, то интерпретируется как unsigned long int. Если она не умещается в unsigned long int, то интерпретируется как long long int. И, наконец, если она слишком велика для long long int, то интерпретируется как unsigned long long int.

### Константы с плавающей точкой

Константа с плавающей точкой состоит из последовательности десятичных цифр, десятичной точки и еще одной последовательности десятичных цифр. Перед значением можно ставить знак «минус» для отрицательных значений. Кроме того, может отсутствовать последовательность цифр до десятичной точки или после нее, но не обе последовательности.

Если в конце константы с плавающей точкой добавлены буква e (или E) и целое число (с необязательным знаком), константа имеет экспоненциальное

представление. Это целое число (*порядок*) указывает, что значение до буквы e (*мантиssa*) умножается на 10 в степени, равной этому целому числу (например, 1.5e-2 представляет  $1.5 \times 10^{-2}$ , или .015).

Шестнадцатеричная константа с плавающей точкой состоит из ведущих символов 0x или 0X, после которых следуют одна или несколько десятичных или шестнадцатеричных цифр, затем буква r или R и затем показатель степени для основания 2 с возможным знаком. Например, 0x3р10 представляет значение  $3 \times 2^{10}$ . Константы с плавающей точкой интерпретируются компилятором как значения с двойной точностью (double). Можно добавить суффикс f или F, и тогда константа будет иметь тип float вместо double; можно добавить суффикс l или L, чтобы указать константу типа long double.

## Символьные константы

Один символ, заключенный в апострофы, является символьной константой. Обработка нескольких символов, заключенных в апострофы, определяется реализацией. В символьной константе можно использовать универсальный символ, чтобы указать символ, не входящий в стандартный набор символов.

### Escape-последовательности

Специальные escape-последовательности распознаются и вводятся с помощью символа «обратный слэш». Ниже приводятся эти escape-последовательности.

| Символ   | Описание                                         |
|----------|--------------------------------------------------|
| \a       | Звуковой сигнал                                  |
| \b       | Символ Backspace (возврат на одну позицию назад) |
| \f       | Form feed (подача страницы)                      |
| \n       | Переход на новую строку                          |
| \r       | Символ возврата каретки                          |
| \t       | Горизонтальное табулирование                     |
| \v       | Вертикальное табулирование                       |
| \\"      | Обратный слэш                                    |
| \“       | Кавычка                                          |
| \‘       | Апостроф                                         |
| \?       | Вопросительный знак                              |
| \ппп     | Восьмеричное значение символа                    |
| \ппппп   | Имя универсального символа                       |
| \Uпппппп | Имя универсального символа                       |
| \хпп     | Шестнадцатеричное значение символа               |

Для восьмеричного значения символа можно указать от одной до трех восьмеричных цифр. В последних трех случаях используются шестнадцатеричные цифры.

### Символьные константы из расширенных наборов

Символьная константа из расширенных наборов записывается как L'x'. Такая константа имеет тип wchar\_t, в соответствии с определением в файле <stddef.h>. Константы из расширенных наборов символов – это способ представить символ из набора символов, который не может быть полностью представлен с помощью обычного типа char.

## Константы символьных строк

Последовательность, содержащая нуль или более символов, заключенных в кавычки, представляет константу символьной строки (строковую константу). В эту строку можно включить любой допустимый символ, в том числе любой из показанных выше escape-символов. Компилятор автоматически вставляет нуль-символ ('\0') в конце этой строки.

Обычно компилятор создает указатель на первый символ строки, имеющий тип «указатель на char». Но если строковая константа используется с оператором sizeof для инициализации массива символов или с оператором &, то строковая константа имеет тип «массив элементов типа char».

Константы символьных строк не могут быть изменены программой.

### Конкатенация символьных строк

Препроцессор автоматически выполняет конкатенацию смежных строковых констант. Строки могут быть любым числом пробелов. Таким образом, три строки

```
"a" "character"
"string"
```

эквивалентны одной строке

```
"a character string"
```

после конкатенации.

### Многобайтные символы

Для перехода между различными состояниями в символьной строке можно использовать последовательности символов, определенные реализацией, что позволяет включать многобайтные символы.

### Строковые константы из расширенных наборов

Строковые константы из расширенного набора символов представляются в формате L'...'. Каждая такая константа имеет тип «указатель на wchar\_t», где wchar\_t определяется в <stddef.h>.

### Объекты константных символьных строк

Для создания *объекта* константной символьной строки нужно поместить символ @ перед константной символьной строкой. Этот объект имеет типа NSConstantString.

Для смежных объектов константных символьных строк выполняется конкатенация. Таким образом, три строковых объекта

```
@"a" @"character"
 @"string"
```

эквивалентны одному строковому объекту

```
@"a character string"
```

### Константы перечислимого типа

Идентификатор, который объявлен как значение для перечислимого типа, рассматривается как константа этого типа, но интерпретируется компилятором как тип int.

## Типы и объявления данных

В этом разделе приводится сводка базовых типов данных, производных типов данных, перечислимых типов данных, а также typedef. Здесь также приводится формат объявления переменных.

### Объявления

При определении какой-либо структуры, объединения (union), перечислимого типа данных или typedef компилятор не резервирует память автоматически. Из определения компилятор просто получает информацию об определенном типе данных и (необязательно) связывает с ним определенное имя. Такое определение можно делать внутри или вне функции или метода. В первом случае только данная функция или метод «знает» о его существовании; во втором случае определение распространяется на весь файл.

После того, как создано определение, можно объявлять переменные с этим типом данных. Для переменной любого типа данных будет выделена память, если только это не объявление extern (в этом случае память может быть выделена или не выделена, см. ниже раздел «Классы памяти и область действия»).

Язык Objective-C позволяет также выделять память одновременно с объявлением определенной структуры, объединения или перечислимого типа данных. Для этого нужно просто перечислить соответствующие переменные, прежде чем завершить определение символом «точка с запятой».

## Базовые типы данных

Базовые типы данных приводятся в таблице В.3. Чтобы объявить переменную с определенным типом данных, нужно использовать формат

`тип имя = начальное_значение;`

Присваивание начального значения переменной не является обязательным и подчиняется правилам, описанным ниже в разделе «Переменные». Для объявления нескольких переменных используется формат

`тип имя = начальное_значение, имя = начальное_значение, ...;`

Перед объявлением типа можно дополнительно указать класс памяти, что описывается ниже в разделе «Переменные». Если указан класс памяти и переменная должна иметь тип `int`, то `int` можно опустить. Например,

`static counter;`

объявляет `counter` как переменную `static int`.

**Табл. В.3.** Сводка базовых типов данных

| Тип                        | Описание                                                                                                                                     |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int</code>           | Целое значение, то есть значение, не содержащее десятичной точки; имеет точность не менее 32 бит.                                            |
| <code>short int</code>     | Целое значение, которое занимает половину памяти по сравнению с типом <code>int</code> на некоторых машинах; имеет точность не менее 16 бит. |
| <code>long int</code>      | Целое значение увеличенной точности; имеет точность не менее 32 бит.                                                                         |
| <code>long long int</code> | Целое значение дополнительно увеличенной точности; имеет точность не менее 64 бит.                                                           |
| <code>unsigned int</code>  | Положительное целое значение; может содержать положительные значения, вдвое больше, чем <code>int</code> ; имеет точность не менее 32 бит.   |

Табл. В.3. Сводка базовых типов данных (окончание)

| Тип                  | Описание                                                                                                                                                                                                              |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| float                | Значение с плавающей точкой, то есть может содержать знаки после десятичной точки; имеет точность не менее 6 цифр.                                                                                                    |
| double               | Значение с плавающей точкой увеличенной точности; имеет точность не менее 10 цифр.                                                                                                                                    |
| long double          | Значение с плавающей точкой дополнительно увеличенной точности; имеет точность не менее 10 цифр.                                                                                                                      |
| char                 | Значение с одним символом; на некоторых машинах может добавляться знак при использовании в выражении.                                                                                                                 |
| unsigned char        | То же самое, что и char, за исключением того, что в случае преобразования в целое значение не будет добавляться знак.                                                                                                 |
| signed char          | То же самое, что и char, за исключением того, что в случае преобразования в целое значение будет добавляться знак.                                                                                                    |
| _Bool                | Булев тип; хранит значение 0 или 1.                                                                                                                                                                                   |
| float _Complex       | Комплексное число.                                                                                                                                                                                                    |
| double _Complex      | Комплексное число увеличенной точности.                                                                                                                                                                               |
| long double _Complex | Комплексное число дополнительно увеличенной точности.                                                                                                                                                                 |
| void                 | Не обозначает никакого типа; используется при объявлении функции или метода, не возвращающих никакого значения, или для явной отмены результатов выражения; используется также как обобщенный тип указателя (void *). |

Отметим, что модификатор signed можно также указывать перед типами short int, int, long int и long long int. Поскольку эти типы всегда используются со знаком, модификатор не оказывает на них дополнительного влияния.

Типы данных \_Complex и \_Imaginary позволяют объявлять комплексные и мнимые числа и работать с ними с помощью функций из библиотеки, поддерживающей арифметические операции с этими типами. Обычно следует включать в свою программу файл <complex.h>, где определяются макросы и объявляются функции для работы с комплексными и мнимыми числами. Например, переменную c1 типа double \_Complex можно объявить и инициализировать со значением 5 + 10.5i с помощью оператора

```
double _Complex c1 = 5 + 10.5 * i;
```

Затем можно использовать библиотечные процедуры creal и cimag для извлечения вещественной и мнимой частей c1 соответственно.

Используемая реализация не обязательно поддерживает типы `_Complex` и `_Imaginary`. Она может поддерживать один тип и не поддерживать другой.

## Производные типы данных

*Производный (derived) тип данных* формируется из одного или нескольких базовых типов данных. Производными типами данных являются массивы, структуры, объединения и указатели (включая объекты). Функция или метод, которые возвращают значение указанного типа, также считаются производным типом данных. Функции и классы описываются по отдельности соответственно в разделах «Функции» и «Классы».

### Массивы

#### Одномерные массивы

В определении массива может содержаться любой базовый тип данных или любой производный тип данных. Массивы функций не допускаются (хотя разрешается использовать массивы указателей на функции).

Объявление массива имеет следующий основной формат.

```
тип имя[l] = { Выражение_инициализации, Выражение_инициализации, .. };
```

Выражение *l* определяет число элементов массива с этим именем, и его можно опустить, если задан список начальных значений. В этом случае размер массива определяется числом заданных начальных значений или элементом с максимальным указанным номером, если используются назначенные инициализаторы.

Если определяется глобальный массив, каждое начальное значение должно быть константным выражением. Число значений в списке инициализации может быть меньше числа элементов массива (но не больше). Если задано меньше значений, то инициализируется только соответствующее число элементов массива; остальным элементам присваивается значение 0.

Особым случаем инициализации массива являются массивы символов, которые можно инициализировать с помощью константной символьной строки. Например,

```
char today[] = "Monday";
```

объявляет *today* как массив символов. Элементы этого массива инициализируются в виде начальных значений соответственно символами 'M', 'o', 'n', 'd', 'a', 'y' и '\0'.

Если объявить явным образом размер массива, не оставив элемента для конечного нуль-символа, то компилятор не поместит нуль-символ в конце массива.

```
char today[6] = "Monday";
```

В этом объявлении *today* является массивом из шести символов, и его элементам присваиваются соответственно символы 'M', 'o', 'n', 'd', 'a' и 'y'.

Указывая элементы в прямоугольных скобках, можно инициализировать конкретные элементы массива в любом порядке. Например, ниже объявляется

массив из 10 элементов с именем a (число элементов массива определяется элементом с максимальным указанным номером), а также задается начальное значение  $x + 1$  (1234) для последнего элемента и значения 1, 2, 3 соответственно для первых трех элементов.

```
int x = 1233;
int a[] = { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

### Массивы переменной длины

Внутри функции, метода или блока можно задать размер массива, используя выражение, содержащее переменные. В этом случае размер вычисляется на этапе выполнения. Например, функция

```
int makeVals (int n)
{
 int valArray[n];
 ...
}
```

определяет автоматический массив с именем valArray и размером n элементов, где n вычисляется на этапе выполнения программы и может быть различным при различных вызовах этой функции. Массивы переменной длины нельзя инициализировать.

### Многомерные массивы

Объявление многомерного массива имеет основной формат

тип имя[n] = [d1][d2]...[dn] = списокИнициализации;

Здесь определяется, что массив с этим именем содержит  $d_1 \times d_2 \times \dots \times d_n$  элементов указанного типа. Например,

```
int three_d [5][2][20];
```

определяет трехмерный массив three\_d, содержащий 200 целых элементов.

Отдельный элемент многомерного массива указывается набором индексов каждой размерности, каждый из которых заключен в отдельные прямоугольные скобки. Например, выражение

```
three_d [4][0][15] = 100;
```

присваивает значение 100 указанному элементу массива three\_d.

Многомерные массивы можно инициализировать таким же образом, как одномерные. Для управления присваиванием значений элементам массива можно использовать вложенные пары фигурных скобок.

Ниже матрица определяется как двумерный массив, содержащий четыре строки и три столбца.

```
int matrix[4][3] =
{ { 1, 2, 3 },
{ 4, 5, 6 },
{ 7, 8, 9 } };
```

Элементам первой строки матрицы присваиваются соответственно значения 1, 2, 3; элементам второй строки – 4, 5, 6; и элементам третьей строки – 7, 8, 9. Элементам четвертой строки присваивается значение 0, поскольку для этой строки значения не заданы. Объявление

```
int matrix[4][3] =
{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

инициализирует матрицу, содержащую те же значения, поскольку элементы многомерного массива инициализируются в порядке размерностей, то есть от левой размерности к правой. Объявление

```
int matrix[4][3] =
{ { 1 },
{ 4 },
{ 7 } };
```

присваивает 1-му элементу 1-й строки матрицы значение 1, 1-му элементу 2-й строки – значение 4 и 1-му элементу 3-й строки – значение 7. Всем остальным элементам присваивается по умолчанию значение 0.

И, наконец, объявление

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

инициализирует для указанных элементов матрицы заданные значения.

## Структуры

### Общий формат

```
struct имя
{
 объявлениеКомпонента
 объявлениеКомпонента
 ...
} списокПеременных;
```

Для структуры с заданным именем указывается, что она содержит компоненты в соответствии с каждым объявлениемКомпонента. Каждое такое объявление содержит указание типа, после которого следует список из одного или нескольких имен компонентов.

Переменные можно объявить непосредственно во время определения структуры, указав их перед завершающим символом «точка с запятой», или их можно объявить в дальнейшем, используя формат

```
struct имя список_именПеременных,
```

Этот формат нельзя использовать, если в определении структуры не указано имя. В этом случае все переменные структуры должны быть объявлены с помощью определения.

Формат инициализации переменной-структурой аналогичен формату для массивов. Компоненты структуры можно инициализировать, заключив список начальных значений в фигурные скобки. Каждое значение в этом списке должно быть константным выражением, если инициализируется глобальная структура.

В объявлении

```
struct point
{
 float x;
 float y;
} start = {100.0, 200.0};
```

определяется структура с именем `point` и переменная-структура с именем `start` с указанными начальными значениями. Конкретные компоненты структуры можно задавать для инициализации в любом порядке с помощью формы записи

`.компонент = значение`

в списке инициализации, например,

```
struct point end = { .y = 500, .x = 200 };
```

В объявлении

```
struct entry
{
 char *word;
 char *def;
} dictionary[1000] = {
 {"a", "first letter of the alphabet"},
 {"aardvark", "a burrowing African mammal"},
 {"aback", "to startle"}
};
```

объявляется словарь, содержащий 1000 структурных записей, причем первые три элемента инициализируются как указатели на указанные символьные строки. Используя назначенные инициализаторы, можно записать это следующим образом.

```
struct entry
{
 char *word;
 char *def;
} dictionary[1000] = {
 [0].word = "a", [0].def = "first letter of the alphabet",
 [1].word = "aardvark", [1].def = "a burrowing African mammal",
 [2].word = "aback", [2].def = "to startle"
};
```

или в эквивалентной форме

```
struct entry
{
 char *word;
 char *def;
} dictionary[1000] = {
 { .word = "a", .def = "first letter of the alphabet" },
 { .word = "aardvark", .def = "a burrowing African mammal" },
 { .word = "aback", .def = "to startle" }
};
```

Автоматическую переменную-структуру можно инициализировать, присвоив ей другую структуру того же типа, например,

```
struct date tomorrow = today;
```

Здесь объявляется переменная-структура типа `date` с именем `tomorrow`, и ей присваивается содержимое (ранее объявленной) переменной-структуры типа `date` с именем `today`.

*Объявление Компонента*, которое имеет формат

*тип имяПоля: п*

определяет *поле* длиной *п* бит внутри структуры, где *п* – целое значение. Поля могут располагаться слева направо на одних машинах и справа налево на других. Если *имяПоля* опущено, то резервируется указанное число битов, но без ссылки. Если *имяПоля* опущено и *п* равно 0, то следующее поле выравнивается по границе следующего блока памяти, где размер блока зависит от реализации. *Поле* может иметь тип `int`, `signed int` или `unsigned int`. В зависимости от реализации поле типа `int` интерпретируется как `signed` или `unsigned`. К полю нельзя применять адресный оператор (`&`), и нельзя определять массивы полей.

## Объединения

### Общий формат

```
union имя
{
 объявление Компонента
 объявление Компонента
 ...
} списокПеременных;
```

Здесь определяется объединение с заданным *именем* и *компонентами*, соответствующими каждому *объвлению Компонента*. Каждый компонент объединения использует общую область памяти, и компилятор обеспечивает резервирование области памяти, достаточной, чтобы вместить самый большой компонент данного объединения.

Переменные можно объявить непосредственно во время определения объединения, или их можно объявить в дальнейшем, используя форму записи

`union имя список именПеременных;`

при условии, что объединению было присвоено имя при его определении.

Программист сам должен следить за тем, чтобы значение, считываемое из объединения, соответствовало типу значения, которое было в последний раз сохранено в объединении. Первый компонент объединения можно инициализировать, заключив в фигурные скобки начальное значение, которое в случае глобальной `union`-переменной должно быть константным выражением.

```
union shared
{
 long long int l;
 long int w[2];
} swap = { 0xffffffff };
```

Для инициализации другого компонента нужно указать имя этого компонента, как в

```
union shared swap2 = {.w[0] = 0x0, .w[1] = 0xffffffff};
```

Здесь объявляется union-переменная swap, и ее компоненту l присваивается шестнадцатеричное значение ffffff.

Автоматическую инициализацию можно также инициализировать, присвоив ей объединение того же типа, например,

union shared swap2 = swap;

## Указатели

Объявление переменной-указателя имеет базовый формат

**тип \*имя**

Идентификатор *имя* объявляется с типом «указатель на *тип*»; *типов* может быть какой-либо базовый тип данных или производный тип данных. Например,

**int \*in:**

объявляет `ip` как указатель на тип `int`, и

struct entry \*en;

объявляет ер как указатель на структуру entry. Если определен класс Fraction, то

Fraction \*myFract;

объявляет `myFrac1` как объект типа `Fraction`, или, точнее, `myFrac1` используется для хранения указателя на структуру данных объекта после того, как создан экземпляр объекта и он присвоен этой переменной.

Указатели, которые указывают на элементы массива, объявляются для указания на тип элемента, содержащегося в этом массиве. Например, предыдущее объявление `ip` можно было бы также использовать для объявления указателя на массив целых элементов.

Допускаются также более сложные формы объявлений указателей. Например,

```
char *tp[100];
```

объявляет `tp` как массив из 100 указателей на тип `char`, и

```
struct entry (*fnPtr) (int);
```

объявляет `fnPtr` как указатель на функцию, которая возвращает структуру `entry` и принимает один аргумент типа `int`.

Можно проверить, является ли указатель пустым (`null`), сравнив его с константным выражением, значение которого равно 0. В реализации можно выбрать внутреннее представление `null`-указателя в виде значения, отличного от 0. Однако сравнение между указателем с таким внутренним представлением и константным значением 0 должно показывать равенство.

Преобразование указателей в целые значения и целых значений в указатели, а также размер целой переменной, необходимой для хранения указателя, зависят от типа машины.

Тип «указатель на `void`» — это обобщенный тип указателя. Язык Objective-C гарантирует, что указатель любого типа может быть присвоен `void`-указателю, и затем можно выполнить обратное присваивание без изменения его значения.

Тип `id` — это обобщенный указатель на объект. Любой объект любого класса может быть присвоен `id`-переменной, и наоборот.

За исключением этих двух особых случаев присваивание указателей различного типа не допускается, и обычно при такой попытке компилятор выдает предупреждающее сообщение.

## Перечислимые типы данных

### Общий формат

```
enum имя { enum_1, enum_2, .. } списокПеременных;
```

Перечислимый тип с заданным именем определяется с перечисляемыми значениями `enum_1`, `enum_2`..., каждое из которых является идентификатором или идентификатором, после которого следует знак равенства и константное выражение. Можно также объявить списокПеременных (с необязательными начальными значениями), и эти переменные будут иметь тип enum имя.

Компилятор присваивает перечисляемым идентификаторам последовательные целые значения, начиная с 0. Если идентификатор указан со знаком = и константным выражением, то идентификатору присваивается значение этого выражения. Последующим идентификаторам присваиваются значения, начиная со значения этого константного выражения плюс 1. Перечисляемые идентификаторы интерпретируются компилятором как константные целые значения.

Если нужно объявить переменную определенного ранее (и именованного) перечислимого типа, можно использовать конструкцию

**enum имя списокПеременных;**

Если объявлена переменная определенного перечислимого типа, то ей можно присвоить значение только того же типа, хотя компилятор, возможно, не зарегистрирует это как ошибку.

## **typedef**

Оператор **typedef** используется, чтобы присвоить новое имя базовому или производному типу данных. Этот оператор не определяет какой-либо новый тип, а просто назначает новое имя для существующего типа. Поэтому переменные, объявленные с типом, имеющим новое имя, интерпретируются компилятором точно так же, как с типом, для которого дополнительно назначено это новое имя.

Создавая определение **typedef**, нужно действовать, как при объявлении обычной переменной. При этом нужно поместить новое имя типа там, где обычно ставится имя переменной. И, наконец, перед всем этим нужно поставить ключевое слово **typedef**.

В показанном ниже примере

```
typedef struct
{
 float x;
 float y;
} POINT;
```

имя **POINT** связывается со структурой, содержащей два компонента типа **float** с именами **x** и **y**. Затем можно объявлять переменные типа **POINT**, например,

```
POINT origin = { 0.0, 0.0 };
```

## **Модификаторы типа: const, volatile и restrict**

Перед объявлением типа можно поместить ключевое слово **const**, чтобы сообщить компилятору, что это значение нельзя изменять. Например,

```
const int x5 = 100;
```

объявляет **x5** как постоянную целую переменную – ей нельзя присвоить другое значение во время выполнения программы. Компилятор не обязательно будет реагировать на попытки изменения значения такой переменной.

Модификатор **volatile** явно указывает компилятору, что значение изменяется (обычно динамически). Если в выражении используется **volatile**-переменная, доступ к ее значению разрешается в любом месте, где она появляется.

Чтобы объявить переменную **port17** как тип «**volatile**-указатель на **char**», можно написать следующую строку.

```
char *volatile port17;
```

Указателями можно использовать ключевое слово **restrict**. Это указание компилятору на возможность оптимизации (аналогично ключевому слову **register** для переменных). Ключевое слово **restrict** сообщает компилятору, что данный указатель будет единственной ссылкой на определенный объект, то есть не бу-

дет указываться никаким другим указателем в пределах той же области действия.  
Строки

```
int * restrict intptrA;
int * restrict intptrB;
```

сообщают компилятору, что в пределах области действия, где определены intptrA и intptrB, они никогда не будут указывать на одно и то же значение. Их использование для указания целых значений (например, в массиве) будет взаимоисключающим.

## Выражения

Имена переменных, имена функций, выражения в виде сообщений, имена массивов, константы, вызовы функций, ссылки на массивы, а также ссылки на структуры и объединения – все это интерпретируется как выражения. Применение унарного (одноместного) оператора к этим выражениям (в допустимых случаях) тоже является выражением, как и сочетание этих выражений с помощью бинарного (двуместного) или тернарного (трехместного) оператора. И, наконец, выражение, заключенное в круглые скобки, тоже является выражением.

Отличное от void выражение любого типа, которое идентифицирует объект данных, называется *lvalue* (*l*-значением). Если ему может быть присвоено значение, оно называется *модифицируемым* (изменяемым) *lvalue*.

Модифицируемые *lvalue*-выражения обязательны в определенных местах. Выражение в левой части оператора присваивания должно быть модифицируемым *lvalue*. Унарный адресный оператор можно применить только к модифицируемом *lvalue* или к имени функции. И, наконец, операторы наращивания и уменьшения на 1 (инкремента и декремента) можно применять только к модифицируемым *lvalue*-выражениям.

## Сводка операторов Objective-C

В таблице В.4 приводится сводка различных операторов языка Objective-C. Эти операторы приводятся в порядке убывания старшинства, и операторы, сгруппированные вместе, имеют одинаковый уровень старшинства.

Рассмотрим на следующем примере, как использовать таблицу В.4.

b | c & d \* e

Оператор умножения имеет более высокий приоритет, чем побитовые операторы OR (|) и AND (&), поскольку он находится выше обоих операторов в таблице В.4. Аналогичным образом, побитовый оператор AND имеет более высокий приоритет, чем побитовый оператор OR, поскольку первый находится в этой таблице выше, чем второй. Поэтому данное выражение вычисляется в следующем порядке.

b | (c & (d \* e))

Теперь рассмотрим следующее выражение.

b % c \* d

Табл. В.4. Сводка операторов Objective-C

| Оператор | Описание                                             | Ассоциативность |
|----------|------------------------------------------------------|-----------------|
| ()       | Вызов функции                                        |                 |
| []       | Ссылка на элемент массива или выражение с сообщением |                 |
| ->       | Указатель ссылки на компонент структуры              | Слева направо   |
| .        | Ссылка на компонент структуры или вызов метода       |                 |
| -        | Унарный минус                                        |                 |
| +        | Унарный плюс                                         |                 |
| ++       | Оператор инкремента                                  |                 |
| --       | Оператор декремента                                  |                 |
| !        | Логическое отрицание                                 |                 |
| ~        | Дополнение до нуля                                   | Справа налево   |
| *        | Ссылка на указатель (косвенным образом)              |                 |
| &        | Адрес                                                |                 |
| sizeof   | Размер объекта                                       |                 |
| (тип)    | Приведение типа (преобразование)                     |                 |
| *        | Умножение                                            |                 |
| /        | Деление                                              | Слева направо   |
| %        | Взятие по модулю                                     |                 |
| +        | Сложение                                             | Слева направо   |
| -        | Вычитание                                            |                 |
| <<       | Левый сдвиг                                          | Слева направо   |
| >>       | Правый сдвиг                                         |                 |
| <        | Меньше                                               |                 |
| <=       | Меньше или равно                                     | Слева направо   |
| >        | Больше                                               |                 |
| >=       | Больше или равно                                     |                 |
| ==       | Равенство                                            | Слева направо   |
| !=       | Неравенство                                          |                 |
| &        | Побитовый AND                                        | Слева направо   |

Табл. В.4. Сводка операторов Objective-C (окончание)

| Оператор                                                                                                       | Описание               | Ассоциативность |
|----------------------------------------------------------------------------------------------------------------|------------------------|-----------------|
| <code>^</code>                                                                                                 | Побитовый XOR          | Слева направо   |
| <code> </code>                                                                                                 | Побитовый OR           | Слева направо   |
| <code>&amp;&amp;</code>                                                                                        | Логический AND         | Слева направо   |
| <code>  </code>                                                                                                | Логический OR          | Слева направо   |
| <code>?:</code>                                                                                                | Условный               | Справа налево   |
| <code>= *= /=</code><br><code>%= +=</code><br><code>-= &amp;= ^=  =</code><br><code>&lt;&lt;= &gt;&gt;=</code> | Операторы присваивания | Справа налево   |
| <code>,</code>                                                                                                 | Оператор «запятая»     | Справа налево   |

Поскольку операторы взятия по модулю и умножения включены в одну группу таблицы В.4, они имеют одинаковый уровень старшинства. Эти операторы выполняются слева направо. Следующее выражение будет вычисляться как

`( b % c ) * d`

В следующем примере выражение

`++a->b`

будет вычисляться как

`++(a->b)`

поскольку оператор `->` имеет более высокий приоритет, чем оператор `++`.

И, наконец, поскольку операторы присваивания группируются справа налево, оператор

`a = b = 0;`

будет вычисляться как

`a = (b = 0);`

в результате чего переменным `a` и `b` будет присвоено значение 0. В случае выражения

`x[i] + ++i`

не определено, в каком порядке компилятор будет выполнять оценку, — начиная с левой части оператора «плюс» или начиная с правой части. В данном случае порядок вычислений влияет на результат, поскольку значение *i* может быть увеличено до того, как будет вычисляться *x[i]*.

В этом выражении порядок вычислений также не определен:

*x[i] = ++i*

В этой ситуации не определено, когда значение *i* будет использовано как индекс для *x*, — до приращения *i* или после приращения.

Порядок вычисления аргументов функции или метода тоже не определен. Поэтому при вызове функции

*f (i, ++i);*

или в выражении с сообщением

*[myFract setTo: i over: ++i];*

возможно, что сначала будет выполнено приращение *j*, в результате чего функции или методу будут переданы два одинаковых значения.

Язык Objective-C гарантирует, что операторы *&&* и *||* будут вычисляться слева направо. Поэтому в случае *&&* гарантируется, что второй operand не будет оцениваться, если первый равен 0; в случае *||* гарантируется, что второй operand не будет оцениваться, если первый не равен 0. Этот факт необходимо учитывать при формировании таких выражений, как

```
if (dataFlag || [myData checkData])
...
```

поскольку в этом случае *checkData* вызывается, только если значение *dataFlag* равно 0. Еще один пример. Если в определении объект-массив *a* содержит *n* элементов, то оператор, который начинается с

```
if (index >= 0 && index < n && ([a objectAtIndex: index] == 0))
...
```

является ссылкой на элемент этого массива, только если *index* является допустимым индексом в этом массиве.

## Константные выражения

*Константное выражение* – это выражение, каждый из членов которого является константным значением. Константные выражения необходимы в следующих ситуациях.

1. Как значение после `case` в операторе `switch`.
2. Для указания размера массива.
3. Чтобы присвоить значение идентификатору перечисления.
4. Чтобы указать размер битового поля в определении структуры.
5. Чтобы присвоить начальные значения внешним или статическим переменным.
6. Чтобы задать начальные значения для глобальных переменных.
7. Как выражение после `#if` в препроцессорном операторе `#if`.

В первых четырех случаях константное выражение должно состоять из целых констант, символьных констант, перечисляемых констант и выражений `sizeof`. Могут использоваться только арифметические операторы, побитовые операторы, операторы отношения, оператор с условным выражением и оператор приведения типа (`cast`).

В пятом и шестом случаях помимо вышеперечисленных правил может использоваться явно или неявно адресный оператор. Однако он может применяться только к внешним или статическим переменным или функциям. Например, выражение

`&x + 10`

будет допустимым константным выражением, если `x` является внешней или статической переменной. Кроме того, выражение

`&a[10] - 5`

является допустимым константным выражением, если `a` является внешним или статическим массивом. И, наконец, поскольку `&a[0]` эквивалентно выражению

`a + sizeof(char) * 100`

оно тоже является допустимым константным выражением.

В последней ситуации, где необходимо константное выражение (после `#if`), применяются такие же правила, как для первых четырех случаев, за исключением того, что не могут использоваться оператор `sizeof`, константы перечисления и оператор приведения типа (`cast`). Однако разрешается специальный определенный оператор (см. раздел «Директива `if`»).

## Арифметические операторы

При условии, что

- a, b выражения любого базового типа данных за исключением void;  
i, j выражения любого целого типа данных

выражение

- a изменяет знак a на противоположный;
- +a дает значение a;
- a + b выполняет сложение a и b;
- a - b выполняет вычитание b из a;
- a \* b выполняет умножение a на b;
- a / b выполняет деление a на b;
- i % j дает остаток от деления i на j.

В каждом выражении выполняются обычные арифметические преобразования над операндами (см. ниже раздел «Преобразование базовых типов данных»). Если a – переменная без знака (unsigned), то для вычисления -a сначала выполняется целочисленное расширение, выполняется вычитание из большего значения расширенного типа и к результату добавляется 1.

При делении двух целых значений выполняется усечение результата. Если один из операндов меньше нуля, то направление усечения не определено (то есть  $-3 / 2$  может дать на некоторых машинах  $-1$  и  $-2$  на других); в противном случае усечение всегда выполняется в сторону 0 ( $3/2$  всегда дает 1). Сводку арифметических операций с указателями см. в разделе «Базовые операции с указателями».

## Логические операторы

При условии, что

- a, b выражения любого базового типа данных за исключением void или оба операнда являются указателями;

выражение

- a && b имеет значение 1, если a и b не равны нулю, и значение 0 в противном случае (b оценивается только в том случае, если a отлично от нуля);
- a || b имеет значение 1, если a или b не равно нулю, и значение 0 в противном случае (b оценивается только в том случае, если a отлично от нуля);
- ! a имеет значение 1, если a равно 0, и значение 0 в противном случае.

К *a* и *b* применяются обычные арифметические преобразования (см. ниже раздел «Преобразование базовых типов данных»). Во всех случаях результат имеет тип *int*.

## Операторы отношения

При условии, что

*a, b* выражения любого базового типа данных за исключением *void* или оба операнда являются указателями;

выражение

*a < b* имеет значение 1, если *a* меньше *b*, и значение 0 в противном случае;

*a <= b* имеет значение 1, если *a* меньше или равно *b*, и значение 0 в противном случае;

*a > b* имеет значение 1, если *a* больше *b*, и значение 0 в противном случае;

*a >= b* имеет значение 1, если *a* больше или равно *b*, и значение 0 в противном случае;

*a != b* имеет значение 1, если *a* не равно *b*, и значение 0 в противном случае.

К *a* и *b* применяются обычные арифметические преобразования (см. ниже раздел «Преобразование базовых типов данных»). Первые четыре оператора отношения имеют смысл для указателей, только если они указывают на один и тот же массив либо на компоненты одной структуры или объединения. Во всех случаях результат имеет тип *int*.

## Побитовые операторы

При условии, что

*i, j, n* выражения любого целого типа данных;

выражение

*i & j* выполняет побитовую операцию AND с *i* и *j*;

*i | j* выполняет побитовую операцию OR с *i* и *j*;

*i ^ j* выполняет побитовую операцию XOR с *i* и *j*;

*-i* получает дополнение до нуля для *i*;

*i << n* выполняет сдвиг *i* влево на *n* бит;

*i >> n* выполняет сдвиг *i* вправо на *n* бит.

К этим операндам применяются обычные арифметические преобразования за исключением << и >>, когда для каждого операнда выполняется целочисленное расширение (см. ниже раздел «Преобразование базовых типов данных»). Если количество битов смещения меньше нуля либо больше или равно числа битов в объекте смещения, то результат смещения не определен. На некоторых машинах правое смещение является арифметическим (заполняется бит знака), а на других – логическим (заполнение нулем). Тип результата операции смещения такой же, как у расширяемого левого операнда.

## Операторы наращивания и уменьшения на 1 (операторы инкремента и декремента)

При условии, что

- | модифицируемое lvalue-выражение, тип которого не уточнен как const; выражение
- +|+ увеличивает | на 1, после чего | используется как значение выражения;
- |++ использует | как значение выражения и затем увеличивает | на 1;
- 1 уменьшает | на 1, после чего | используется как значение выражения;
- |– использует | как значение выражения и затем уменьшает | на 1.

Ниже в разделе «Базовые операции с указателями» описываются эти операции с указателями.

## Операторы присваивания

При условии, что

- | модифицируемое lvalue-выражение, тип которого не уточнен как const;
  - op любой оператор, который можно использовать как оператор присваивания (см. таблицу В.4);
  - a выражение;
  - выражение
- | = a сохраняет значение a в |;
- | op= a применяет op к | и a, сохраняя результат в |.

Если в первом выражении *a* имеет один из базовых типов данных (за исключением *void*), то *a* преобразуется для соответствия типу *l*. Если *l* – указатель, то *a* должен быть указателем того же типа, *void*-указателем или *null*-указателем.

Если *l* – *void*-указатель, то *a* может быть указателем любого типа. Второе выражение интерпретируется как *l = l op (a)*, за исключением того, что *l* оценивается только один раз (например, в случае *x[i++] += 10*).

## Условный оператор

При условии, что

*a, b, c* выражения;

выражение

*a ? b : c* получает значение *b*, если *a* отлично от нуля, или *c* в противном случае.

Вычисляется только выражение *b* или *c*.

Выражения *b* и *c* должны иметь одинаковый тип данных. Если это не так, но оба выражения имеют арифметические типы данных, то применяются обычные арифметические преобразования, чтобы сделать их типы одинаковыми. Если одно из них является указателем, а другое равно 0, то второе выражение используется как *null*-указатель того же типа, что и первое. Если одно из них является указателем на *void*, а другое является указателем на другой тип, то второе преобразуется в указатель на *void* и результат имеет такой же тип.

## Оператор приведения типа

При условии, что

*тип* имя какого-либо базового типа данных, перечислимый тип данных (с ключевым словом *enum*) или тип, определенный как *typedef* либо приведенный тип данных;

*a* выражение;

выражение

(*тип*) преобразует *a* в указанный тип.

Отметим, что использование типа в круглых скобках при объявлении или определении метода не является примером использования оператора приведения типа.

## Оператор `sizeof`

При условии, что

*тип* совпадает с описанием в предыдущем разделе;

*a* выражение;

выражение

`sizeof (тип)` имеет значение, равное числу байтов памяти, необходимых для значения указанного типа;

`sizeof a` имеет значение, равное числу байтов памяти, необходимых для результата вычисления *a*.

Для типа `char` результат по определению равен 1. Если *a* – имя массива, для которого задан размер (явным образом или неявно через инициализацию), и это не формальный параметр и не `extern`-массив, для которого не указан размер, то `sizeof a` дает число байтов, необходимых для хранения этих элементов в *a*.

Если *a* – имя класса, то `sizeof (a)` дает размер структуры, необходимой для хранения экземпляра *a*.

Целое значение, которое дает оператор `sizeof`, имеет тип `size_t`, который определен в стандартном header-файле `<stddef.h>`.

Если *a* – массив переменной длины, то данное выражение вычисляется на этапе выполнения; в противном случае оно вычисляется на этапе компиляции и его можно использовать в константных выражениях (см. выше раздел «Константные выражения»).

## Оператор «запятая»

При условии, что

*a, b* выражения;

выражение

*a, b* означает вычисление *a* и затем вычисление *b*. Результирующее выражение имеет тип и значение *b*.

## Базовые операции с массивами

При условии, что

- a      объявляется как массив из n элементов;
- i      выражение любого целого типа данных;
- v      выражение;

выражение

- a[0]      является ссылкой на первый элемент a;
- a[n - 1]      является ссылкой на последний элемент a;
- a[i]      является ссылкой на элемент a с номером i;
- a[i] = v      сохраняет значение v в a[i].

В каждом случае тип результата совпадает с типом элементов, содержащихся в a. Сводку операций с указателями и массивами см. ниже в разделе «Базовые операции с указателями».

## Базовые операции со структурами

---

Примечание. Это также относится к объединениям (union).

---

При условии, что

- x      модифицируемое lvalue-выражение типа struct s;
- y      выражение типа struct s;
- m      имя одно из компонентов структуры s;
- obj      любой объект;
- M      любой метод;
- v      выражение;

выражение

- x      является ссылкой на всю структуру и имеет тип struct s;
- y.m      является ссылкой на компонент m структуры y и имеет тип, объявленный для компонента m;

- 
- x.m = v присваивает v компоненту m выражения x и имеет тип, объявленный для компонента m;
  - x = y присваивает x значение y и имеет тип struct s;
  - f(y) вызывает функцию f, передавая как аргумент содержимое структуры y (внутри f должен быть объявлен формальный параметр типа struct s);
  - [obj M: y] вызывает метод M для объекта obj, передавая как аргумент содержимое структуры y (внутри метода этот параметр должен быть объявлен как тип struct s);
  - return y; возвращает структуру y (возвращаемое значение, объявленное для функции или метода, должно иметь тип struct s).

## Базовые операции с указателями

При условии, что

- x Ivalue-выражение типа t;
  - pt молифицируемое Ivalue-выражение типа «указатель на t»;
  - v выражение;
- выражение
- &x дает указатель на x и имеет тип «указатель на t»;
  - pt = &x задает pt как указатель на x и имеет тип «указатель на t»;
  - pt = 0 присваивает pt null-указатель;
  - pt == 0 проверяет, является ли pt null-указателем;
  - \*pt является ссылкой на значение, указываемое pt и имеет тип t;
  - \*pt = v сохраняет значение v в том месте, на которое указывает pt, и имеет тип t.

**Указатели на массивы**

При условии, что

- а массив элементов типа *t*;
- ра1 модифицируемое lvalue-выражение типа «указатель на *b*», которое указывает на какой-либо элемент в а;
- ра2 lvalue-выражение типа «указатель на *b*», которое указывает на какой-либо элемент в а или на элемент, следующий за последним элементом в а;
- v выражение;
- n целое выражение;

выражение

- a, &a, &a[0] каждое дает указатель на первый элемент;
- &a[n] создает указатель на элемент с номером n массива а и имеет тип «указатель на *b*»;
- \*pa1 является ссылкой на элемент, на который указывает ра1, и имеет тип «*b*»;
- \*pa1 = v сохраняет значение v в элементе, на который указывает ра1, и имеет тип «*b*»;
- ++pa1 задает указатель ра1 на следующий элемент а независимо от типа элементов, содержащихся в а, и имеет тип «указатель на *b*»;
- pa1 задает указатель ра1 на предыдущий элемент а независимо от типа элементов, содержащихся в а, и имеет тип «указатель на *b*»;
- \*++pa1 увеличивает ра1 на 1 и затем ссылается на значение в а, на которое указывает ра1; имеет тип *t*;
- \*pa1++ ссылается на значение в а, на которое указывает ра1, прежде чем увеличить ра1 на 1; имеет тип *t*;
- pa1 + n создает указатель, который указывает в а на n элементов дальше, чем ра1, и имеет тип «указатель на *b*»;
- pa1 - n создает указатель, который указывает в а на n элементов ближе, чем ра1, и имеет тип «указатель на *b*»;
- \*(pa1 + n) = v сохраняет значение v в элементе, на который указывает ра1 + n, и имеет тип «указатель на *b*»;
- ra1 < ra2 проверяет, что ра1 указывает в а более близний элемент, чем ра2; имеет тип int (для сравнения двух указателей можно использовать любые операторы отношения);
- ra2 - ra1 дает число элементов, содержащихся в а между указателями ра2 и ра1 (в предположении, что ра2 указывает на элемент, который находится дальше, чем ра1), и имеет значение целого типа;
- a + n создает указатель на элемент массива а с номером n и имеет тип «указатель на *b*»; эквивалентно во всех отношениях выражению &a[n];

**\*(a + n)** ссылка на элемент массива a с номером n, имеет тип «t» и эквивалентно во всех отношениях выражению a[n].

Целое значение, получаемое в результате вычитания двух указателей, имеет конкретный тип ptrdiff\_t, который определен в стандартном header-файле <stddef.h>.

### Указатели на структуры

При условии, что

x модифицируемое lvalue-выражение типа struct s;

ps модифицируемое lvalue-выражение типа «указатель на struct s»;

m имя какого-либо компонента структуры s, имеющего тип t;

v выражение;

выражение

&x дает указатель на x и имеет тип «указатель на struct s»;

ps = &x задает ps как указатель на x, имеющий тип «указатель на struct s»;

ps->m является ссылкой на компонент m структуры, указанной с помощью ps, и имеет тип t;

(\*ps).m тоже является ссылкой на этот компонент и эквивалентно во всех отношениях выражению ps->m;

ps->m = v сохраняет значение v в компоненте m структуры, указанной с помощью ps, и имеет тип t.

## Составные литералы

*Составной литерал* (*compound literal*) представляет собой имя типа, заключенное в круглые скобки, после которого следует список инициализации. В результате создается неименованное значение указанного типа, область действия которого ограничена блоком, в котором оно создано, или глобальной областью действия, если оно определено вне блока. В последнем случае все инициализаторы должны включать только константные выражения.

Например,

`(struct point) {.x = 0, .y = 0}`

является выражением, которое создает структуру типа struct point с указанными начальными значениями. Его можно присвоить другой структуре типа struct point, например,

`origin = (struct point) {.x = 0, .y = 0};`

Или его можно передать функции или методу, если предполагается, что аргумент имеет тип struct point, например,

`moveToPoint ((struct point) {.x = 0, .y = 0});`

Можно также определять типы, отличные от структур. Например, если intPtr имеет тип int \*, то оператор

```
intPtr = (int [100]) {[0] = 1, [50] = 50, [99] = 99 };
```

(который может находиться в любом месте программы) задаст `intptr`, указывающий на массив, содержащий 100 целых элементов, причем первые 3 элемента инициализируются указанным образом.

Если размер массива не задан, он определяется списком инициализации.

## Преобразование базовых типов данных

Язык Objective-C преобразует операнды арифметических выражений в заранее определенном порядке, который называется *обычными арифметическими преобразованиями*.

1. Если один из operandов имеет тип `long double`, второй operand преобразуется в `long double`, таким же будет тип результата.
2. Если один из operandов имеет тип `double`, второй operand преобразуется в `double`, таким же будет тип результата.
3. Если один из operandов имеет тип `float`, второй operand преобразуется в тип `float`, таким же будет тип результата.
4. Если один из operandов имеет тип `_Bool`, `char`, `short int`, является битовым полем типа `int` или является перечислимым типом данных, то он преобразуется в тип `int`, если `int` может полностью представлять его диапазон значений; в противном случае он преобразуется в `unsigned int`. Если оба operandы имеют одинаковый тип, таким же будет тип результата.
5. Если оба operandы указаны как `signed` или оба как `unsigned`, то целый тип меньшего размера преобразуется в больший целый тип, и таким же будет тип результата.
6. Если operand с атрибутом `unsigned` имеет размер, который не меньше размера operand с атрибутом `signed`, то operand `signed` преобразуется в тип operand `unsigned`, таким же будет тип результата.
7. Если operand с атрибутом `signed` может представлять все значения operand `unsigned`, то второй преобразуется в тип первого, если он может полностью представлять весь диапазон его значений, таким же будет тип результата.
8. Если был достигнут этот шаг, то оба operandы преобразуются в тип с атрибутом `unsigned`, соответствующий типу с атрибутом `signed`.

Шаг 4 называется более формально *целочисленным расширением*.

Преобразование operandов нормально проходит в большинстве ситуаций, хотя следует отметить следующие особенности.

1. Преобразование `char` в `int` может вызвать на некоторых машинах расширение на знаковый бит, если `char` не объявлен как `unsigned`.
2. Преобразование целого типа с атрибутом `signed` в целый тип большего размера вызывает расширение влево за счет знакового бита; преобразование целого типа с атрибутом `unsigned` в целый тип большего размера вызывает заполнение левого бита нулем.

3. Преобразование любого значения в `_Bool` дает значение 0, если значение равно нулю, и 1 в противном случае.
4. Преобразование более длинного целого типа в более короткий вызывает усечение целого значения слева.
5. Преобразование типа с плавающей точкой в целый тип вызывает усечение дробной части значения. Если целый тип является недостаточным, чтобы вместить преобразованное значение с плавающей точкой, то результат является неопределенным (как и в случае преобразования отрицательного значения с плавающей точкой в целое с атрибутом `unsigned`).
6. Преобразование более длинного значения с плавающей точкой в более короткий тип может вызывать или не вызывать округление, прежде произойдет усечение.

## Классы памяти и область действия

Термин *класс памяти* (*storage class*) относится к способу выделения памяти компилятором и к области действия определения конкретной функции или метода. Классы памяти – это `auto`, `static`, `extern` и `register`. Класс памяти можно не указывать в объявлении, и он будет назначен по умолчанию, как это описывается ниже.

Термин *область действия* (*scope*) относится к границам действия определенного идентификатора внутри программы. Идентификатор, определенный вне любой функции, метода или блока операторов (мы будем называть здесь это **БЛОКОМ**), доступен для ссылки в любой последующей точке файла. Идентификаторы, определенные внутри БЛОКА, являются локальными относительно этого БЛОКА и могут локально переопределять идентификатор, определенный вне этого БЛОКА. Имена меток, а также имена формальных параметров известны во всем БЛОКЕ. Метки, переменные экземпляра, имена структур и компонентов структур, имена объединений и компонентов объединений, а также имена перечислимых типов данных не обязательно должны отличаться друг от друга или от имен переменных, функций или методов. Однако идентификаторы перечисления все же должны отличаться от имен переменных и от других идентификаторов перечисления, определенных в пределах одной области действия. Имена классов имеют глобальную область действия и должны отличаться от имен других переменных и типов в пределах одной области действия.

## Функции

Если при определении функции определяется класс памяти, он должен быть указан как `static` или `extern`. К функции, которая объявляется как `static`, можно обращаться только в пределах файла, который содержит эту функцию. Функции, объявленные как `extern` (или функции, для которых не указан никакой класс), могут вызываться функциями или методами из других файлов.

## Переменные

В таблице В.5 приводится сводка различных классов памяти, которые могут использоваться в объявлении переменных, а также их области действия и способы их инициализации.

**Табл. В.5. Переменные: сводка классов памяти, областей действия и способов инициализации**

| Класс памяти | Место объявления переменной | На нее можно ссылаться | И инициализировать с помощью  | Комментарии                                                                                                                                                                                                                                                                                  |
|--------------|-----------------------------|------------------------|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static       | Вне любого БЛОКА            | Внутри данного файла   | Только константного выражения | Переменные инициализируются только один раз при запуске программы; значения сохраняются в пределах БЛОКОВ; значение по умолчанию 0.                                                                                                                                                          |
|              | Внутри какого-либо БЛОКА    | Внутри этого БЛОКА     |                               |                                                                                                                                                                                                                                                                                              |
| extern       | Вне любого БЛОКА            | Внутри данного файла   | Только константных выражений  | Переменная должна быть объявлена хотя бы в одном месте без ключевого слова <code>extern</code> или в одном месте с использованием ключевого слова <code>extern</code> с присваиванием начального значения.                                                                                   |
|              | Внутри какого-либо БЛОКА    | Внутри этого блока     |                               |                                                                                                                                                                                                                                                                                              |
| auto         | Внутри какого-либо БЛОКА    | Внутри этого блока     | Любого допустимого выражения  | Переменная инициализируется каждый раз при входе в этот БЛОК; нет никакого значения по умолчанию.                                                                                                                                                                                            |
| register     | Внутри какого-либо БЛОКА    | Внутри этого блока     | Любого допустимого выражения  | Присваивание регистра ( <code>register</code> ) не гарантируется; разнообразные ограничения на типы переменных, которые могут быть объявлены; нельзя брать адрес переменной с <code>register</code> ; инициализируется каждый раз при входе в этот БЛОК; нет никакого значения по умолчанию. |

**Табл. В.5.** Переменные: сводка классов памяти, областей действия и способов инициализации (окончание)

| Класс памяти | Место объявления переменной | На нее можно ссылаться                                                                 | Инициализировать с помощью    | Комментарии                                                                                                                                                                    |
|--------------|-----------------------------|----------------------------------------------------------------------------------------|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Не указан    | Вне любого БЛОКА            | Внутри данного файла или из других файлов, которые содержат соответствующие объявления | Только константного выражения | Это объявление может появляться только в одном месте; переменная инициализируется при запуске программы; значение по умолчанию 0; по умолчанию используется класс памяти auto. |
|              | Внутри какого-либо БЛОКА    | (См. auto)                                                                             | (См. auto)                    |                                                                                                                                                                                |

## Переменные экземпляра

К переменным экземпляра можно получать доступ с помощью любого метода экземпляра, определенного для данного класса в секции `interface`, где явно объявляется эта переменная, или в категориях, созданных для данного класса. К наследуемым переменным экземпляра тоже можно осуществлять непосредственный доступ без каких-либо специальных объявлений. Методы класса не имеют доступа к переменным экземпляра.

Для управления областью действия переменной экземпляра можно использовать специальные директивы `@private`, `@protected` и `@public`. После появления этих директив они действуют, пока не встретится закрывающая фигурная скобка, заканчивающая объявление соответствующих переменных экземпляра, или пока не будет использована другая из этих трех директив. Например, следующие строки являются началом объявления секции `interface` для класса с именем `Point`, содержащего четыре переменные экземпляра.

```

@interface Point: NSObject
{
 @private
 int internalID;
 @protected
 float x;
 float y;
 @public
 BOOL valid;
}

```

Переменная `internalID` объявлена как `private`, переменные `x` и `y` – как `protected` (директива по умолчанию), переменная `valid` – как `public`.

Эти директивы описываются в таблице В.6.

**Табл. В.6.** Область действия переменных экземпляра

| Директива               | На переменную можно ссылаться                                                                                                                                                                                                                                                                                                                                                                                             | Комментарии                                                                                            |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>@protected</code> | С помощью методов экземпляра в данном классе, методов экземпляра в подклассах и методов экземпляра в расширениях категорий на данный класс.                                                                                                                                                                                                                                                                               | Это директива по умолчанию.                                                                            |
| <code>@private</code>   | С помощью методов экземпляра в данном классе, методов экземпляра в расширениях категорий на данный класс, но не из каких-либо подклассов.                                                                                                                                                                                                                                                                                 | Эта директива ограничивает доступ самим классом.                                                       |
| <code>@public</code>    | С помощью методов экземпляра в данном классе, методов экземпляра в подклассах и методов экземпляра в расширениях категорий на данный класс; доступ к такой переменной можно также выполнять из других функций или методов, применяя к экземпляру класса оператор косвенного указателя структуры ( <code>-&gt;</code> ), после которого следует имя переменной экземпляра (например, <code>myFract-&gt;numerator</code> ). | Эту директиву следует использовать только при необходимости; она нарушает понятие инкапсуляции данных. |

## ФУНКЦИИ

В этом разделе содержатся сведения о синтаксисе и работе функций.

### Определение функции

#### Общий формат

```
возвращаемыйТип имя (тип1 параметр1, тип2 параметр2, ..)
{
 ОбъявленияПеременных
 программныйОператор
 программныйОператор
 ...
 return выражение;
}
```

Определяется функция *имя*, которая возвращает значение типа *возвращаемыйТип* и имеет формальные параметры *параметр1*, *параметр2*, ...; *параметр1* объявляется с типом *тип1*, *параметр2* с типом *тип2*, и т.д.

Локальные переменные обычно объявляются в начале функции, но это не обязательно. Они могут быть объявлены в любом месте, и тогда доступ к ним ограничен операторами, которые следует в функции после их объявления.

Если функция не возвращает значения, то *возвращаемыйТип* указывается как *void*.

Если в круглых скобках содержится только *void*, функция не принимает никаких аргументов. Если применяются .. в качестве последнего (или единственного) параметра в списке, функция принимает переменное число аргументов, например,

```
int printf (char *format, ...)
{
 ...
}
```

В объявлениях для аргументов одномерных массивов не обязательно указывать число элементов этого массива. Для многомерных массивов нужно обязательно указать размер каждой размерности, кроме первой.

Описание оператора *return* см. ниже в разделе «Оператор *return*».

По-прежнему поддерживается старый способ определения функций. Для этого используется следующий общий формат.

```
возвращаемыйТип имя (параметр1, параметр2, ..)
объявления_параметров
{
 ОбъявленияПеременных
 программныйОператор
 программныйОператор
```

```

...
 return выражение;
}

```

Здесь в круглых скобках указываются только имена параметров. Если функция не принимает никаких аргументов, то указываются только левая и правая круглые скобки. Тип каждого параметра объявляется вне круглых скобок и перед левой фигурной скобкой, открывающей определение функции. Например, ниже определяется функция с именем `rotate`, которая принимает два параметра с именами `value` и `n`.

```

unsigned int rotate (value, n)
unsigned int value;
int n;
{
...
}

```

Первый аргумент определяется как `unsigned int` и второй как `int`.

Перед определением функции можно поместить ключевое слово `inline` как указание компилятору. Некоторые компиляторы заменяют вызов такой функции конкретным кодом самой функции, что ускоряет выполнение. Ниже показан соответствующий пример.

```

inline int min (int a, int b)
{
 return (a < b ? a : b);
}

```

## Вызов функции

### Общий формат

`имя ( arg1, arg2, .. )`

Происходит вызов функции `имя`, и значения `arg1, arg2, ..` передаются функции как аргументы. Если функция не принимает никаких аргументов, указываются только открывающая и закрывающая круглые скобки (например, `initialize ()`).

Если вызывается функция, которая определена после самого вызова или находится в другом файле, то необходимо включить объявление прототипа для этой функции, которое имеет следующий общий формат.

`возвращаемыйТип имя (тип1 парам1, тип2 парам2, .. )`

Это указывает компилятору тип возвращаемого значения функции, число аргументов, которое она принимает, и тип каждого аргумента. Например, в строке

`long double power (double x, int n);`

содержится объявление `power` как функции, которая возвращает значение типа `long double` и принимает два аргумента типа `double` и типа `int`. Имена аргументов

внутри круглых скобок на самом деле являются фиктивными именами, и они могут быть опущены, то есть

```
long double power (double, int);
```

означает то же самое.

Если компилятор уже встретил определение функции или объявление прототипа для этой функции, то при вызове функции тип каждого аргумента преобразуется (там, где возможно) в соответствии с типом, который объявлен в функции.

Если компилятор не встретил ни определения функции, ни объявления прототипа, то он предполагает, что функция возвращает значение типа `int`, автоматически преобразует все аргументы типа `float` в тип `double` и выполняет целочисленное расширение любых целых аргументов, как описано выше в разделе «Преобразование базовых типов данных». Другие аргументы функции передаются без преобразования.

Функции, которые принимают переменное число аргументов, должны быть объявлены соответствующим образом. В противном случае компилятор будет предполагать, что функция принимает фиксированное число аргументов, основываясь на числе аргументов, фактически использованных при вызове функции.

Если функция была определена в старом формате (см. выше раздел «Определение функции»), то объявление для такой функции имеет следующий формат.

```
возвращаемыйТип имя ();
```

Аргументы, передаваемые такой функции, преобразуются в соответствии с описанием в предыдущем абзаце.

Для функций, возвращаемый тип которых объявлен как `void`, компилятор выводит сообщение для любых вызовов, где делается попытка использования возвращаемого значения.

Все аргументы передаются функции по их значению, поэтому их значения не могут быть изменены функцией. Но если функции передается указатель, то функция может изменить значения, которые указываются этим указателем, но она все же не может изменить значение самой переменной-указателя.

## Указатели на функции

Имя функции без последующей пары круглых скобок создает указатель на эту функцию. К имени функции можно также применить адресный оператор, чтобы создать указатель на эту функцию.

Если `fp` – указатель на функцию, то соответствующую функцию можно вызвать, написав

```
fp ()
```

или

```
(*fp) ()
```

Если функция принимает аргументы, они должны быть указаны в круглых скобках.

## Классы

В этом разделе приводятся сведения о синтаксисе и семантике классов.

### Определение класса

В определение класса включается объявление переменных экземпляра и методов в секции `interface`, а также определение кода для каждого метода в секции `implementation`.

#### Секция `interface`

##### Общий формат

```
@interface имяКласса: родительскийКласс <протокол, ...>
{
 объявленияПеременныхЭкземпляра
}

объявлениеМетода
объявлениеМетода
...
@end
```

Класс `имяКласса` объявляется с помощью родительского класса `родительскийКласс`. Если `имяКласса` принимает также один или несколько формальных протоколов, то имена протоколов перечисляются в угловых скобках после родительского класса. В этом случае соответствующая секция `implementation` должна содержать определения всех методов из перечисленных протоколов.

Если двоеточие и `родительскийКласс` не указаны, это означает, что объявляется новый корневой класс.

##### Объявления переменных экземпляра

В необязательной секции `объявленияПеременныхЭкземпляра` указываются тип и имя каждой переменной экземпляра для данного класса. Каждый экземпляр класса `имяКласса` получает свой собственный набор этих переменных плюс переменные, наследуемые из класса `родительскийКласс`. Доступ к таким переменным можно осуществлять непосредственно по имени в методах экземпляра, определенных в классе `имяКласса`, или из подклассов класса `имяКласса`. Если доступ ограничен директивой `@private`, то подклассы не могут осуществлять доступ к таким переменным (см. выше раздел «Переменные экземпляра»).

Методы класса не имеют доступа к переменным экземпляра.

##### Объявления свойств

##### Общий формат

```
@property (атрибуты) списокИмен;
```

Объявляются свойства с помощью заданного списка атрибутов с разделителями-запятыми.

*списокИмен* — это список с разделителями-запятыми, содержащий имена свойств объявляемого типа.

(тип) *имяСвойства1, имяСвойства2, имяСвойства3,...*

Директива @property может присутствовать в любом месте секции объявления метода для класса, протокола или категории.

**Табл. B.7. Атрибуты свойств**

| Атрибут            | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| assign             | Используется простое присваивание, чтобы задать значение переменной экземпляра в методе-установщике (setter). (Это атрибут по умолчанию.)                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| copy               | Используется метод copy, чтобы задать значение переменной экземпляра.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| getter= <i>имя</i> | Используется <i>имя</i> для имени метода-получателя (getter) вместо имени <i>имяСвойства</i> , которое используется по умолчанию для синтезируемого метода-получателя.                                                                                                                                                                                                                                                                                                                                                                                                                          |
| nonatomic          | Значение может быть возвращено непосредственно из синтезируемого метода-получателя. Если этот атрибут не объявлен, то методы доступа (accessor) действуют с атрибутом atomic; это означает, что доступ к переменным экземпляра защищен блокировкой mutex. Это обеспечивает защиту в среде с несколькими потоками за счет того, что операция get или set выполняется только в одном потоке. Кроме того, по умолчанию в среде без сборки мусора синтезируемый метод-получатель будет удерживать (retain) и автоматически высвобождать (autorelease) свойство, прежде чем возвратить его значение. |
| readonly           | Значение свойства нельзя задать. Никакой метод-установщик не предполагается компилятором и не может быть синтезирован. (Это атрибут по умолчанию.)                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| “fwrite”           | Значение свойства можно читать и задавать. Компилятор предполагает, что вы сами предоставили метод-получатель и метод-установщик (getter и setter), или он будет синтезировать оба метода, если используется директива @synthesize.                                                                                                                                                                                                                                                                                                                                                             |
| retain             | Свойство должно удерживаться (retain) во время присваивания. Этот атрибут можно указывать только для типов Objective-C.                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| setter= <i>имя</i> | Используется <i>имя</i> для имени метода-установщика (setter) вместо заданного имени <i>имяСвойства</i> , которое используется по умолчанию для синтезируемого метода доступа.                                                                                                                                                                                                                                                                                                                                                                                                                  |

Можно указывать только один из атрибутов assign, copy или retain. Если у вас не используется сборка мусора, то один из этих атрибутов должен быть использован явным образом; в противном случае вы получите предупреждение от компилятора. Если у вас используется сборка мусора и вы не указали один из этих трех атрибутов, то будет применяться атрибут по умолчанию assign. В этом слу-

чае компилятор выдаст предупреждение, только если данный класс подчиняется протоколу `NSCopying` (в этом случае для свойства может потребоваться атрибут `copy`, а не `assign`).

Если вы используете атрибут `copy`, то синтезируемый метод-установщиком применяет метод `copy` данного объекта. Это дает немутабельную копию. Если вам требуется мутабельная копия, то вы можете предоставить вместо этого свой собственный метод-установщик.

### Объявление метода

#### Общий формат

`мТип (возвращаемыйТип) имя??1 : (тип1) параметр1 имя??2: (тип2) параметр2, ...;`

Объявляется метод `имя1:имя2:...`, который возвращает значение типа `возвращаемыйТип`; он имеет формальные параметры `параметр1, параметр2, ...; параметр1` объявляется с типом `тип1, параметр2` с типом `тип2`, и т.д.

Любое из имен после `имя1` (то есть `имя2, ...`) может быть опущено; в этом случае все же используется двоеточие в качестве заполнителя, и оно становится частью имени метода (см. следующий пример).

Если в качестве `мТип` указан знак `+`, это означает, что объявляется метод класса, а если знак `-`, то объявляется метод экземпляра.

Если объявляемый метод наследуется из родительского класса, то новое определение подавляет определение родительского класса. В этом случае все же имеется доступ к методу родительского класса, и для этого нужно передать сообщение к `super`. Вызов методов класса происходит, когда соответствующее сообщение передается объекту-классу, а вызов методов экземпляра происходит, когда соответствующее сообщение передается экземпляру класса. Методы класса и методы экземпляра могут иметь одинаковые имена.

Однаковое имя метода может также использоваться в различных классах. Способность объектов различных классов реагировать на методы с одинаковыми именами называется *полиморфизмом* (*polymorphism*).

Если метод не возвращает значения, то `возвращаемыйТип` указывается как `void`. Если метод возвращает значение типа `id`, то `возвращаемыйТип` можно не указывать, хотя практика надежного программирования рекомендует указывать `id` как возвращаемый тип.

Если используется `, ...` в качестве последнего (или единственного) параметра в списке, то метод принимает переменное число параметров, например,

```
- (void) print: (NSSTRING *) format, ...
{
 ...
}
```

Ниже приводится пример объявления класса в секции `interface`: объявляется класс с именем `Fraction`, родительским классом для которого является `NSObject`.

```
@interface Fraction: NSObject
{
 int numerator, denominator;
}
```

```

+(Fraction *) newFract;
-(void) setTo: (int) n : (int) d;
-(void) setNumerator: (int) n andDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end

```

Класс Fraction имеет две целые переменные экземпляра с именами numerator и denominator. Он также содержит один метод класса с именем newFract, который возвращает объект типа Fraction. В него включены также два метода экземпляра с именами setTo:: и setNumerator:andDenominator:. Каждый из них принимает два аргумента и не возвращает никакого значения. В нем содержатся также два метода экземпляра с именами numerator и denominator, которые не принимают никаких аргументов и возвращают значение типа int.

## Секция implementation

### Общий формат

```

@implementation имяКласса;
 определениеМетода
 определениеМетода
 ...
@end

```

Определяется класс с именем *имяКласса*. Родительский класс и переменные экземпляра обычно не объявляются повторно в секции implementation (хотя это можно делать), поскольку они уже объявлены ранее в секции interface.

Если реализуются методы не из какой-либо категории (см. ниже раздел «Определение категории»), то в секции implementation должны быть определены все методы, объявленные в секции interface. Если в секции interface были указаны один или несколько протоколов, то должны быть определены все методы этих протоколов – неявным образом через наследование или явным образом путем определения в секции implementation.

Каждое *определениеМетода* содержит код, который будет выполняться при вызове этого метода.

### Определение метода

#### Общий формат

```

мТип (возвращаемыйТип) имя1: (тип1) парам1: имя2(тип2) парам2, ...
{
 объявленияПеременных

 программныйОператор
 программныйОператор
 ...
 return выражение;
}

```

Определяется метод *имя1:имя2...*, который возвращает значение типа *возвращаемыйТип* и имеет формальные параметры *параметр1, параметр2, ..., параметрn* объявляется с типом *тип1, параметр2* с типом *тип2*, и т.д. Если в качестве *мТип* указан знак +, это означает, что определяется метод класса, а если знак -, то определяется метод экземпляра. Это объявление метода должно быть согласовано с соответствующим объявлением метода в секции *interface* или с определенным ранее определением протокола.

В методе экземпляра можно обращаться к переменным экземпляра данного класса и любым переменным, которые унаследованы этим методом непосредственно по имени. Если определяется метод класса, он не может обращаться к каким-либо переменным экземпляра.

Внутри метода можно использовать идентификатор *self* для ссылки на объект, для которого вызывается метод, то есть на получателя сообщения. Внутри метода можно использовать идентификатор *super* для ссылки на родительский класс объекта, для которого вызывается метод.

Если *возвращаемыйТип* не указан как *void*, то в определении метода должны содержаться один или несколько операторов *return* с выражениями типа *возвращаемыйТип*. Если *возвращаемыйТип* указан как *void*, использовать оператор *return* не обязательно, и в случае использования он не может содержать возвращаемого значения.

Ниже приводится пример определения метода, где определяется метод *setNumerator:andDenominator:* в соответствии с его объявлением (см. выше раздел «Объявление метода»).

```
- (void) setNumerator: (int) n andDenominator: (int) d
{
 numerator = n;
 denominator = d;
}
```

В этом методе двум его переменным экземпляра присваиваются передаваемые аргументы и не выполняется оператор *return* (хотя это можно сделать), поскольку в объявлении метода он не возвращает никакого значения.

В объявлениях для аргументов одномерных массивов не обязательно указывать число элементов этого массива. Для многомерных массивов нужно обязательно указать размер каждой размерности, кроме первой.

Внутри метода можно объявлять локальные переменные, и они обычно объявляются в начале определения метода. Память для автоматических локальных переменных выделяется при вызове метода, и она освобождается при выходе из метода.

Описание оператора *return* см. ниже в разделе «Оператор *return*».

## **Синтезируемые методы доступа**

### **Общий формат**

*@synthesize свойство1, свойство2, ...*

Здесь указывается, что для перечисленных свойств *свойство1, свойство2, ...* должны быть синтезированы методы доступа.

В списке можно использовать форму записи *свойство* = *instance\_var*, чтобы указать, что свойство будет связано с переменной экземпляра *instance\_var*. Синтезируемые методы будут иметь характеристики, базирующиеся на атрибутах, объявленных ранее для свойства с помощью директивы @property.

## Определение категории

### Общий формат

```
@interface имяКласса (имяКатегории) <протокол,...>
 объявлениеМетода
 объявлениеМетода
 ...
@end
```

Здесь объявляется категория *имяКатегории* для класса *имяКласса* с перечисленными методами. Если указаны один или несколько протоколов, категория подчиняется перечисленным протоколам.

Компилятору должно быть известно *имяКласса* из предшествующего объявления в секции @interface для этого класса. Можно объявить любое число категорий в любом числе различных исходных файлов. Перечисленные методы становятся частью данного класса и наследуются подклассами.

Категории уникально определяются парами *имяКласса/имяКатегории*. Например, в определенной программе может быть только одна категория NSArray (Private). Но можно использовать одинаковые имена категорий. Например, определенная программа может содержать категорию NSArray (Private) и категорию NSString (Private), и это будут различные категории.

Вы не обязаны реализовать методы категории, которые не собираетесь использовать.

Категория может только расширять определение класса дополнительными методами или переопределять существующие методы этого класса. Она не может определять какие-либо новые переменные экземпляра для этого класса.

Если в нескольких категориях объявлен метод с одним и тем же именем для одного класса, то нельзя определить, какой метод будет выполняться при вызове.

Например, в следующем примере для класса Complex определяется категория с именем ComplexOps, содержащая четыре метода экземпляра.

```
#import «Complex.h»
@interface Complex (ComplexOps)
-(Complex *) abs;
-(Complex *) exp;
-(Complex *) log;
-(Complex *) sqrt;
@end
```

Можно предположить, что где-либо будет присутствовать секция implementation, в которой реализуются один ли несколько из этих методов.

```
#import "ComplexOps.h"
@implementation Complex (ComplexOps)
-(Complex *) abs
{
 ...
}
-(Complex *) exp
{
 ...
}
-(Complex *) log
{
 ...
}
-(Complex *) sqrt
{
 ...
}
@end
```

Категория, содержащая методы, которые будут реализоваться другими подклассами, называется *неформальным (informal) протоколом* или *абстрактной (abstract) категорией*. В отличие от формальных протоколов, компилятор не выполняет никаких проверок на подчинение неформальному протоколу. На этапе выполнения объект может проверяться на подчиненность неформальному протоколу в зависимости от конкретного метода.

## Определение протокола

### Общий формат

```
@protocol имяПротокола <протокол, ...>
 объявленияМетодов
@optional
 объявленияМетодов
@required
 объявленияМетодов
 ...
@end
```

Здесь определяется протокол *имяПротокола* с указанными методами. Если включены другие протоколы, то протокол *имяПротокола* принимает (*adopt*) перечисленные протоколы.

Это определение называют также определением формального протокола. Класс подчиняется (*conform*) протоколу *имяПротокола*, если в нем определяются или наследуются все обязательные (*required*) методы, объявленные в этом протоколе, а также все методы любых других перечисленных протоколов. Компилятор проверяет подчиненность и выводит предупреждение, если класс не подчиняется объявленному формальному протоколу. На этапе выполнения

объекты могут проверяться или не проверяться на подчиненность формальному протоколу.

Перед списком методов, реализация которых не обязательна, может ставиться директива `@optional`. В дальнейшем можно использовать директиву `@required`, чтобы обновить список обязательных методов, которые должны быть реализованы для подчинения протоколу.

Протоколы часто не привязываются к какому-либо определенному классу, но применяются как способ определения общего интерфейса, используемого несколькими классами.

### Специальные модификаторы типов

Для типов параметров и возвращаемого значения методов, объявляемых в протоколах, могут использоваться спецификаторы типов, которые приводятся в таблице В.8. Эти спецификаторы используются для распределенных объектных приложений.

**Табл. В.8. Специальные модификаторы типов в протоколах**

| Спецификатор        | Описание                                                                                                                                                                                                                                                     |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>in</code>     | Аргумент относится к объекту, значение которого будет изменено отправителем и передано (т.е. скопировано) назад получателю.                                                                                                                                  |
| <code>out</code>    | Аргумент относится к объекту, значение которого будет изменено получателем и передано назад отправителю.                                                                                                                                                     |
| <code>inout</code>  | Аргумент относится к объекту, значение которого будет изменяться как отправителем, так и получателем, и будет передаваться между ними; это спецификатор по умолчанию.                                                                                        |
| <code>oneway</code> | Используется для объявлений типа возвращаемого значения; обычно используется ( <code>one way void</code> ), чтобы указать, что объект, вызвавший этот метод, не обязан ожидать получения возвращаемого значения, то есть метод может выполняться асинхронно. |
| <code>bycopy</code> | Аргумент или возвращаемое значение должно копироваться.                                                                                                                                                                                                      |
| <code>byref</code>  | Аргумент или возвращаемое значение передается по ссылке и не копируется.                                                                                                                                                                                     |

## Объявление объекта

### Общий формат

```
имяКласса *var1, *var2, ...;
```

Здесь определяются переменные *var1*, *var2*, ... как объекты из класса *имяКласса*. Отметим, что это объявление переменных-указателей, при котором не выделяется пространство для конкретных данных, содержащихся в каждом объекте. В объявлении

```
Fraction *myFract;
```

определяется *myFract* как объект класса *Fraction* или, точнее, как указатель на этот объект. Чтобы фактически выделить пространство для структуры данных класса *Fraction*, нужно вызвать обычно *alloc* или новый метод для этого класса, например,

```
myFract = [Fraction alloc];
```

В результате для объекта типа *Fraction* будет выделен достаточный объем памяти и возвращен указатель на этот объект, который будет присвоен переменной *myFract*. Переменную *myFract* часто называют объектом или экземпляром класса *Fraction*. Если метод *alloc* определен в корневом объекте, всем переменным экземпляра нового выделяемого объекта присваивается значение 0. Но это не означает, что объект инициализирован должным образом; прежде чем использовать этот объект, для него должен быть вызван какой-либо метод инициализации (например, *init*).

Поскольку переменная *myFract* была объявлена явным образом как объект из класса *Fraction*, она называется переменной со *статическим контролем типа*. Компилятор может проверять применение переменных со статическим контролем типа на согласованность с определением класса с точки зрения правильности использования методов, а также типов их аргументов и возвращаемых значений.

### Объявление объекта с типом id

### Общий формат

```
id <протокол,...> var1, var2, ...;
```

Здесь объявляются переменные *var1*, *var2*, ... как объекты из неопределенного класса, который подчиняется протоколам, перечисленным в угловых скобках. Список протоколов не является обязательным.

Переменным типа *id* можно присваивать объекты из любого класса, и наоборот. Если указаны один или несколько протоколов, компилятор проверяет, что методы, используемые из перечисленных протоколов для любой из объявленных переменных, используются соответствующим образом, то есть в соответствии с типами аргументов и возвращаемых значений для методов, объявленных в формальном протоколе.

Например, в строках

```
id <MathOps> number;
...
result = [number add: number2];
```

компилятор проверяет, определяется ли метод `add:` в протоколе `MathOps`. Если да, то компилятор проверяет для этого метода согласованность с типами аргументов и возвращаемого значения. Например, если метод `add:` принимает целый аргумент, а вы передаете ему объект класса `Fraction`, то компилятор выводит соответствующее сообщение.

Система следит за классом, которому принадлежит каждый объект; поэтому на этапе выполнения она может определить класс объекта и затем выбрать для вызова подходящий метод. Эти две процедуры называются соответственно *динамическим контролем типов* (*dynamic typing*) и *динамическим связыванием* (*dynamic binding*).

## Выражения с сообщениями

### Формат 1

[*получатель имя1: arg1 имя2: arg2, имя3: arg3 ..* ]

Выполняется вызов метода *имя1:имя2:имя3* из класса, указанного *получателем*, и значения *arg1*, *arg2*, ... передаются как аргументы. Это называется *выражением с сообщением* (*message expression*). Значением выражения является значение, возвращаемое методом, или `void`, если метод объявлен соответствующим образом и не возвращает никакого значения. Тип этого выражения совпадает с типом, объявленным для вызываемого метода.

### Формат 2

[*получатель имя*];

Если метод не принимает никаких аргументов, то этот формат используется для вызова метода *имя* из класса, указанного *получателем*.

Если получатель имеет тип `id`, то компилятор ищет среди объявленных классов определение или наследуемое определение указанного метода. Если не найдено такого определения, компилятор выводит предупреждение, что получатель не ответит на указанное сообщение. Кроме того, предполагается, что метод возвращает значение типа `id` и преобразует любые аргументы типа `float` в тип `double`, а также выполняет целочисленное расширение для любых целых аргументов, как описано выше в разделе «Преобразование базовых типов данных». Другие аргументы метода передаются без преобразования.

Если *получатель* является *объектом-классом* (который можно создать, просто указав имя класса), то вызывается указанный метод *класса*. В противном случае *получатель* является *экземпляром класса*, и тогда вызывается соответствующий метод *экземпляра*.

Если *получатель* является переменной или выражением со статическим контролем типа, то компилятор ищет метод в определении класса (или среди любых наследуемых методов) и преобразует любые аргументы (когда это возможно), чтобы они соответствовали ожидаемым аргументам данного метода. Например,

если метод должен получать значение с плавающей точкой и ему передано значение целого типа, соответствующий аргумент автоматически преобразуется при вызове этого метода.

Если получатель является указателем на null-объект, то есть nil, ему тоже можно передавать сообщения. Если метод, связанный с сообщением, возвращает объект, то выражение с этим сообщением дает значение nil. Если метод не возвращает какого-либо объекта, то значение этого выражения не определено.

Если один и тот же метод определен более чем в одном классе (путем явного определения или в результате наследования), компилятор проверяет среди этих классов соответствие типам аргументов и возвращаемого значения.

Все аргументы передаются методу по значению, поэтому их значения не могут быть изменены методом. Если методу передается указатель, то метод может изменять значения, на которые ссылается этот указатель, но метод все же не может изменять значение самого указателя.

### Формат 3

*получатель.свойство*

Это вызов getter-метода, то есть метода-получателя (по умолчанию *свойство*) для *получателя*, если выражение не используется как lvalue-выражение (см. Формат 4). Имя getter-метода можно изменять с помощью директивы @property, и тогда это будет вызываемый метод.

Если используется имя getter-метода по умолчанию, то приведенное выше выражение эквивалентно следующему.

[*получатель* *свойство*]

### Формат 4

*получатель.свойство = выражение*

Это вызов setter-метода (метода-установщика), связанного со свойством *свойство*, и в качестве его аргумента передается значение *выражения*. По умолчанию вызывается setter-метод *setСвойство:*, если для этого свойства не было назначено имя другого setter-метода с использованием предшествующей директивы @property.

Если используется имя setter-метода по умолчанию, то приведенное выше выражение эквивалентно следующему.

[*получатель* *setСвойство:* *выражение*]

## Программные операторы

Программным оператором (program statement) является любое допустимое выражение (обычно присваивание или вызов функции), которое заканчивается точкой с запятой, или это один из специальных операторов, описанных ниже. Перед любым оператором можно ставить необязательную *метку*; метка состоит из идентификатора, после которого ставится символ «двоеточие» (см. также оператор *goto*).

## Составные операторы

Программные операторы, содержащиеся в фигурных скобках, называются *составным оператором*, или *блоком*, и могут находиться в любом месте программы, где допустим хоть один оператор. Блок может содержать свой собственный набор объявлений переменных, которые могут заменять одноименные переменные, определенные вне этого блока. Областью действия таких локальных переменных является блок, в котором они определены.

## Оператор break

### Общий формат

```
break;
```

Выполнение оператора `break` внутри области действия оператора `for`, `while`, `do` или `switch` вызывает прекращение работы этого оператора. Выполнение продолжается с оператора, непосредственно следующего после цикла или переключателя (`switch`).

## Оператор continue

### Общий формат

```
continue;
```

Выполнение оператора `continue` внутри цикла вызывает пропуск операторов, которые следуют в этом цикле непосредственно после `continue`. В противном случае выполнение цикла продолжается обычным образом.

## Оператор do

### Общий формат

```
do
 программныйОператор
 while (выражение);
```

Если *выражение* имеет ненулевое значение, то выполняется *программныйОператор*. Отметим, что поскольку *выражение* вычисляется каждый раз после того, как выполнен *программныйОператор*, здесь гарантируется, что *программныйОператор* будет выполнен хотя бы один раз.

## Оператор for

### Формат 1

```
for (выражение_1; выражение_2; выражение_3)
 программныйОператор
```

*Выражение\_1* вычисляется один раз, когда начинается выполнение цикла. Затем вычисляется *выражение\_2*. Если это выражение имеет ненулевое значение,

то выполняется *программныйОператор* и затем вычисляется *выражение\_3*. Это происходит, пока *выражение\_2* имеет ненулевое значение. Поскольку *выражение\_2* вычисляется каждый раз перед тем, как выполняется *программныйОператор*, этот оператор может быть никогда не выполнен, если *выражение\_2* имеет значение 0 при первом входе в цикл.

Для *выражения\_1* можно объявить переменные, локальные для данного цикла *for*. Эти переменные действительны в области действия этого цикла *for*. Например, в

```
for (int i = 0; i < 100; ++i)
```

...

объявляется целая переменная *i*, которой присваивается начальное значение 0, когда начинается цикл. Эта переменная доступна для любых операторов внутри этого цикла, но она недоступна после завершения цикла.

## Формат 2

```
for (var in выражение)
 программныйОператор
```

В этом варианте цикла *for* задается *быстрое перечисление* (*fast enumeration*). Для переменной *var* можно также объявлять ее тип, что делает область ее действия локальной для данного цикла *for*. Выражение *выражение* дает результат, который подчиняется протоколу *NSFastEnumeration*. Обычно *выражение* является коллекцией, например, массивом или словарем.

При каждом прохождении цикла *for* переменной *var* присваивается следующий объект, полученный при начальном вычислении *выражения*, и выполняется тело цикла, которое представляет *программныйОператор*. Выполнение цикла завершается, когда выполнен перебор всех объектов в *выражении*.

Отметим, что в данном цикле *for* нельзя изменить содержимое коллекции. Если это происходит, то создается исключение.

Для массива происходит перечисление по порядку каждого из его элементов. Для словаря происходит перечисление каждого ключа без определенного порядка. Для набора (*set*) происходит перечисление каждого члена набора без определенного порядка.

## Оператор *goto*

### Общий формат

```
goto идентификатор;
```

Выполнение оператора *goto* вызывает передачу управления непосредственно оператору с меткой *идентификатор*. Оператор с меткой должен находиться в той же функции или методе, где и *goto*.

## Оператор if

### Формат 1

```
if (выражение)
 программныйОператор
```

Если результат вычисления выражения не равен нулю, то выполняется программныйОператор; иначе он пропускается.

### Формат 2

```
if (выражение)
 программныйОператор1
else
 программныйОператор2
```

Если значение выражения не равно нулю, то выполняется программныйОператор1; иначе выполняется программныйОператор2. Если программныйОператор2 является еще одним оператором if, образуется цепочка if-else if, например,

```
if (выражение1)
 программныйОператор1
else if (выражение2)
 программныйОператор2
...
else
 программныйОператорn
```

Предложение else всегда связано с последним оператором if, который не содержит else. При необходимости можно использовать фигурные скобки, чтобы изменить эту связь.

## Оператор null

### Общий формат

```
;
```

Выполнение null-оператора (пустого оператора) не оказывает никакого влияния и используется в основном, чтобы выполнить требование программного оператора в цикле for, do или while. В следующем операторе выполняется копирование символьной строки, указанной с помощью from, в строку, указанную с помощью to.

```
while (*to++ = *from++)
;
```

В этом операторе используется null-оператор, чтобы выполнить требование того, что после выражения цикла while должен присутствовать программный оператор.

## Оператор return

### Формат 1

```
return;
```

Выполнение оператора `return` вызывает немедленный возврат выполнения программы в вызывающую функцию или метод. Этот формат можно использовать только для возврата из функции или метода, которые не возвращают никакого значения.

Если выполнение доходит до конца функции или метода, не встретив оператор `return`, то происходит возврат, как при выполнении оператора `return` в этом формате, поэтому в таком случае не возвращается никакого значения.

### Формат 2

```
return выражение;
```

Вызывающей функции или методу возвращается значение *выражения*. Если тип *выражения* не согласуется с типом возвращаемого значения, указанным в объявлении функции или метода, то его значение автоматически преобразуется перед возвратом в объявленный тип.

## Оператор switch

### Общий формат

```
switch (выражение)
{
 case константа_1:
 программный оператор
 программный оператор
 ...
 break;
 case константа_2:
 программный оператор
 программный оператор
 ...
 break;
 ...
 case константа_n:
 программный оператор
 программный оператор
 ...
 break;
 default:
 программный оператор
 программный оператор
 ...
 break;
}
```

Выполняется вычисление и сравнение *выражения* со значениями константных выражений *константа\_1*, *константа\_2*, ..., *константа\_n*. Если значение *выражения* совпадает с одним из этих case-значений, то выполняются последующие программные операторы. Если ни одно из case-значений не совпадает со значением *выражения*, то выполняется (если он включен) вариант по умолчанию *default*. Если вариант *default* не включен, то не выполняются никакие операторы, включенные в *switch*.

Результат вычисления *выражения* должен быть целого типа, и никакие два варианта case не должны иметь одинаковое значение. Отсутствие оператора *break* в определенном case-варианте вызывает продолжение выполнения в следующем case-варианте.

## Оператор while

### Общий формат

```
while (выражение)
 программныйОператор
```

Если значение *выражения* не равно нулю, выполняется *программныйОператор*. Поскольку *выражение* вычисляется каждый раз перед тем, как выполняется *программныйОператор*, этот оператор может никогда не выполняться.

## Обработка исключений

Для обработки исключений во время выполнения нужно включить операторы, которые могут генерировать исключение, в блок @try, который имеет общий формат

```
@try
 программныйОператор 1
@catch (исключение)
 программныйОператор 2
@catch (исключение)
 ...
@finally
 программныйОператор n
```

Если исключение выдает *программныйОператор 1*, то проверяются (по порядку) последующие блоки @catch на совпадение соответствующего исключения с выданным исключением. Если да, то будет выполнен соответствующий *программныйОператор*. Независимо от факта выдачи и перехвата исключения будет выполнен блок @finally (если он задан).

## Препроцессор

Препроцессор анализирует исходный файл до того, как компилятор рассмотрит сам код. Препроцессор выполняет следующие действия.

1. Он заменяет триграммы (группы из трех последовательных символов) на их эквиваленты (см. выше раздел «Составные операторы»).
2. Он объединяет в одну строку любые строки, которые заканчиваются обратным слешем (\).
3. Он разделяет программу на поток маркеров.
4. Он удаляет комментарии, заменяя их одним пробелом.
5. Он обрабатывает препроцессорные директивы (см. ниже раздел «Директивы препроцессора») и раскрывает макросы.

## Последовательности из триграмм

Для обработки наборов символов, не соответствующих ASCII, используются следующие трехсимвольные последовательности (*триграммы*), которые распознаются и обрабатываются специальным образом там, где они находятся в программе (а также внутри символьных строк).

| Триграмма | Значение |
|-----------|----------|
| ??=       | #        |
| ??(       | [        |
| ??)       | ]        |
| ??<       | {        |
| ??>       | }        |
| ??/       | \        |
| ??'       | ^        |
| ??!       |          |
| ??-       | ~        |

## Директивы препроцессора

Все директивы препроцессора начинаются с символа #, который должен быть первым символом в строке, отличным от пробела. После # могут следовать один или несколько символов «пробел» или tab.

### Директива `#define`

#### Формат 1

```
#define имя текст
```

Определяется имя идентификатора для препроцессора, это имя связывается с *текстом*, который начинается после первого пробела, следующего за *именем*, и заканчивается концом строки. При последующем использовании *имени* в программе оно заменяется *текстом*.

## Формат 2

```
#define имя (параметр_1, параметр_2,..., параметр_n) текст
```

Определяется макрос *имя*, принимающий аргументы *параметр\_1, параметр\_2, ..., параметр\_n*, каждый из которых является идентификатором. При последующем использовании *имени* в программе со списком аргументов происходит подстановка *текста*, причем аргументы вызова этого макроса заменяют все экземпляры соответствующих параметров внутри *текста*.

Если макрос принимает переменное число параметров, то в конце списка аргументов используются три точки. Остальные аргументы в списке обозначаются специальным идентификатором *\_VA\_ARGS\_*. Например, ниже определяется макрос с именем *myPrintf*, принимающий переменное число аргументов.

```
#define myPrintf(...) printf ("DEBUG: " _VA_ARGS_);
```

Этот макрос можно использовать, например, в форме

```
myPrintf ("Hello world!\n");
```

или

```
myPrintf ("i = %i, j = %i\n", i, j);
```

Если для определения требуется более чем одна строка, то для продолжения каждой строки ее нужно заканчивать символом обратного слеша. После определения *имени* его можно использовать в любом месте данного файла.

В директивах *#define*, которые принимают аргументы, можно использовать оператор *#*, после которого следует имя аргумента. Препроцессор помещает в кавычки фактическое значение, передаваемое макросу при его вызове, то есть значение превращается в символьную строку. Например, определение

```
#define printint(x) printf (# x "= %d\n", x)
```

при вызове

```
printint (count);
```

раскрывается препроцессором как

```
printf ("count" "= %i\n", count);
```

или эквивалентно как

```
printf ("count = %i\n", count);
```

Препроцессор помещает символ \ перед любой кавычкой или символами \ при выполнении этой операции преобразования в строку. Например, в случае определения

```
#define str(x) # x
```

вызов

```
str (The string "\t"contains a tab)
```

раскрывается следующим образом

"The string \"\\t\" contains a tab"

В директивах `#define`, принимающих аргументы, допускается также оператор `##`. Перед ним (или после него) ставится имя аргумента для макроса. Препроцессор берет значение, передаваемое при вызове макроса, и создает один маркер из этого аргумента и из маркера, который следует за `##` (или предшествует `##`). Например, в случае определения макроса

```
#define printx(n) printf ("%i\n", x ## n);
```

вызов

```
printx (5)
```

дает

```
printf ("%i\n", x5);
```

Определение

```
#define printx(n) printf ("x" # n "= %i\n", x ## n);
```

при вызове

```
printx(10)
```

дает

```
printf ("x10 = %i\n", x10);
```

после подстановки и конкатенации символьных строк.

Вокруг операторов `#` и `##` можно не ставить пробелы.

### **Директива `#error`**

#### **Общий формат**

```
#error текст
```

...

Указанный текст записывается препроцессором как сообщение об ошибке.

### **Директива `#if`**

#### **Формат 1**

```
#if константное_выражение
```

...

```
#endif
```

Вычисляется *константное выражение*. Если результат не равен нулю, то обрабатываются все строки программы до директивы `#endif`; в противном случае они автоматически пропускаются и не обрабатываются препроцессором или компилятором.

#### **Формат 2**

```
#if константное_выражение_1
```

...

```
#elif константное_выражение_2
...
#elif константное_выражение_n
...
#else
...
#endif
```

Если *константное\_выражение\_1* не равно нулю, то обрабатываются все строки программы до *#elif*, а остальные строки до *#endif* пропускаются. В противном случае, если *константное\_выражение\_2* не равно нулю, то обрабатываются все строки программы до следующей директивы *#elif*, а остальные строки до *#endif* пропускаются. Если все константные выражения равны нулю, то обрабатываются строки после *#else* (если включена эта директива).

Как часть константного выражения можно использовать специальный оператор *defined*. Например, при использовании

```
#if defined (DEBUG)
...
#endif
```

будет обрабатываться код между *#if* и *#endif*, если ранее был определен идентификатор *DEBUG* (см. также *#ifdef* в следующем разделе). Идентификатор не обязательно заключать в круглые скобки, то есть

```
#if defined DEBUG
```

действует точно так же.

### Директива *#ifdef*

#### Общий формат

```
#ifdef идентификатор
...
#endif
```

Если значение *идентификатора* было ранее определено (с помощью *#define* или опции командной строки *-D* при компиляции программы), то обрабатываются все строки программы до *#endif*; в противном случае они пропускаются. Как и в случае директивы *#if*, с директивой *#ifdef* можно использовать директивы *#elif* и *#else*.

### Директива *#ifndef*

#### Общий формат

```
#ifndef идентификатор
...
#endif
```

Если значение *идентификатора* не было ранее определено, то обрабатываются все строки программы до *#endif*; в противном случае они пропускаются. Как и в случае директивы *#if*, с директивой *#ifndef* можно использовать директивы *#elif* и *#else*.

## Директива `#import`

### Формат 1

```
#import "имяФайла"
```

Если файл *имяФайла* был ранее включен в программу, этот оператор пропускается. В противном случае препроцессор ищет сначала файл *имяФайла* в папках, которые определены реализацией. Обычно поиск начинается с той же папки, где содержится исходный файл, и если файл не найден, то выполняется поиск в последовательности стандартных мест, определенных реализацией. После того, как файл найден, его содержимое включается в программу в том месте, где находится директива `#import`. Препроцессорные директивы, содержащиеся во включенном файле, тоже анализируются. Поэтому сам включенный файл может содержать другие директивы `#import` или `#include`.

### Формат 2

```
#import <имяФайла>
```

Если этот файл не был включен ранее, препроцессор ищет его только в стандартных местах. В частности, текущая папка исходных файлов исключается из поиска. В остальном действия, выполняемые после того, как файл найден, идентичны тому, что описано выше.

При любом формате можно указать определенное ранее имя, после чего происходит раскрытие этого имени. Например, можно использовать следующую последовательность.

```
#define ROOTOBJECT <NSObject.h>
...
#import ROOTOBJECT
```

## Директива `#include`

Эта директива действует таким же образом, как `#import`, но не выполняется проверка предшествующего включения указанного header-файла.

## Директива `#line`

### Общий формат

```
#line константа "имяФайла"
```

Эта директива указывает компилятору, что последующие строки программы нужно обрабатывать в предположении, что имя исходного файла — *имяФайла* и что отсчет номеров строк во всех последующих строках начинается с *константы*. Если *имяФайла* не указано, то используется имя файла, указанное последней директивой `#line`, или имя исходного файла (если ранее не было указано никакого имени файла).

Директива `#line` в основном используется для задания имени файла и номера строки, которые выводятся при выдаче ошибки компилятором.

## Директива #pragma

### Общий формат

`#pragma текст`

Препроцессор выполнит некоторые действия, определенные реализацией. Например,

`#pragma loop_opt(on)`

вызывает специальную оптимизацию циклов в определенном компиляторе. Если эту директиву встретит компилятор, который не распознает указание `loop_opt`, то она будет игнорироваться.

## Директива #undef

### Общий формат

`#undef идентификатор`

Указанный идентификатор становится нонпределенным для препроцессора. Последующие директивы `#ifdef` или `#ifndef` действуют так, как будто данный идентификатор никогда не был определен.

## Директива #

Это null-директива, и она игнорируется препроцессором.

## Заранее определенные идентификаторы

Следующие идентификаторы определены препроцессором.

| Идентификатор               | Описание                                                                                 |
|-----------------------------|------------------------------------------------------------------------------------------|
| <code>_LINE_</code>         | Номер текущей компилируемой строки.                                                      |
| <code>_FILE_</code>         | Имя текущего компилируемого исходного файла.                                             |
| <code>_DATE_</code>         | Дата компилируемого файла в формате «Ммм дд гггг».                                       |
| <code>_TIME_</code>         | Время компилируемого файла в формате «чч:мм:сс».                                         |
| <code>_STDC_</code>         | Определен как 1, если компилятор согласуется со стандартом ANSI, и 0 в противном случае. |
| <code>_STDC_HOSTED_</code>  | Определен как 1, если данная реализация поддерживается (hosted), и 0 в противном случае. |
| <code>_STDC_VERSION_</code> | Определен как 199901L.                                                                   |

## Приложение С

# Исходный код адресной книги

В справочных целях здесь приводятся в полном виде файлы секций `interface` и `implementation` для примера адресной книги, с которым мы работали в части II. Сюда включены определения классов `AddressCard` и `AddressBook`. Вы должны реализовать эти классы на своем компьютере; затем определения этих классов нужно расширить, чтобы сделать их более применимыми и усилить их возможности. Для вас это станет превосходным способом изучения языка и ознакомления с созданием программ, работой с классами и объектами, а также работой с Foundation framework.

## Файл секции `interface` для `AddressCard`

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

@interface AddressCard : NSObject <NSCopying, NSCoding>
{
 NSString *name;
 NSString *email;
}

@property (nonatomic, copy) NSString *name, *email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail;
-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail;
-(NSComparisonResult) compareNames: (id) element;

-(void) print;

@end
```

## Файл секции interface для AddressBook

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

@interface AddressBook: NSObject <NSCopying, NSCoding>
{
 NSString *bookName;
 NSMutableArray *book;
}

@property (nonatomic, copy) NSString *bookName;
@property (nonatomic, copy) NSMutableArray *book;

-(id) initWithName: (NSString *) name;
-(void) sort;
-(void) addCard: (AddressCard *) theCard;
-(void) removeCard: (AddressCard *) theCard;
-(int) entries;
-(void) list;
-(AddressCard *) lookup: (NSString *) theName;

-(void) dealloc;

@end
```

## Файл секции implementation для AddressCard

```
#import "AddressCard.h"

@implementation AddressCard

@synthesize name, email;

-(void) setName: (NSString *) theName andEmail: (NSString *) theEmail
{
 [self setName: theName];
 [self setEmail: theEmail];
}

// Сравнение двух имен из указанных адресных карточек
-(NSComparisonResult) compareNames: (id) element
{
```

```
 return [name compare: [element name]];
 }

-(void) print
{
 NSLog (@"===== ");
 NSLog (@"|");
 NSLog (@"| %-31s |", [name UTF8String]);
 NSLog (@"| %-31s |", [email UTF8String]);
 NSLog (@"|");
 NSLog (@"|");
 NSLog (@"|");
 NSLog (@"| 0 0 |");
 NSLog (@"===== ");
}

-(AddressCard *) copyWithZone: (NSZone *) zone
{
 AddressCard *newCard = [[AddressCard allocWithZone: zone] init];
 [newCard retainName: name andEmail: email];
 return newCard;
}

-(void) retainName: (NSString *) theName andEmail: (NSString *) theEmail
{
 name = [theName retain];
 email = [theEmail retain];
}

-(void) encodeWithCoder: (NSCoder *) encoder
{
 [encoder encodeObject: name forKey: @"AddressCardName"];
 [encoder encodeObject: email forKey: @"AddressCardEmail"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
 name = [[decoder decodeObjectForKey: @"AddressCardName"] retain];
 email = [[decoder decodeObjectForKey: @"AddressCardEmail"] retain];

 return self;
}

-(void) dealloc
```

```
{
 [name release];
 [email release];
 [super dealloc];
}
@end
```

## Файл секции **implementation** для AddressBook

```
#import "AddressBook.h"

@implementation AddressBook

@synthesize book, bookName;

// задание имени адресной книги и пустой книги

-(id) initWithName: (NSString *) name{
 self = [super init];

 if (self) {
 bookName = [[NSString alloc] initWithString: name];
 book = [[NSMutableArray alloc] init];
 }

 return self;
}

-(void) sort
{
 [book sortUsingSelector: @selector(compareNames:)];
}

-(void) addCard: (AddressCard *) theCard
{
 [book addObject: theCard];
}

-(void) removeCard: (AddressCard *) theCard
{
 [book removeObjectIdenticalTo: theCard];
}

-(int) entries
```

```
{
 return [book count];
}

-(void) list
{
 NSLog (@"===== Contents of: %@ =====", bookName);

 for (AddressCard *theCard in book)
 NSLog (@"%-20s %-32s", [theCard.name UTF8String],
 [theCard.email UTF8String]);

 NSLog (@"=====");
}

// поиск адресной карточки по имени - требуется точное совпадение

-(AddressCard *) lookup: (NSString *) theName
{
 for (AddressCard *nextCard in book)
 if ([[nextCard name] caseInsensitiveCompare: theName] == NSOrderedSame)
 return nextCard;

 return nil;
}

-(void) dealloc
{
 [bookName release];
 [book release];
 [super dealloc];
}

-(void) encodeWithCoder: (NSCoder *) encoder
{
 [encoder encodeObject:bookName forKey: @"AddressBookBookName"];
 [encoder encodeObject:book forKey: @"AddressBookBook"];
}

-(id) initWithCoder: (NSCoder *) decoder
{
 bookName = [[decoder decodeObjectForKey: @"AddressBookBookName"] retain];
 book = [[decoder decodeObjectForKey: @"AddressBookBook"] retain];

 return self;
}
```

```
// Метод для протокола NSCopying

-(id) copyWithZone: (NSZone *) zone
{
 AddressBook *newBook = [[self class] allocWithZone: zone];

 [newBook initWithName: bookName];
 [newBook setBook: book];

 return newBook;
}
@end
```

## Приложение D

# Ресурсы

В этом приложении содержится список некоторых ресурсов с дополнительной информацией. Мы собрали здесь ресурсы, посвященные программированию на языке C, Objective-C, в Сосоа и для iPhone/iTouch. Этот список послужит хорошей отправной точкой для поиска нужной информации.

## Ответы на вопросы упражнений, опечатки и пр.

На веб-сайте издателя [www.informit.com/register](http://www.informit.com/register) вы можете найти ответы на упражнения.

## Язык Objective-C

К этим ресурсам вы можете обратиться за дополнительной информацией по языку Objective-C.

### КНИГИ

- Objective-C 2.0 Programming Language (Язык программирования Objective-C). Apple Computer, Inc., 2008. Лучший из имеющихся справочников по языку Objective-C. Мы рекомендуем обратиться к нему после завершения работы с этой книгой. Доступ к тексту книги можно получить в окне Xcode Help-> Documentation или в Интернете на веб-сайте Apple. Pdf-версия находится по ссылке: <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>.
- Object-Oriented Programming: An Evolutionary Approach (Объектно-ориентированное программирование: эволюционный подход), Second Edition. Brad Cox and Andy Novobilski. Addison-Wesley, 1991. Первоисточник всех сведений по Objective-C. Бред Кокс, разработчик этого языка, был одним из авторов этой книги.
- Objective-C Pocket Reference (Карманный справочник по Objective-C). Andrew M. Duncan. O'Reilly Associates Inc., 2003. Краткий справочник по языку Objective-C.

## Веб-сайты

- <http://developer.apple.com/documentation/Cocoa/ObjectiveCLanguage-date.html>. Часть веб-сайта Apple, посвященная языку Objective-C. Содержит, помимо остального, онлайн-документацию, примеры кодов и технические замечания.

## Язык программирования С

Поскольку С является базовым языком программирования, вам может потребоваться более глубокое изучение этого языка. Он широко используется более 25 лет, и информации о нем существует более чем достаточно.

## Книги

- Programming in C (Программирование на С). Third Edition. Stephen Kochan. Sams Publishing, 2004. Это моя первая книга, которая несколько раз переиздавалась и обновлялась. В ней детально рассматриваются многие возможности языка, о которых говорится в главе 13 настоящей книги.
- The C Programming Language (Язык программирования С), Second Edition. Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. Эта книга давно является как авторитетным руководством, так и справочником по данному языку. Это была первая книга по С, ее соавтором был Деннис Ричи, создавший этот язык.
- C: A Reference Manual (C: справочное руководство), Fifth Edition. Samuel P. Harbison, III and Guy L. Steele, Jr. Prentice Hall, 2002. Еще одно превосходное справочное издание для программистов, использующих С.

## Cocoa

Если вы всерьез собираетесь разрабатывать приложения в системе Mac OS X, то вам нужно научиться программировать в среде Cocoa. Существует много книг по Cocoa, и постоянно появляются новые. Наберите «Cocoa» в окне поиска amazon.com, и вы увидите огромный список. Ниже приводится лишь небольшая выборка из этого списка.

## Книги

- *Introduction to Cocoa Fundamentals Guide* (Введение в основы Cocoa). Apple Computer, Inc., 2007. Превосходная книга по разработке приложений с помощью Cocoa. Доступ к тексту книги можно выполнить из окна Xcode Documentation. Pdf-версия доступна по ссылке: <http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals.pdf>.
- *Cocoa Programming for Mac OS X* (Программирование в Cocoa для Mac OS X), Third Edition. Aaron Hillegass. Addison-Wesley, 2008. Хорошее введение в Cocoa, написанное простым языком.
- *Cocoa in a Nutshell*. Michael Beam and James Duncan Davidson. O'Reilly & Associates, Inc., 2003. Справочный ресурс по многим классам и методам, являющимся частью системы разработки Cocoa.
- *Learning Cocoa with Objective-C* (Изучение Cocoa с помощью Objective-C), Second Edition. James Duncan Davidson and Apple Computer, Inc. O'Reilly & Associates, Inc., 2002. Это введение по программированию в Cocoa.

## Веб-сайты

- <http://developer.apple.com/cocoa/>. Основной веб-сайт Apple для разработчиков приложений в Cocoa. Имеется документация, примеры кодов, технические замечания и разнообразная информация.
- <http://www.cocoadevcentral.com/>. Веб-сайт для тех, кто хочет научиться программировать в Cocoa, используя Objective-C.
- <http://www.cocoadev.com/>. Открытый веб-сайт, на котором может вносить изменения любой человек. Здесь можно найти много полезной информации.

## Разработка приложений для iPhone и iTouch

Популярность iPhone, несомненно, будет способствовать выпуску литературы по разработке приложений для этого устройства. Ниже приводятся названия книг, которые были опубликованы или анонсированы на момент отправки в печать этой книги.

## Книги

- iPhone OS Programming Guide (Руководство по программированию в iPhone OS). Apple Computer, Inc., 2008. Превосходная книга, рассматривающая разработку приложений для iPhone. Для ее чтения можно использовать окно Xcode Documentation. Pdf-версия доступна по ссылке: <http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf>.
- The iPhone Developer's Cookbook: Building Applications with the iPhone SDK (Рецептурный справочник разработчика для iPhone: создание приложений с помощью SDK iPhone). Erica Sadun. Addison-Wesley Professional, 2008. Содержит «рецепты» написания различных типов приложений iPhone.
- Beginning iPhone Development: Exploring the iPhone SDK (Начало разработки для iPhone: изучение SDK iPhone). Dave Mark. Apress, 2008. Введение в написание приложений для iPhone и iPod Touch.
- iPhone Application Development: Building Applications for the AppStore (Разработка приложений iPhone: создание приложений для AppStore). Jonathan Zdziarski, 2009, O'Reilly Media, Inc., 2002. Пока в производстве.

## Веб-сайты

- <http://developer.apple.com/iphone/>. Основной веб-сайт Apple для разработчиков приложений iPhone (называется iPhone DevCenter). Здесь можно найти документацию, учебные видеоматериалы, примеры кодов, технические замечания и разнообразную информацию. Здесь вы можете также загрузить iPhone SDK.
- <http://www.iphonedevcentral.org/>. Веб-сайт, на котором можно получить бесплатные учебные материалы, и форум, где можно обмениваться идеями и задавать вопросы.

# Предметный указатель

- ##, оператор, 247  
#, директива, 567  
#, оператор, 246  
#define, оператор, 239-241, 562-564  
    аргументы, 243  
    константные значения, 241  
    макросы, 244-245  
        #, оператор, 246  
        ##, оператор, 247  
     массивы, 241  
    местоположение, 240  
    обратный слэш (\), символ, 243  
определенные имена, 241  
определенные имена, 241-245  
препроцессор, 239-241  
    синтаксис, 240  
#elif, оператор препроцессора, 252  
#else, оператор, 250-251  
#endif, оператор, 250-251  
#егто, директива, 564  
#if, оператор, 252, 564-565  
#ifdef, оператор, 250-251, 565  
#ifndef, оператор, 250-251, 565  
#import, оператор, 247-250, 565-566  
#include, оператор, 247-250, 566  
#line, директива, 566  
#pragma, директива, 566  
#undef, оператор, 253, 567  
% (процент), 24  
&& (логический оператор AND), 107  
&, оператор, указатели, 283  
(-), оператор уменьшения (декремента), 292, 296-299  
(\*), оператор косвенного обращения «звездочка», 284  
(++), оператор приращения (инкремента), 292, 296-299  
: (двоеточие), условный оператор, 128  
; (точка с запятой), 21  
? (вопросительный знак), условный оператор, 128  
@class, директива, 167, 171  
@implementation, секция (объектно-ориентированное программирование), 32, 37-38  
@interface, секция (объектно-ориентированное программирование), 32  
    имена, выбор, 33-34  
    методы, 33-37  
объявления классов, 133-134  
переменные экземпляра, 33-35  
родительские классы, 33  
@package, директива, 208  
@private, директива, 208  
@protected, директива, 208  
@protocol, директива, 233  
@public, директива, 208  
@selector, директива, 197  
@synthesize, директива, 139  
@try, блоки, 200-202  
\ (обратный слэш), в операторе #define, 243  
|| (логический оператор OR), 107  
= (одиночный знак равенства), оператор if, 106  
== (двойной знак равенства), оператор if, 106
- ## A
- addCard:, метод, 356  
addObject:, метод, 344, 373, 408  
AddressBook, класс  
    файл секции implementation, 572-574  
    файл секции interface, 570  
AddressCard, класс, 411-412  
    файл секции implementation, 571  
    файл секции interface, 569  
alloc, метод, 39-40, 324  
allocF, метод класса, как вход, 211-213  
AND, оператор (&), 107  
AND, оператор (побитовый), 68  
AppKit (Application Kit), фреймворк (Cocoa), 317, 456-457  
argc, аргумент, функция main, 308  
arguments, метод, 396  
argv, аргумент, функция main, 308  
arrayWithCapacity:, метод, 342  
arrayWithObjects:, метод, 342  
ASCII, формат хранения символов, 114  
auto, ключевое слово, 213  
autorelease, сообщения, 324, 419
- ## B
- BOOL, специальный тип, 127  
Bool, тип данных, 73  
break, операторы, 95, 123, 557  
    операторы switch, 121  
Build, меню, Xcode, 15

**C**

**Calculator**, класс, 115, 488-489  
 операторы присваивания, 65-67  
**changeCurrentDirectoryPath:**, метод, 386  
**char**, тип данных, 49-53  
**charPtr**, переменная, 286  
**char-символы**, unichar-символы, 326  
**class**, сообщение, 196  
**class.h**, файлы, 133-134  
**Cocoa**, 455  
 AppKit, фреймворк, 456-457  
 Cocoa Touch, 456-457  
 Foundation, фреймворк, 456-457  
 веб-ресурсы, 577  
**compare**, метод, 325, 330  
**compareNames:**, метод, 364-365  
**Complex**, класс, 187-190, 195  
**Complex**, тип данных, 73  
**conformToProtocol:**, метод, протоколы, 233  
**const**, ключевое слово, 214, 523-524  
**containsObject:**, метод, 360, 373  
**contentsAtPath:**, метод, 384  
**continue**, операторы, 96, 557  
**convertToNum:**, метод, 101-104  
**convertToString:**, метод, 488  
**copy**, метод, 424-425  
 классы, добавление к, 429-432  
**copyPath:toPath:**, метод, 405  
**copyPath:toPath:handler:**, метод, 383, 393, 397  
**copyWithZone:**, метод, 430-434  
**count**, метод класса, как вход, 211-213  
**countForObject:**, метод, 374  
**createFileAtPath:contents:attributes:**, метод, 384  
**currentDirectory**, метод, 391  
**С-строки**, 296

**D**

**dataValue**, переменная, 192  
**dealloc**, метод, 417  
 замещение, 179  
**decodeObjectForKey:**, метод, 440  
**defaultManager**, сообщения, 380  
**deleteCharactersInRange:**, метод, 336  
**denominator**, целая переменная экземпляра, 101  
**description**, метод, 374  
**dictionaryWithObjectsAndKeys:**, метод, 368-369  
**directoryContentsAtPath:**, метод, 387, 389  
**divide:**, метод, 120  
**do**, операторы, 94-95, 557  
**doesNotRecognize:**, метод, 198  
**double**, тип данных, 49-51  
**do-while**, циклы, 344

**E**

**else if**, конструкция, 111-114, 117-120  
 логическое решение с п значениями, 111  
 условный оператор, 129  
**else**, предложение, 110-111, 119  
**encodeObjectForKey:**, метод, 440, 442  
**encodeWithCoder:**, метод, 440, 442  
**enumerationAtPath:**, метод, 389  
**enumeratorAtPath:**, метод, 387  
**escape-последовательности**, 511  
**extern**, ключевое слово, 209-212

**F**

**fileAttributesAtPath:traverseLink:**, метод, 380, 382  
**fileExistsAtPath:**, метод, 389  
**fileHandleForUpdatingAtPath:**, метод, 398  
**fileHandleForWritingAtPath:**, метод, 398  
**float**, тип данных, 49, 51  
**for**, оператор, 78-89, 557  
 варианты циклов, 88-89  
 ввод из окна терминала, 85-86  
 ввод с клавиатуры, 84-86  
 вложенные циклы, 86-88  
**for**, циклы, порядок выполнения, 81  
**Foundation Framework**, 317  
 Cocoa, 456-457  
**NSCountedSet**, класс, 373  
**typedef**, оператор, 219  
 архивация, 317  
 документация Мак, 317-319  
 методы  
 copy, метод, 424-425  
 mutableCopy:, метод, 424-425  
 методы инициализации, 206  
 мутабельные строки, 333-337  
 объекты-массивы, 341-344  
 создание адресных карточек, 345-356  
 создание адресных книг, 345-365  
 объекты-наборы, 370-374  
 объекты-словари, 367-369  
 протоколы, 231  
 строковые объекты, 326-340  
 числовые объекты, 322-326  
**Foundation**, библиотека, функция NSLog, 327  
**Foundation**, строковый класс, 250  
**Fraction**, класс, 30-32, 39, 101, 157, 195-197, 485-488  
**convertToNum:**, метод, 101-104  
**copy**, метод, добавление, 429-432  
 метод класса, 211-213

**G**

gcc, команда, 17

**goto**, операторы, 306, 558  
**GUI** (графические пользовательские интерфейсы), 3

**I**

**id**, объявление объектов, 554-555  
**id**, тип данных, 55, 191-194, 311  
**if**, оператор, 99-104, 558  
  = (знак равенства), 106  
  вложенные, 109-111  
  условие удовлетворяется, 125  
**if-else**, оператор, 104-106  
**Imaginary**, тип данных, 73  
**include**-файлы, 250  
**indexForObject:**, двоеточие, 360  
**init**, метод, 40  
**initialize**, метод, 207  
**initVar:**, метод, 158  
**initWithCoder:**, метод, 440, 442  
**initWithName:**, метод, 352, 354  
**insertStringAtIndex:**, метод, 336  
**int**, тип данных, 49-50  
**Interface Builder**, 460  
**intersect:**, метод, 373  
**IntPtr**, переменная, 283, 285  
**iPhone**  
  Cocoa Touch, 456-457  
  веб-ресурсы, 578  
  калькулятор дробей, создание, 476-480  
    Calculator class, 488-489  
    Fraction, класс, 485-488  
  контроллер представлений,  
    определение, 480-485  
  пользовательский интерфейс,  
    проектирование, 490  
  приложение, создание, 461-463  
    интерфейс, проектирование, 467-468,  
      471-472, 476  
    код, ввод, 463, 466  
  разработка в Objective-C, 2  
  шаблоны приложений, 461  
**iPhone SDK**, установка, 459  
**iPhone**, имитатор, 460  
**isa**, компонент, 310  
**isEqualToString:**, метод, 325  
**isEqualToNumber:**, метод, 325

**L**

**lastPathComponent:**, метод, 391  
**length**, метод, строковые объекты, 330  
**LinuxSTEP**, среда разработки, разработка  
  Objective-C, 1  
**long long**, квалификатор, 53-54  
**long**, квалификатор, 53-54

**lookup:**, метод, 356-362  
**lowercaseString:**, метод, 330  
**lvalue** (l-значения), выражения, 524

**M**

**Macintosh, Foundation framework**,  
  документация, 317, 319  
**main**, функция, 308  
**movePath:toPath:**, метод, 382, 405  
**movePath:toPath:handler:**, метод, 386  
**multiply**, метод, 67  
**mutableCopy:**, метод, 424-425, 428

**N**

**navigation-based**, шаблоны приложений  
  iPhone, 462  
**NEXTSTEP**, среда разработки, разработка  
  Objective-C, 1  
**NSArray**, класс, методы, 366  
**NSCoding**, протокол, 440  
**NSCopying**, протокол, 452  
**NSCountedSet**, класс, 373  
**NSData**, класс, 383-384  
  нестандартные архивы, 447-450  
**NSDictionary**, класс, его методы, 369  
**NSFileHandle**, класс, 399-402  
  его методы, 378, 397-398  
**NSFileManager**, 380-382  
  defaultMessage, сообщения, 380  
  NSData, класс, 383-384  
  методы, 379, 384-385  
  напки  
    основные операции, 385-386  
    перечисление содержимого папки,  
      387-389  
**NSHomeDirectory**, функция, 391  
**NSKeyedArchiver**, класс, архивация с  
  помощью, 437-439  
**NSLog**, функция, 20, 327  
  % (процент), символ, 24  
**NSMutableArray**, класс, его методы, 366  
**NSMutableArray**, метод, 362  
**NSMutableDictionary** class, методы, 370  
**NSMutableSet**, класс, его методы, 374  
**NSMutableString**, класс, 328, 338, 340, 411  
**NSNumber**, класс, 322-324, 326  
**NSNumber**, объекты, 324-325, 408  
**NSObject**, класс, 157, 159  
**NSPathUtilities.h**, 389-391  
  **NSProcessInfo**, класс, 393-396  
  методы, 392-393  
  функции, 393  
**NSProcessInfo**, класс, 393-396  
**NSSet**, класс, его методы, 374

`NSString`, класс, 338-340, 410  
`NSTemporaryDirection`, функция, 391  
`Null-операторы`, 306, 559  
`numberOfDays`, функция, 277-278  
`numerator`, целая переменная экземпляра, 101

**O**

`Object`, класс, 197  
`objectAtIndex:`, метод, 342, 428  
`objectForKey:`, метод, 368  
**Objective-C**  
  веб-ресурсы, 576  
  как процедурный язык, 2  
  разработка, 1  
**ON ESTEP**, среда разработки, разработка  
  Objective-C, 1  
**OpenGL ES**, шаблоны приложений iPhone, 462  
`outlet`-переменные, 464

**P**

`pathComponents:`, метод, 392  
`pathExtension:`, метод, 391  
`perform`, метод, 197  
`plists`  
  архивация с помощью, 435-437  
  словари, создание из, 436  
`primes`, массив, выделение памяти, 344  
`print`, метод, 38, 41, 371, 373  
`printVar`, метод, 159

**Q**

`qsort`, функция, 301

**R**

`rangeOfString:`, метод, 333, 337  
`readDataOfLength:`, метод, 400  
`readDataToEndOfFile:`, метод, 400  
`Rectangle`, класс, 198-199  
`release`, метод, замещение, 179-180  
`release`, сообщения, управление памятью, 419  
`removeCard:`, метод, 359-362  
`removeFileAtPath:`, метод, 380  
`removeFileAtPath:handler:`, метод, 382, 397  
`removeObject:`, метод, 360-362, 373  
`removeObjectAtIndex:`, метод, 409  
`removeObjectIdenticalTo:`, метод, 360  
`replaceObjectAtIndex:withObject:`, метод, 429  
`restrict`, ключевое слово, 523-524  
`retainNameandReturn:`, метод, 434  
`return`, операторы, 559

**S**

`seekToEndOfFile:`, метод, 402  
`self`, ключевое слово, 149-150  
`setEmail:`, метод, 346  
`setName:`, метод, 346, 411-412  
`setNameandReturn:`, метод, 434  
`setNumerator:`, метод, 38, 41  
`setObject:forKey:`, метод, 368  
`setOrigin:`, метод, 172, 179  
`setStr:`, метод, 413-416  
`setString:`, метод, 337  
`setWithObject:`, метод, 373  
`short`, квалификатор, 53-54  
`sign`, функция, 111  
`signed char`, переменная, 222  
`signed`, квалификатор, 53-54  
`sizeof`, операторы, 307-308, 533  
`skipDescendants`, сообщения, 387  
`sort`, метод, 364  
`sortUsingSelector:`, метод, 362  
`Square`, класс, 198-199  
  как подкласс, 164  
`Square`, объект, 196  
`stringByAppendingPathComponent:`, метод, 391, 397  
`stringByAppendingString:`, метод, 330  
`stringByExpandingTildeInPath:`, метод, 392  
`stringByStandardizingPath:`, метод, 392  
`stringWithFormat:`, метод, 488  
`stringWithString:`, метод, 411, 413  
`substringFromIndex:`, метод, 332  
`substringToIndex:`, метод, 332  
`substringWithRange:`, метод, 333  
`super`, ключевое слово, замещение, 179-180  
`switch`, оператор, 120-123, 560

**T**

`Tab Bar`, шаблоны приложений iPhone, 462  
`typedef`, оператор, 205, 218-219, 523

**U**

`unichar`-символы, 326  
`union:`, метод, 373  
`unsigned`, квалификатор, 53-54  
`uppercaseString:`, метод, 330  
`utility`, шаблоны приложений iPhone, 462

**V**

`view-based`, шаблоны приложений iPhone, 462  
`volatile`-переменная, 214-215, 523-524

**W**

`while`, оператор, 89-93, 560  
**Window-based**, шаблоны приложений  
 iPhone, 462  
`writeToFile:atomically:`, сообщения  
 plist, архивы, 436  
 нестандартные архивы, 449

**X - Y - Z**

**Xcode**, 460  
 Build, меню, 15  
 компиляция на Маках, 10-16  
 приложение iPhone, создание, 461  
 интерфейс, проектирование, 467  
 код, ввод, 463  
**XYPoint**, класс, 166, 199  
**zone-аргументы**, 430

**A**

**Абсолютные имена пути**, 378  
**Абстрактные классы**, 183-184  
**Абстрактные протоколы**, 234  
**Автоматически высвобождаемые пулы**  
 (`autorelease pool`), 20, 323-326, 406, 418-419  
**Автоматические локальные переменные**, 213, 265  
**Адреса**, в памяти  
 оператор косвенного обращения  
 (`(indirection)`), 302  
 указатели, 301  
**Адресные карточки**, создание, 345-354  
**Адресные книги**  
 поиск имен в, 356-359  
 создание, 345-356  
 сортировка записей, 362-365  
 удаление имен из, 359-362  
**Адресный оператор**, указатели, 283  
**Алфавитный указатель**  
**Аргументы**  
`argc`, аргумент, функция `main`, 308  
`argv`, аргумент, функция `main`, 308  
**zone-аргументы**, 430  
 аргументы командной строки, 20, 308-310  
 дроби, передача как аргументов, 144  
 методы, 36-37  
 без имен аргументов, 143  
 локальные переменные, 147  
 несколько аргументов в, 141-143

параметры, 20

разделитель-запятая, 23  
 функции, 263-265

**Аргументы командной строки**, 20, 308-310

**Арифметика**, целочисленная, 58-60

оператор взятия по модулю, 60-61  
 оператор приведения типа, 63  
 преобразования, 62-63

**Арифметические операторы**, 529

бинарный, 56

битовые операторы, 67

дополнения до единицы, 70-71

левого сдвига, 71

побитовый AND, 68

побитовый включающего ИЛИ  
 (Inclusive-OR), 69

побитовый исключающего ИЛИ  
 (Exclusive-OR), 69

правого сдвига, 72

свойства ассоциативности, 56

старшинство, 56-58

унарный оператор «минус», 60

**Арифметические преобразования**, 538-539

**Архивация**

базовые типы данных, 445-446

нестандартные архивы, объект `NSData`, 447-450

определение, 435

`Foundation`, фреймворк, 317

архив с ключами, определение, 437

`NSKeyedArchiver`, класс, 437-439

объекты, 440

копированиe, 450-452

**Архивы с ключами**, определение, 437

**Б**

**Базовые типы данных**, 514-515

**Бинарные арифметические операторы**, 56

**Битовые операторы**, 67, 280, 530-531

дополнения до единицы, 70-71

левого сдвига, 71

побитовый AND, 68

побитовый включающего ИЛИ (Inclusive-OR), 69

побитовый исключающего ИЛИ  
 (Exclusive-OR), 69

правого сдвига, 72

**Битовые поля**, 280-282

**Блоки**

перечислимые типы данных, 218

оборудование, 83

**Блоки (unit)**, 282

Бред Кокс, 1

**Булевые переменные**, 123-126, 128

**Быстрое перечисление**, 354-356

**В**

Варианты циклов `for`, 88-89  
 Вывод из окна терминала, циклы `for`, 85-86  
 Ввод с клавиатуры, `for`, циклы, 84-86  
 Веб-ресурсы  
     Cocoa, 577  
     iPhone, 578  
     Objective-C, 576  
 Вещественные числа, 23  
 Вложенные операторы `if`, 109-111  
 Вложенные циклы `for`, 86-88  
 Внешняя глобальная переменная, 209  
 Внешняя переменная, 209-211  
     `extern`, ключевое слово, 209-212  
     `static`, ключевое слово, 211-213  
     определение, 209  
 Внуки (классов), 158  
 Возвращаемые значения, методы, 36  
 Восьмеричная форма записи, тип данных `int`, 50  
 Встроенные значения, 127  
 Выделение памяти, 166  
 Вызов функций, 544-545  
 Вызовы функций, 311  
 Вызовы, процедуры, 20  
 Выравнивание, 84  
 Выравнивание, по правому краю, 84  
 Выравнивание, по правому краю, 84  
 Выражения  
     `lvalue` (`l`-значения), 524  
     константные, 49, 528  
     операторы, 524-527  
     с сообщениями, 311, 555-556  
     составные,  
     формирование, 107

**Г**

Глобальная переменная, 209-211  
 Глобальные структуры определение, 276  
 Глубокая копия, 426-428, 451-452

**Д**

Двоеточие (:), условный оператор, 128  
 Двумерные массивы, 260  
     инициализация, 261-262  
     матрицы, 260  
     элементы, 260  
 Деархивация объектов, 440  
 Декодирование  
     базовые типы данных, 445-446  
     объекты, 440  
 Диапазоны для строк, 331-332  
 Диапазоны, для строк, 331-332

Диграфы, символы, 505  
 Динамический контроль типов, 187, 192, 195-196  
 Динамическое связывание, 183, 187, 191-193  
 Директивы компилятора, 507-508  
 Директивы, 507-508  
     `@package`, директива, 208  
     `@private`, директива, 208  
     `@protected`, директива, 208  
     `@protocol`, директива, 233  
     `@public`, директива, 208  
     `@selector`, директива, 197  
     переменные экземпляра, управление  
         областью действия, 208  
 Добавление знака к целым, 539  
 Домашние папки (~), 378  
 Драйверы устройств, иерархия уровней для приложения, 455  
 Дроби  
     аргументы, передача в виде дробей, 144  
     копирование, 429-432  
     сложение, 144-146  
     ссылка на объект `Fraction`, 145

**З**

Завершающий нуль-символ, 260  
 Зависимость от реализации, 50  
 Зависимость от регистра букв, соглашения по именованию, 34  
 Замещение методов, 175-176  
     `dealloc`, метод, 179  
     `release`, метод, 179-180  
     `super`, ключевое слово, 179-180  
     категории, подклассы, 230  
 За занятые, в аргументах, 23  
 Заранее определенные идентификаторы, 509, 567  
 Заранее скомпилированные заголовочные файлы, 321  
 Зарезервированные имена/слова (соглашения по именованию), 34  
 Знак (#), препроцессор, 239  
 Значения  
     возврат, 265-266  
     возвращаемые, методы, 36  
     встроенные, 127  
     переменные, вывод значений, 22-24  
     состояние 0, FALSE или off, 125-126  
     состояние 1, TRUE или on, 126  
     состояние 1, TRUE или оп, 125  
 Значения со знаком, символы как, 222

**И**

Идентификаторы, 506

- директивы, 507-508  
заранее определенные, 509  
идентификатор перечисления, 215-218  
имена универсальных символов, 506  
ключевые слова, 506-507  
Идентичные объекты, 360  
Иерархия уровней для приложения, 455  
Имена  
адресные книги  
поиск имен в, 356-359  
сортировка в, 362-365  
удаление из, 359-362  
файлы, 12  
Имена путей доступа к файлу  
`NSPathUtilities.h`, 389-391  
`NSProcessInfo`, класс, 393-396  
методы, 392-393  
функции, 393  
абсолютные имена пути, 378  
относительные имена пути, 378  
Имена универсальных символов, 506  
Именованные пары объект/категория, 231  
Именованные пары, объект/категория, 231  
Индекс (порядковый номер), 256  
Инициализаторы, назначенные, 206  
Инициализация  
двумерные массивы, 261-262  
классы, 205-207  
массив символов, указатели, 296  
методы, 206  
переменные экземпляра, 206  
структуры, 277  
элементы массивов, 258-259  
Инкапсуляции данных, 44-47  
Интерфейс, проектирование для проекта приложения iPhone, 467-468, 471-472, 476
- К**
- Кавычки, локальные файлы, 137  
Калькулятор дробей, создание для iPhone, 476-480  
`Calculator`, класс, 488-489  
`Fraction`, класс, 485-488  
контроллер представлений, определение, 480-485  
пользовательский интерфейс, проектирование, 490  
Каталоги. См. Папки (каталоги)  
Категории, 165, 230  
допустимое число, 230  
замещение методов, 230  
именованные пары объект/категория, 231  
мастер-файл определения класса, 230  
методы
- добавление, 230  
замещение, подклассы, 230  
определение, 227-230  
определение, 225-226, 551-552  
переменные экземпляра, 230  
подклассы, 230  
протоколы, принятие, 234  
Квалификаторы, типов данных, 53-54  
Класс памяти, 539-540  
Классы  
`AddressBook`  
файл секции `implementation`, 572-574  
файл секции `interface`, 570  
`AddressCard`  
файл секции `implementation`, 571  
файл секции `interface`, 569  
`Calculator`, 115  
операторы присваивания, 65-67  
`Complex`, 187-190, 195  
copy, метод, добавление, 429-432  
`Foundation`, 250  
`Fraction`, 30-32, 39, 101, 157, 195-197  
`NSObject`, 157-159  
`Object`, 197  
`Rectangle`, 198-199  
`Square`, 198-199  
`XYPoint`, 199  
абстрактные классы, 183-184  
владение объектами, 171-174  
внуки, 158  
вопросы о, 195-200  
выражения с сообщениями, 555-556  
инициализация, 205-207  
категории, 165  
определение, 551-552  
корневые классы, 157-158, 161  
методы, 28, 211  
добавление, 162-174  
наследование, расширение классов, 162-174  
объявления, 133-138  
операции с объектами, 310-311  
определение класса, 549-550  
определение объекта, 554-555  
определение протокола, 552-553  
определение, 30-32, 546  
объявления методов, 547-549  
объявления свойств, 546-547  
переменные экземпляра, 546  
расширение, 155-156  
секция `interface`, 546  
подклассы, 157  
альтернативы, 235  
получатели, 28  
расширение, наследование, 162-174  
родительские классы, 33

сообщения, 28  
**Ключевые слова**, 506-507  
 const, 214, 523-524  
 enum, 215-218  
 restrict, 523-524  
 volatile, 523-524  
**Код, комментарии**, 14, 18-22  
**Кодирование**  
 базовые типы данных, 445-446  
 объекты, 440  
**Команды**, gcc, 17  
**Комментарии**, 14, 18-22, 509  
 Xcode, 14  
 отладка, 19  
**Компиляторы**, файлы секции interface, 136  
**Компиляция**, Маки и  
 Xcode, 10-16  
 окно Terminal, 16-18  
**Конкатенация**, символьные строковые константы, 512  
**Константные выражения**, 49, 528  
**Константы с плавающей точкой**, 51, 510  
**Константы с плавающей точкой**, шестнадцатеричные, 51  
**Константы**, 49  
 константы перечислимого типа, 513  
 с плавающей точкой, 51, 510  
 символьные строковые константы, 512-513  
 целого типа, 510  
**Конструкции языка**  
 else if, оператор, 111-114, 117-120  
 if, оператор, 99-104  
 if-else, оператор, 104-106  
**Контроллер представлений**, определение для проекта калькулятора дробей для iPhone, 480-485  
**Координаты**, класс XYPoint, 166  
**Копирование**. См. также сору, метод глубокая копия, 426-428  
 дроби, 429-432  
 мутабельная копия, 424-425  
 объекты, 423-425  
 глубокие копии, 451-452  
 метод-получатель, 432-434  
 методы-установщики, 432-434  
 через archiver, 450-452  
 поверхностная копия, 426-428  
 файлы, NSProcessInfo, класс, 393-396  
**Корневые классы**, 157-158, 161  
**Корневые папки**, 378  
**Корневые службы**, иерархия уровней для приложения, 455  
**Косвенное обращение**, указатели, 283  
**Круглые скобки** вокруг условных операторов, 128

**Л**

Линейные массивы, 260  
**Литералы, составные**, 305, 538  
**Логические операторы правого сдвига**, 72  
**Логические операторы**, 529  
**Логический оператор AND (&&)**, 107  
**Логический оператор OR (||)**, 107  
**Логический оператор отрицания**, 126  
**Локальная структура**, определение, 276  
**Локальные переменные**, 146-147  
 auto, ключевое слово, 213  
 static, ключевое слово, 147-149, 265  
 аргументы для метода, 147  
 функции, 263, 265  
**Локальные файлы**, кавычки, 137

**М**

**Маки**  
 Cocoa, среда разработки, разработка Objective-C, 1  
 iPhone, разработка в Objective-C, 2  
**компиляция и**  
 Xcode, 10-16  
 окно Terminal, 16-18  
**Макросы**, 244-245  
**Макросы**, 244-246  
 #, оператор, 246  
 ##, оператор, 247  
 #define, операторы, 244-245  
**Мантисса**, значения с плавающей точкой, 51  
**Массивы переменной длины**, 517  
**Массивы символов**, 259-260  
 завершающий нуль-символ, 260  
 указатели, 296  
**Массивы**, 241, 257  
 двумерные массивы, 260-262  
 линейные массивы, 260  
 массивы символов, 259-260  
 методы, передача, 268-269  
 многомерные массивы, 260-262, 517-518  
 мутабельные массивы, 341  
 немутабельные массивы, 341  
 объявление, 257  
 одномерные массивы, 260  
 одномерные, 516  
 операторы #define, 241  
 операторы, 534  
 определение, 256-258  
 объединения, 303  
 число элементов, 259  
 переменной длины, 517  
 работа с, 256-258  
 сортировка, 362, 364-365  
 структур, 278

- 
- указатели на, 290-293, 536-537  
функции, передача, 268-269  
элементы массивов, 256-260  
элементы, передача для функций/  
методов, 268-269
- Матрицы**, 260
- Машинно-зависимый диапазон**, 50
- Метки**, 306
- Методы декодирования, запись**, 440-445
- Методы доступа (accessor method)**, 139-140
- Методы завода**, 29
- Методы класса**, 29, 35-36
- Методы кодирования, запись**, 440-445
- Методы экземпляра**, 35-36
- Методы**, 33
- conformToProtocol, протоколы, 233
  - convertToNum, 101-104
  - copy, метод, немутабельные объекты, 433
  - doesNotRecognize:, метод, 198
  - perform, метод, 197
  - setOrigin:, метод, 172, 179
  - аргументы, 36-37
    - без имен аргументов, 143
    - локальные переменные, 147
      - несколько, 141-143
    - возвращаемые значения, 36
    - добавление в классы, 162-174
    - замещение, 175-176
      - dealloc, метод, 179
      - release, методы, 179-180
      - подклассы для категорий, 230
    - как функции, 311
    - категории, определение, 227-230
    - массивы, передача, 268-269
    - методы декодирования, запись, 440-445
    - методы завода, 29
    - методы инициализации, 206
    - методы класса, 29, 35-36, 211-213
    - методы кодирования, запись, 440-445
    - методы экземпляра, 28-29, 35-36
    - методы-получатели, 46, 432-434
    - методы-установщики (setter), 46, 432-434
    - наследование, 161
    - наследуемые методы, 161
    - объекты, выделение памяти/возврат, 150-154
    - объявления, 37, 547-549
    - определение, 37
      - класса, 549-550
      - протоколы, 232
    - определение, какой метод выбирается, 177-179
    - синтезируемые методы доступа, 550
    - указатели, возврат как результата, 288-289
    - элемент многомерного массива, передача, 270
- Методы-получатели (getter)**, 46, 432-434
- Методы-установщики (setter)**, 46, 432-434
- Многобайтные символы, символьные строковые константы**, 512
- Многомерные массивы**, 260-262, 270, 517-518
- Модификаторы типа, определения протоколов**, 553
- Модуль, определение**, 207
- Мутабельные массивы**, 341
- Мутабельные объекты**, 328-332
- Мутабельные словари**, 367
- Мутабельные строки**, 333-337, 426-427
- Н**
- Назначенный инициализатор**, 206
- Наследование**
- корневые классы, 157-158, 161
  - методы, 161
    - добавление в классы, 162-174
    - замещение, 175-176, 179
    - определение, какой метод выбирается, 177-179
    - переменные экземпляра, 181-183, 208
    - расширение классов, 162-174
  - наследуемые методы, 161
- Немутабельные массивы**, 341
- Немутабельные объекты**, 328-332
- Немутабельные словари**, 367
- Немутабельные строки**, 426-427
- Нестандартные архивы**
- NSData, объект, 447-450
  - writeToFile:atomically:, сообщения, 449
- Неформальные протоколы**, 234-235
- Нуль-символ**, 260
- О**
- Область действия**, 539-540
- переменные экземпляра, директивы, 208
  - переменные, 207
- Область хранения данных**, 383
- Обобщенный тип указателя, тип id**, 311
- Обработка исключений**, 200-202, 561
- Объединения**, 302-304, 520-521
- Объектно-ориентированное программирование**, 28, 282
- @implementation, секция, 32, 37-38
  - @interface, секция, 32
    - выбор имени, 33-34
    - методы, 33-37
    - переменные экземпляра, 33-35
    - родительские классы, 33
  - классы
    - дебаги, 30-32
    - определение, 30-32
    - получатели, 28

- сообщения, 28
- методы, 28-29
- объекты, определение, 27
- программная секция, 32, 38-44
- экземпляры, 27-29
- Объекты**
  - `id`, объект, объявление, 554-555
  - `Square`, объект, 196
  - классы, владение, 171-174
  - копирование, 423-425
    - глубокие копии, 426-428, 451-452
    - методы-получатели, 432-434
    - методы-установщики (`setter`), 432-434
    - поверхностное копирование, 426-428
    - с помощью архиватора, 450-452
  - методы, выделение памяти/возврат, 150-154
  - метод `copy`, 433
  - мутабельное копирование, 424-425
  - мутабельные объекты, 328-332
  - немутабельные объекты, 328-332
  - немутабельные объекты, 328-333
  - объекты константных символьных строк, 513
  - объекты-массивы, 341-344
    - создание адресных карточек, 345-356
    - создание адресных книг, 345-365
  - объекты-наборы, 370-374
  - объекты-словари, 367-369
  - определение, 27, 554-555
  - освобождение памяти, 338-340
  - присваивание, 423, 428
  - составные объекты, 235-236
  - состояние, 28
  - строковые объекты, 326-340
  - счетчики ссылок, 407-409
  - числовые объекты, 322-326
- Объекты константных символьных строк**, 296, 327
- Объекты-массивы**, 341-344, 362
  - адресные карточки, создание, 345-356
  - адресные книги, создание, 345-365
- Объекты-наборы**, 370-374
- Объекты-словари**, 367-369
- Объявление прототипа**, 267
- Объявление типа возвращаемого значения**, функции, 267-268
- Объявление типов аргументов**, 267-268
- Объявления**, 513
  - `@interface`, секция, 133-134
  - методов, 37
  - файлы секции `implementation`, 133-138
  - файлы секции `interface`, 133-138
- Обычные арифметические преобразования**, 538-539
- Одномерные массивы**, 260
- Одномерные массивы**, 516
- Окно Terminal, компиляция на Маках**, 16-18
- Оператор «точка»**, 271
  - свойства, доступ с помощью, 140
- Оператор OR (`||`)**, 107
- Оператор взятия по модулю**, 60-61, 104
- Оператор включающего ИЛИ (Inclusive-OR)** (побитовый), 69
- Оператор дополнения до единицы**, 70-71
- Оператор исключающего ИЛИ (Exclusive-OR)** (побитовый), 69
- Оператор косвенного обращения (\*)**, 284
  - указатели, адрес памяти, 302
- Оператор левого сдвига**, 71
- Оператор отрицания**, 126
- Оператор правого сдвига**, 72
- Оператор приращения (инкремента) (++)**, 83, 292, 296-299, 531
- Оператор уменьшения, декремента (-)**, 83, 292, 296-299, 531
- Операторы «запятая»**, 306-307, 533
- Операторы (предложения)**, 20, 556
  - `#define`, оператор, 239-241
    - аргументы, 243
    - константные значения, 241
    - макросы, 244-245
    - местоположение, 240
    - обратный слэш (\), символ, 243
    - определенные имена, 241-245
    - синтаксис, 240
  - `#elif`, оператор препроцессора, 252
  - `#else`, 250-251
  - `#endif`, оператор, 250-251
  - `#if`, оператор препроцессора, 252
  - `#ifdef`, оператор, 250-251
  - `#ifndef`, оператор, 250-251
  - `#import`, оператор, 247-250
  - `#include`, оператор, 247-250
  - `#undef`, оператор, 253
  - `break`, оператор, 95, 121, 123
  - `continue`, оператор, 96
  - `do`, циклы, 94-95
  - `else if`, оператор, 111-114, 117-120
  - `for`, циклы, 78-89
  - `goto`, оператор, 306
  - `if`, оператор, 99-104
    - вложенные, 109-111
    - программные операторы, 99
  - `if-else`, оператор, 104-106
  - `null-оператор`, 306
  - `switch`, оператор, 120, 123
    - `if`, оператор, преобразование в, 121
  - `typedef`, оператор, 205, 218-219
  - `while`, циклы, 89-93
  - блоки, 83
  - отладочные операторы, 251
  - программный оператор, оператор `if`, 99
  - составные, 556

- Операторы**, 524-527  
 &, указатели, 283  
`sizeof`, оператор, 307-308, 533  
 адресный, указатели, 283  
 арифметический, 529  
     бинарные, 56  
     битовые, 67-72  
     приведения типа, 63  
     свойства ассоциативности, 56  
     старшинство, 56-58  
 битовый, 280  
 взятия по модулю, 60-61, 104  
 запятая, 306-307, 533  
 косвенного обращения (\*), 284  
 логические, 529  
 логический AND (&&), 107  
 логический OR (||), 107  
 логический отрицания, 126  
 массивы, 534  
 побитовые, 530-531  
 пост-оператор приращения, 297-298  
 пред-оператор приращения, 297-298  
 приведения типов, 532  
 приращения (инкремента) (++), 83, 292,  
     296-299, 531  
 присваивания, 64-67, 531  
 пробелы, 109  
 сравнения, 80, 530  
 структуры, 534-535  
 считывания в символьную переменную, 117  
 тернарный, 128  
 точка, 271  
 указатели, 535-537  
 уменьшения (декремента) (—), 83, 292,  
     296-299, 531  
 унарный «минус», 126  
 унарный, указатели, 283  
 условный, 128-129, 532
- Операции**  
     приращения, 297  
     с указателями, 300-301
- Определения имен**, 241-245
- Определенные имена**, 241-245
- Отбрасывание дробной части чисел**, 62-63
- Открытие файлов**, 378
- Отладка**  
     комментарии, 19  
     операторы, 251
- Относительные имена пути**, 378
- Ошибка программирования**, 118-119
- П**
- Память**, 405  
     автоматически высвобождаемые пулы, 20,  
         323-326, 406
- пример, 418-419  
 адреса, 301-302  
 выделение, 166  
 высвобождение объектов, 338-340  
 использование, 301  
 правила управления, сводка, 419-420  
 сборка мусора, 420  
 счетчики ссылок  
     объекты, 407-409  
     переменные экземпляра, 411-417  
     строки, 409-411, 423  
 утечка, 153  
 ячейки, 301
- Папки (каталоги)**  
     домашние папки (~), 378  
     корневые папки, 378  
     содержимое, перечисление, 387-389  
     управление с помощью NSFileManager,  
         380-382, 405  
         defaultMessage, сообщения, 380  
         NSData, класс, 383-384  
         методы, 379, 384-385  
         основные операции с папками, 385-386  
         перечисление содержимого папки,  
             387-389
- Переменная цикла**, 125
- Переменные**  
     charPtr, переменная, 286  
     dataValue, переменная, 192  
     intPtr, 283, 285  
     signed char, переменная, 222  
     булевы переменные, 123-128  
     внешняя глобальная переменная, 209  
     внешняя переменная, 209-213  
     глобальная переменная, 209-211  
     значения, вывод, 22-24  
     класс памяти, 540  
     ключевые слова  
         auto, 213  
         const, 214  
         extern, 209-212  
         static, 211-213  
         volatile, 214-215  
     локальные, 263-265  
     область действия, 207  
     определение структуры, 280  
     переменная цикла, 125  
     переменная-структура, 271, 333  
     переменные для свойств, объявления,  
         546-547  
     переменные экземпляра, 33-35, 44, 541-  
         542  
     доступ, 44-47  
     инициализация, 206  
     категории, 230  
     наследуемые, 208

- объявления, 546
- получатели, 46
- расширение классов, 181-183
- счетчики ссылок, 411-417
- управление областью действия, 208
- установщики, 46
- переменные-объекты, как переменные-указатели, 311
- символьные переменные, без знака, 222
- целые переменные экземпляра, 101
- Переменные для свойств, объявления, 546-547
- Переменные экземпляра, 33, 35, 44, 541-542
  - в структурах, 310-311
  - доступ, 44-47
  - инициализация, 206
  - категории, 230
  - наследуемые, 208
  - область действия, 208
  - объявления, 546
  - получатели, 46
  - расширение классов путем наследования, 181-183
  - счетчики ссылок, 411-417
  - установщики, 46
- Переменные-объекты, как переменные-указатели, 311
- Переменные-структуры, 271, 333
- Пересылка, 162
- Перечислимый тип
  - идентификатор, 215-218
  - константы, 513
  - словари, 368-369
  - содержимое каталога, 387-389
- Перечислимый тип данных, 522-523
  - enum, ключевое слово, 215-218
  - блоки, 218
  - определение, 205
- Поверхностная копия, 426-428
- Подклассы для категорий, 230
- Подклассы, 157
  - альтернативы подклассам, 235
  - назначенный инициализатор, 206
  - создание, 164
- Подчинение протоколам, 231
- Полиморфизм, 187-192
- Положительные целые числа, простые числа, 123
- Получатели, 28
- Пользовательский интерфейс, проектирование для проекта калькулятора приложения iPhone, 490
- Поля, битовые поля, 282
- Порядковый номер (иншекс), 256
- Порядок, значения с плавающей точкой, 51
- Пост-оператор приращения, 297-298
- Предложения, else, 110-111
- Пред-оператор приращения, 297-298
- Преобразования
  - арифметические, 538-539
  - типы данных, 220-222, 538-539
  - целые числа, 62-63
  - числа с плавающей точкой, 62-63
- Преобразования типов данных, 220-222
- Препроцессоры
  - директивы, 562
    - #, 567
    - #define, 239-241, 562-564
    - #error, 564
    - #if, 564-565
    - #ifdef, 565
    - #ifndef, 565
    - #import, 247-250, 565-566
    - #include, 247-250, 566
    - #line, 566
    - #pragma, 566
    - #undef, 567
  - заранее определенные идентификаторы, 567
  - знак (#), 239
  - определение, 239, 242
  - триграмммы, последовательности, 561
  - условная компиляция, 250-251
- Приложения, iPhone, 461
  - калькулятор для дробей, создание, 476-490
- Принятие (adopt) протоколов, 231
- Пробелы, операторы, 109
- Проверка во время выполнения, 193
- Проверка во время компиляции, 193
- Проверка, составная операция сравнения, 106-109
- Программирование. См. Объектно-ориентированное программирование
- Программная секция (объектно-ориентированное программирование), 32, 38-44
- Программные операторы, оператор if, 99
- Проектирование интерфейса для проекта приложения iPhone, 467-468, 471-472, 476
- Производные типы данных, 516
  - массивы
    - многомерные, 517-518
    - одномерные, 516
    - переменной длины, 517
  - объединения, 520-521
  - структуры, 518-520
  - указатели, 521-522
- Простые числа, 123
- Протоколы
  - @protocol, директива, 233

- F**
- Foundation, фреймворк, 231
  - абстрактные протоколы, 234
  - методы, 232-233
  - неформальные, 234-235
  - определение, 231, 552-553
  - подчинение протоколам, 231
  - принятие категорией, 234
  - принятие, 231-232
- Процедурный язык, Objective-C как, 2
- Процедуры. См. также Функции; Методы
- вызовы, 20
  - процедуры-получатели, 432-434
- Процент (%), символ, 24
- P**
- Расширение для знака, 221-222
  - Расширение классов, наследование, 162-174, 181-183
  - Расширения имен файлов, 12
  - Расширения, имена файлов, 12
  - Результаты, возврат, 265-266
  - Ритчи, Денис, 1
  - Родительские классы, 33
- C**
- Сборка мусора (управление памятью), 420
  - Свойства ассоциативности, арифметические операторы, 56
  - Свойства, доступ с помощью оператора «точка», 140
  - Связывание, динамическое, 183, 187
  - Связь Objective-C с языком C, 310
  - Секция `implementation`, определение класса, 549
  - Секция `interface`, определение класса, 546
  - Селекторы, 197
  - Символы
    - завершающий нуль-символ, 260
    - как значения со знаком, 222
    - новой строки, 22
    - формат хранения символов, ASCII, 114
  - Символьная переменная operator,
    - считывание операторов в, 117
  - Символьная переменная без знака, 222
  - Символьные константы из расширенных наборов, 512
  - Символьные константы, 511-512
  - Символьные переменные, без знака, 222
  - Символьные строки, 260
    - указатели на, 294-295
  - Символьные строковые константы, 512-513
  - Синтаксис, структуры, 271
  - Синтезируемые методы доступа, 139-140, 550
- S**
- Словари
    - plists, создание из, 436
    - мутабельные, 367
    - немутабельные, 367
    - перечислимого типа, 368-369
    - словарь атрибутов, 380
  - Словарь атрибутов, 380
  - Службы приложений, уровень, иерархия уровней для приложения, 455
  - Слэши (/ /), комментарии, 19
  - Соглашения по именованию, 33-34
  - Создание проекта приложения iPhone, 461, 463
    - интерфейс, проектирование, 467-468, 471-472, 476
    - код, ввод, 463, 466
  - Сообщения, 28, 196
  - Сортировка массивов, 362-365
  - Составная операция сравнения, 106-109
  - Составные выражения сравнения, формирование, 107
  - Составные литералы, 305, 538
  - Составные объекты, 235-236
  - Составные операторы, 556
  - Состояние (объекты), 28
  - Специальный тип BOOL, 127
  - Списки свойств (plists), 435
  - Ссылки, на элементы массива, 256
  - Статическая переменная, 211-213
  - Статические локальные переменные, 147-149, 265
  - Статические функции, 268
  - Статический контроль типов, 194
  - Строки
    - мутабельные строки, 333-337, 426-427
    - немутабельные строки, 426-427
    - символьные строки, 260
    - символьные строковые константы, 512-513
    - строки формата, 25
    - счетчики ссылок, 409-411, 423
  - Строковые константы из расширенных наборов, 513
  - Строковые объекты, 326-340
  - Структуры, 271, 273-275, 277, 280, 518-520
    - битовые поля, 281-282
    - глобальные структуры, определение, 276
    - инициализация, 277
    - классы, 276
    - компоненты, 310
    - локальная структура, определение, 276
    - массивы структур, 278
    - операторы, 534-535
    - определение, 278-280
    - переменные экземпляра, хранение в, 310-311

- переменные, 271-273  
синтаксис, 271  
структуры в структурах, 278-279  
указатели, 287-288, 537
- Сумматор**, 65
- Счетчики ссылок**  
объекты, 407-409  
переменные экземпляра, 411-417  
строки, 409-411, 423
- Счетчики удержаний (retain)**, 426
- Т**
- Таблицы**  
вызовов, 301  
истинности, 68
- Тернарный оператор**, 128
- Тильда (~)**, домашняя папка, 378
- Типы**  
BOOL (специальный тип), 127  
динамический контроль типов, 187, 192, 195-196  
статический контроль типов, 194
- Типы данных**, 49  
Bool, 73  
char, 49, 51-53, 222  
Complex, 73  
double, 49, 51  
float, 49, 51  
id, 55, 191-194  
Imaginary, 73  
int, 49-50  
базовые типы данных, 55, 514-515  
квалификаторы, 53-55  
модификаторы, 523-524  
перечислимый тип данных, 205, 522-523  
ключевое слово enum, 215-218  
преобразования, 538-539  
производные типы данных, 516  
 массивы, 516-518  
 объединения, 520-521  
 структуры, 518-520  
 указатели, 521-522  
 размеры хранения данных, 50
- Точка с запятой (;)**, 21
- Точки начала прямоугольника**, сохранение в виде отдельных значений, 166
- Треугольные числа**, 77
- Триграммы**, 561
- У**
- Удаление имен из адресных книг**, 359-362
- Указатели**, 39, 283-284, 521-522  
&, оператор., 283  
charPtr, переменная, 286
- intPtr, переменная, 283, 285  
адреса памяти, 301-302  
адресный оператор, 283  
константные символьные строки, 296  
косвенное обращение, 283  
массивы, 290-293, 536-537  
массивы символов, 296  
методы, передача, 288-289  
оператор приращения (инкремента) (++), 292, 296-299  
оператор уменьшения (декремента) (--), 292, 296-299  
операторы, 535-537  
операции, 300-301  
определение, 283  
передача как аргументов, 288-289  
пост-оператор приращения, 297-298  
пред-оператор приращения, 297-298  
символьные строки, 294-295  
структуры, 287-288, 537  
типы, тип id, 311  
указатели на функции, 545  
унарный оператор, 283  
функции, 300-301  
 передача, 288-289  
 таблицы вызовов, 301
- Унарный оператор «минус»**, 60, 126
- Унарный оператор**, указатели, 283
- Условие удовлетворяется**, операторы if, 125
- Условия цикла**, 79
- Условная компиляция**, 250-251
- Условный оператор**, 128-129, 532
- Установка SDK iPhone**, 459
- Ф**
- Файлы**  
class.h, 133-134  
include-файлы, 250  
NSFileHandle, класс, 399-402  
 методы, 378, 397-398  
имена пути доступа к файлу, 378  
NSPathUtilities.h, 389-396  
именование, 12  
копирование, класс NSProcessInfo, 393-396  
 мастер-файл определения класса, категории в, 230  
 основные операции с файлами, 397-402  
открытие, 378  
 управление с помощью NSFileManager, 380-382, 405  
defaultMessage, сообщения, 380  
NSData, класс, 383-384  
 методы, 379, 384-385  
 основные операции с папками, 385-386

- 
- перечисление содержимого папки, 387-389
- Файлы секции implementation**  
AddressBook, класс, 572-574  
AddressCard, класс, 571  
Fraction.m, 136
- Файлы секции interface**, 134  
AddressBook, класс, 570  
AddressCard, класс, 569  
компиляторы, 136  
расширение, 155-156
- Фреймворки**, определение, 3
- Функции**, 543  
main, 262-263, 308  
printMessage, 262-263  
qsort, функция, 301  
sign, 111  
аргументы, 263-265  
вызов, 544-545  
значения, возврат, 265-266  
класс памяти, 540  
локальные переменные, 263-265  
массивы, передача, 268-269  
определения, 543-544  
прототипы, объявление, 267  
результаты, возврат, 265-266  
статические функции, 268  
типы аргументов, объявление, 267-268  
типы возвращаемых значений,  
объявление, 267-268  
указатели, 300-301, 545  
возврат как результата, 288-289  
как аргумент, 301  
таблицы вызовов, 301
- указатель на массив, передача, 293-294
- элемент многомерного массива, передача, 270
- Х**
- Хранение (данных), 383
- Ц**
- Целочисленная арифметика, 58-60  
оператор взятия по модулю, 60-61
- оператор приведения типа, 63  
преобразования, 62-63
- Целые переменные экземпляра, 101
- Целые числа, 510
- Циклы**  
break, оператор, 95  
continue, оператор, 96  
do, циклы, 94-95  
do-while, циклы, 344  
for, циклы, 78-89  
варианты, 88-89  
ввод из окна терминала, 85-86  
ввод с клавиатуры, 84-86  
вложенные, 86-88  
порядок выполнения, 81  
while, циклы, 89-93
- Ч**
- Числа**  
вещественные числа, 23  
индекс (порядковый номер), как ссылка, 256  
отбрасывание дробной части чисел, 62-63  
с плавающей точкой, 23, 62-63  
треугольные, 77
- Числа Фибоначчи, 257-258
- Числовые объекты**, 322-326
- Ш**
- Шаблоны (приложений iPhone), 461
- Шестнадцатеричные значения, константы с  
плавающей точкой, 51
- Э**
- Экземпляры  
методы, 28-29  
определение, 27
- Экспоненциальное представление, 51
- Элементы массивов, 256
- Я**
- Ячейки, память, 301